



Universitat Oberta
de Catalunya

Bases de JavaScript

Front end web developer



Finançat per
la Unió Europea



Next Generation
Catalunya



SOC / Servei d'Ocupació
de Catalunya

- ❖ Introducción 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 - interacción JS con HTML y CSS (291)

Introducción

```
),removeSelf,a.fn.scrollspy=d,this},a(window).on("load.bs.  
y),+function(a){use strict;function b(b){return this.each(function()  
&e[b]())}var c=function(b){this.element=a(b)};c.VERSION="3.3.7",c.1  
dropdown-menu"),d=b.data("target");if(d||(d=b.attr("href")),d=d&&d.re  
st a),f=a.Event("hide.bs.tab",{relatedTarget:b[0]}),g=a.Event("sh  
faultPrevented()){var h=a(d);this.activate(b.closest("li"),c),this.  
trigger({type:"shown.bs.tab",relatedTarget:e[0]}))}}},c.prototype.  
u>.active).removeClass("active").end().find('[data-toggle="tab"]  
aria-expanded",!0),h?(b[0].offsetWidth,b.addClass("in")):b.removeClass()  
().find('[data-toggle="tab"]').attr("aria-expanded",!0),e&&e()}{var  
de)||!!d.find(">.fade").length);g.length&&h?g.one("bsTransitionE  
;var d=a.fn.tab;a.fn.tab=b,a.fn.tab.Constructor=c,a.fn.tab.noConfi  
"show");a(document).on("click.bs.tab.data-api",'[data-toggle="tab"]  
use strict';function b(b){return this.each(function(){var d=a(this)  
=typeof b&&e[b]())}var c=function(b,d){this.options=a.extend({},  
",a.proxy(this.checkPosition,this)).on("click.bs.affix.data-api",  
null,this.pinnedOffset=null,this.checkPosition());c.VERSION="3.3.  
tState=function(a,b,c,d){var e=this.$target.scrollTop(),f=this.$e  
"bottom"==this.affixed) return null!=c?!(e+this.unpin<=f.top)&&"b  
!&e=c?"top":null!=d&&i+j>=a-d&&"bottom"},c.prototype.getPin  
.RESET).addClass("affix");var a=this.$target.scrollTop(),b=this.  
WithEventLoop=function(){setTimeout(a.proxy(this.checkPosition,t  
ent.height(),d=this.options.offset,e=d.top,f=d.bottom,  
peof e&&(e=d.top(this.$element)),"c  
ent.css("top","","",
```

¿Sabías que...?

El pájaro del logo de Twitter se llama Larry



HTML + CSS = maquetación

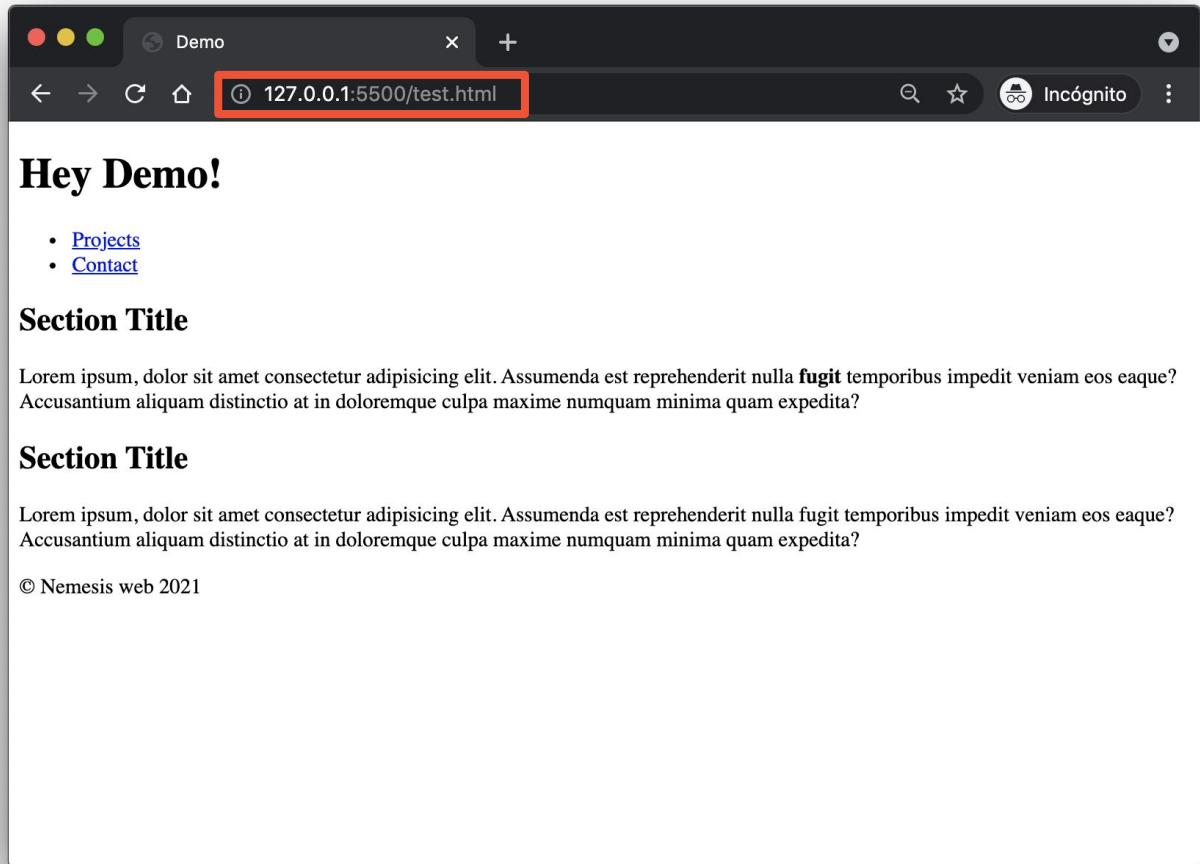
HTML + CSS + JS = front-end

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.



A screenshot of a web browser window titled "Demo". The address bar shows the URL "127.0.0.1:5500/test.html", which is highlighted with a red box. The page content includes:

- A heading "Hey Demo!"
- A list of links:
 - [Projects](#)
 - [Contact](#)
- A section titled "Section Title" containing placeholder text.
- Another section titled "Section Title" containing placeholder text.
- A copyright notice at the bottom: "© Nemesis web 2021"

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](#)

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 directamente 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>
```

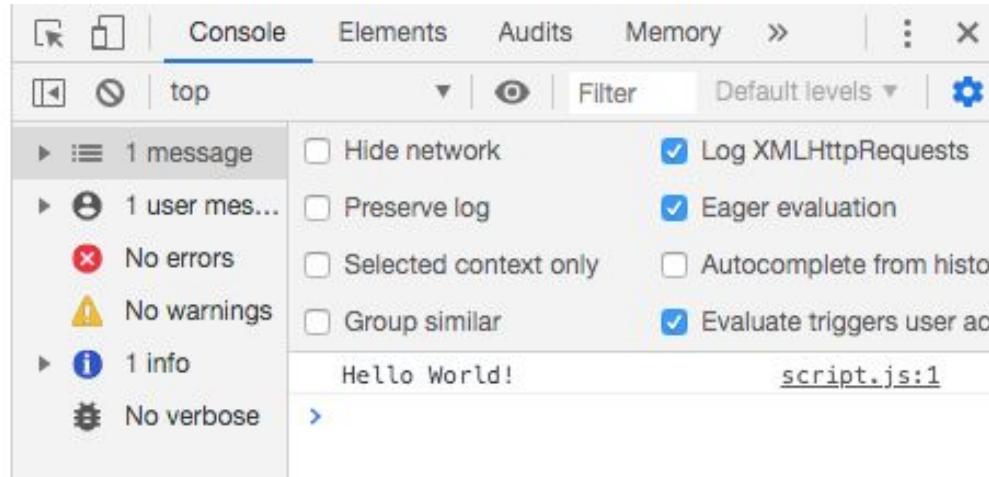
Si ejecutamos el código anterior en el navegador veremos que sólo aparece el texto que tenemos en:

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

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

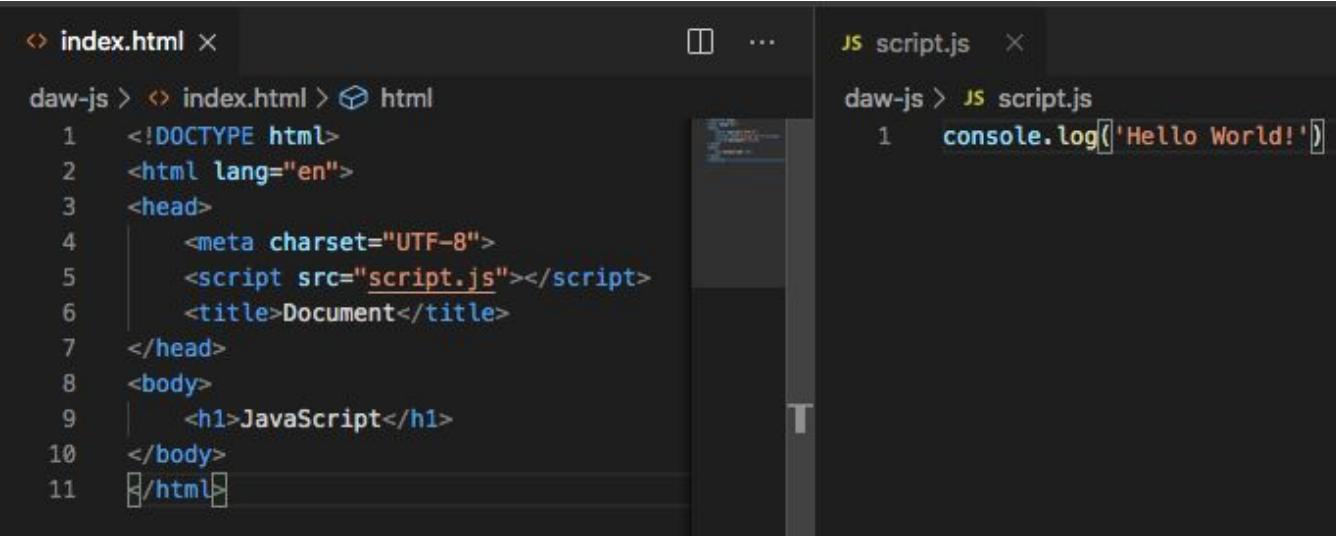
```
console.log('Hello World!');
```

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



The image shows a code editor interface with two files open:

- index.html**:
A simple HTML document structure. It includes a DOCTYPE declaration, an HTML element with lang="en", a head section containing a meta charset="UTF-8" and a script element linking to "script.js", and a body section with a single h1 element containing the text "JavaScript".

```
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>
```
- script.js**:
A JavaScript file containing a single line of code that logs "Hello World!" to the console.

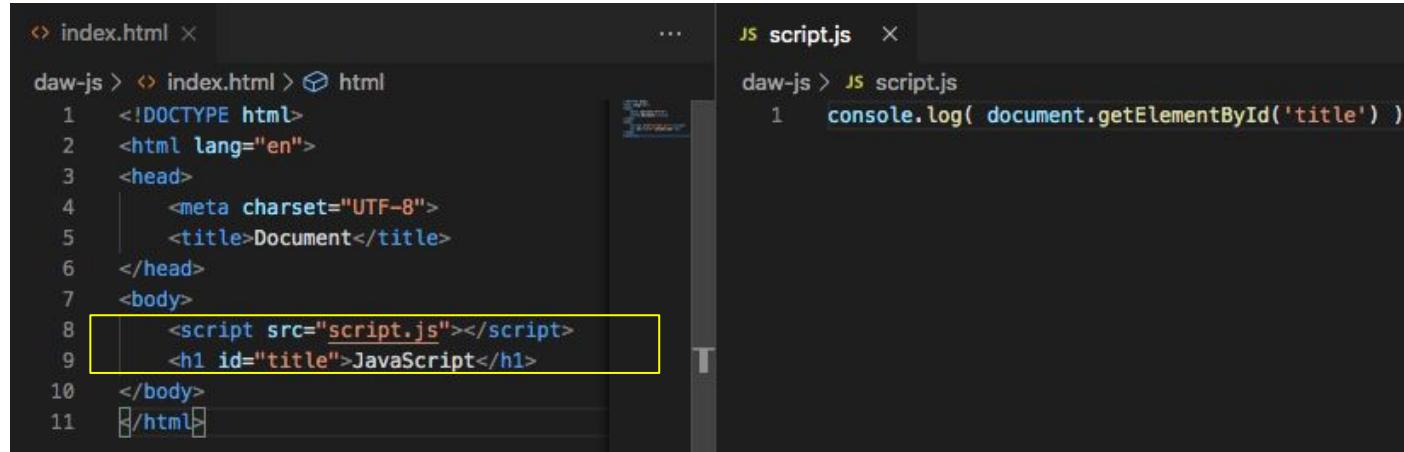
```
daw-js > script.js
1  console.log('Hello World!')
```

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



The image shows a code editor with two tabs: 'index.html' and 'script.js'. The 'index.html' tab contains the following code:

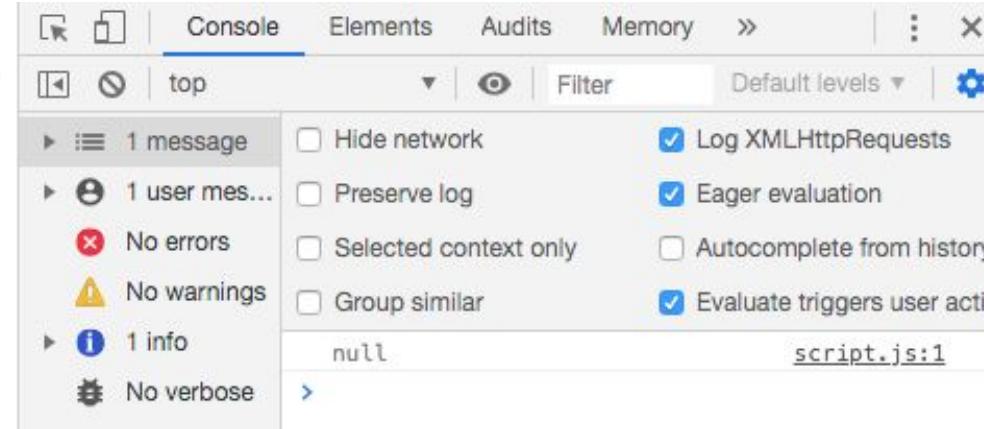
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <script src="script.js"></script>
9   <h1 id="title">JavaScript</h1>
10 </body>
11 </html>
```

The line '8 <script src="script.js"></script>' is highlighted with a yellow box.

The 'script.js' tab contains the following code:

```
1 console.log( document.getElementById('title') );
```

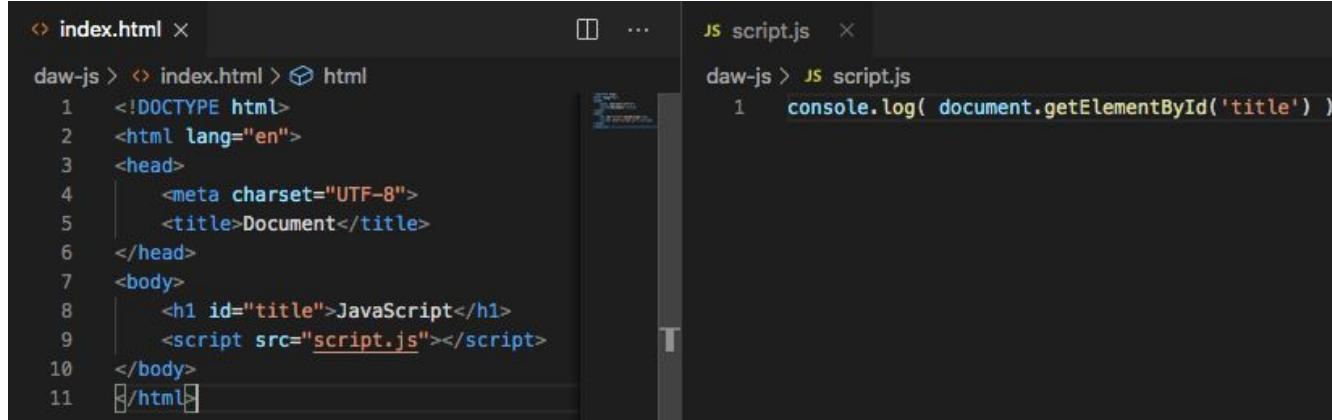
JavaScript



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.

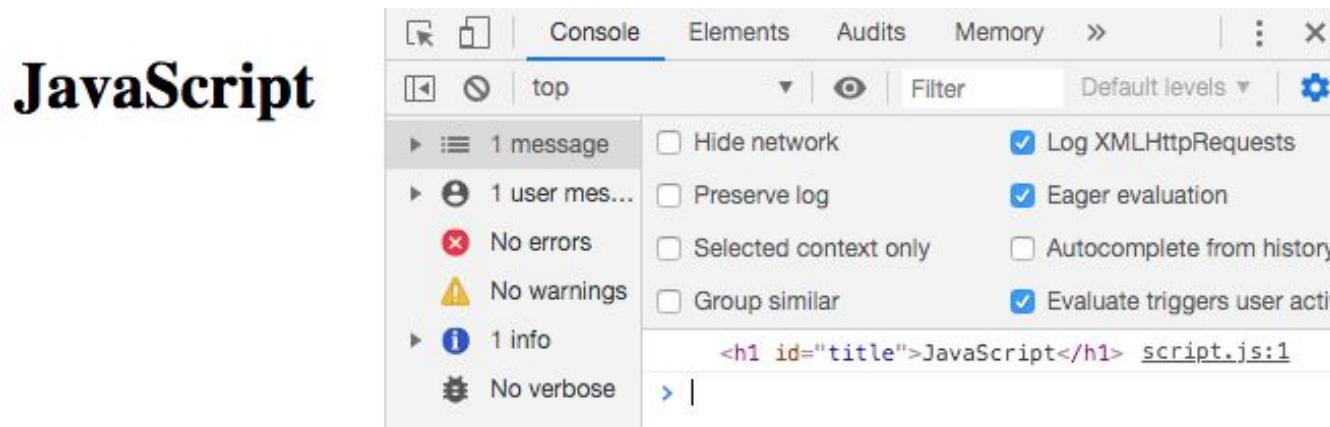


The image shows a code editor with two tabs: 'index.html' and 'script.js'. The 'index.html' tab contains the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1 id="title">JavaScript</h1>
    <script src="script.js"></script>
</body>
</html>
```

The 'script.js' tab contains the following code:

```
console.log( document.getElementById('title') );
```



The image shows the 'Console' tab of a browser developer tools interface. The left sidebar displays the following log entries:

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

The main area shows the following log output:

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

The log output at the bottom shows:

```
<h1 id="title">JavaScript</h1> script.js:1
```

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>

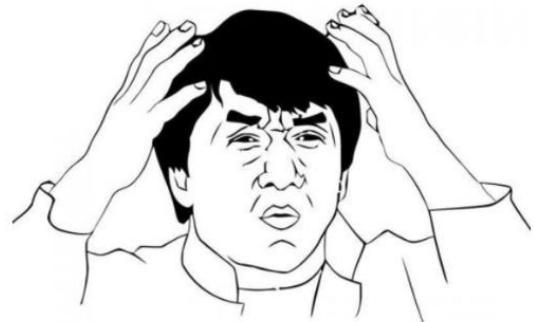
<https://www.w3schools.com/js/default.asp>

Fundamentos

```
    a.fn.scrollspy=d,this},a(window).on("load.bs.
y),+function(a){use strict;function b(b){return this.each(function(e[0]))}var c=function(b){this.element=a(b)};c.VERSION="3.3.7",c.
dropdown-menu"),d=b.data("target");if(d||(d=b.attr("href")),d=d&&d.re
st a),f=a.Event("hide.bs.tab",{relatedTarget:b[0]}),g=a.Event("show.
faultPrevented()){var h=a(d);this.activate(b.closest("li"),c),this.
trigger({type:"shown.bs.tab",relatedTarget:e[0]}))}},c.prototype.
u>.active").removeClass("active").end().find('[data-toggle="tab"]
aria-expanded",!0),h?(b[0].offsetWidth,b.addClass("in")):b.removeClass('.
find('[data-toggle="tab"]').attr("aria-expanded",!0),e&&e()})var
de)||!d.find(">.fade").length);g.length&&h?g.one("bsTransitionE
;var d=a.fn.tab;a.fn.tab=b,a.fn.tab.Constructor=c,a.fn.tab.noConfig=".
show");a(document).on("click.bs.tab.data-api",'[data-toggle="tab
use strict";function b(b){return this.each(function(){var d=a(this
=typeof b&&e[b]())}var c=function(b,d){this.options=a.extend({},".
",a.proxy(this.checkPosition,this)).on("click.bs.affix.data-api".
null,this.pinnedOffset=null,this.checkPosition());c.VERSION="3.3.
tState=function(a,b,c,d){var e=this.$target.scrollTop(),f=this.$e
"bottom"==this.affixed)return null!=c?!{e+this.unpin<=f.top)&&"b
!=c&&e<=c?"top":null!=d&&i+j>=a-d&&"bottom"},c.prototype.getPin
.RESET).addClass("affix");var a=this.$target.scrollTop(),b=this.
WithEventLoop=function(){setTimeout(a.proxy(this.checkPosition,t
ent.height(),d=this.options.offset,e=d.top,f=d.bottom,
pof e&&(e=d.top(this.$element)),".
ent.css("top","");}
```

¿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

The screenshot shows a browser window with two tabs: "index.html" and "JS script.js". The "script.js" tab is active, displaying the following code:

```
1
2 var a;
3 console.log(a);
```

The browser's developer tools are open, specifically the "Console" tab. The output pane shows:

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

The "info" message is highlighted with a red border. The details for this message are expanded, showing:

- undefined
- script.js:3

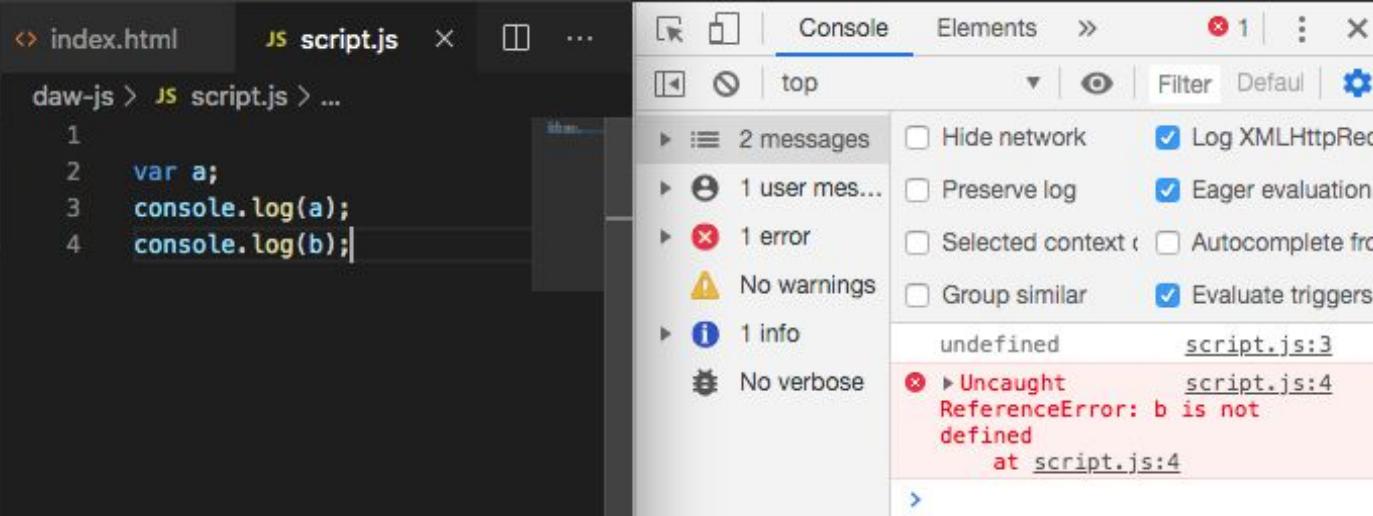
The right side of the developer tools interface contains several configuration checkboxes, many of which are checked:

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

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



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there is a code editor window showing a file named 'script.js' with the following content:

```
1
2 var a;
3 console.log(a);
4 console.log(b);
```

To the right of the code editor is the browser's address bar showing 'index.html' and 'script.js'. The browser's status bar indicates 'daw-js > script.js > ...'.

The main area of the developer tools shows the following message list:

- 2 messages
- 1 user mes...
- 1 error
- No warnings
- 1 info
- No verbose

The '1 error' item is expanded, showing the following details:

undefined [script.js:3](#)

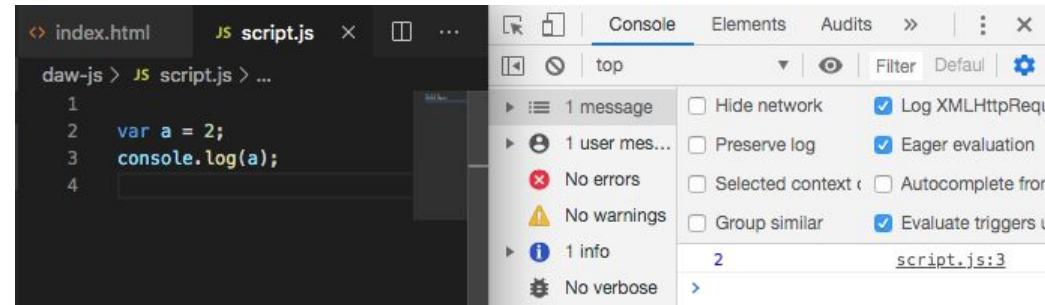
✖ **Uncaught ReferenceError: b is not defined**
at [script.js:4](#)

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

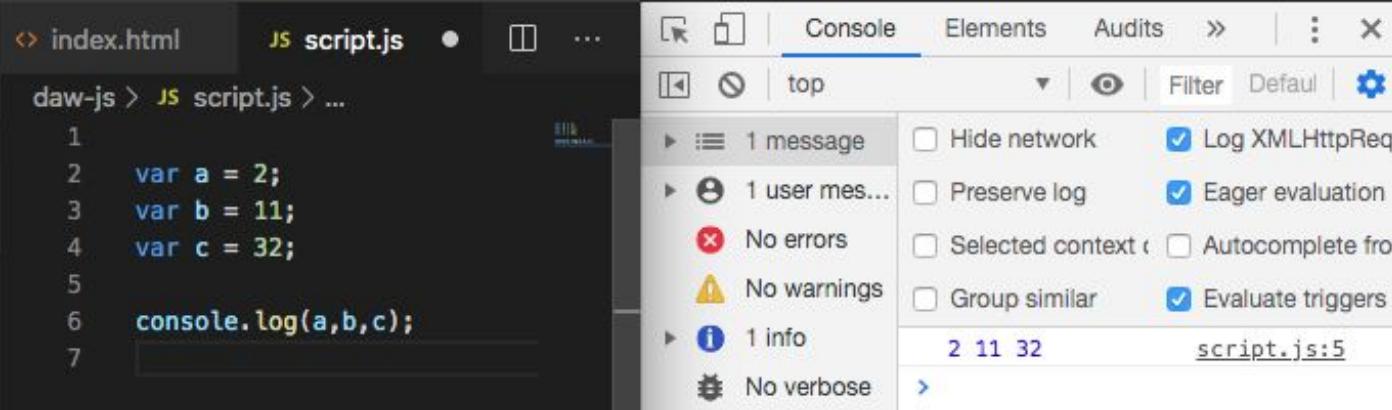
```
var a, b, c;
```

```
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



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, the code in 'script.js' is visible:

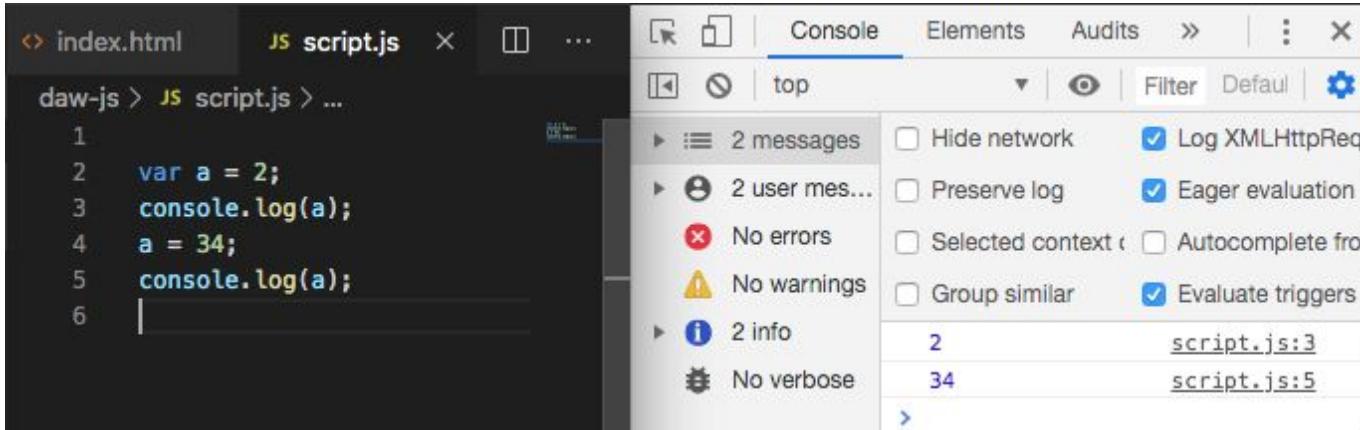
```
1
2  var a = 2;
3  var b = 11;
4  var c = 32;
5
6  console.log(a,b,c);
7
```

On the right, the console output is shown:

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

The 'info' section shows the values 2, 11, 32 and the source script.js:5.

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



The screenshot shows a browser's developer tools with the 'Console' tab selected. On the left, there is a code editor window showing a script named 'script.js' with the following content:

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

On the right, the 'Console' panel displays the output of the script:

- 2 messages
 - 2 user messages
 - No errors
 - No warnings
 - 2 info
 - No verbose
- 2 Hide network
- Preserve log
- Selected context
- Group similar
- Log XMLHttpRequests
- Eager evaluation
- Autocomplete from
- Evaluate triggers

Value	Location
2	script.js:3
34	script.js:5

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.

JavaScript

The screenshot shows a browser window with two tabs: "index.html" and "JS script.js". The "script.js" tab is active, displaying the following code:

```
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
```

To the right is the browser's developer tools Console tab, which lists the following messages:

- 4 messages
- 3 user mes...
- 1 error
- No warnings
- 3 info
- No verbose

The "1 error" message is highlighted in red and details the following error:

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

Below the error message, there are checkboxes for filtering logs:

- Hide network
- Log XMLHttpRequest
- Preserve log
- Eager evaluation
- Selected context
- Autocomplete from
- Group similar
- Evaluate triggers

The error message is also listed in the main log area:

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

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.

JavaScript

The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'script.js'. The 'script.js' tab is active, displaying the following code:

```
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();
```

The right side of the interface shows the console output. It lists three messages: '3 user messages' (No errors, No warnings, 3 info), and '3 info' (No verbose). The '3 info' section contains three entries:

Message	Location
10	script.js:5
20	script.js:8
20	script.js:10

Operadores aritméticos

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

fuente: [w3schools.com](https://www.w3schools.com/jsref/operators_arith.asp)

Asignación de operaciones

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

fuente: [w3schools.com](https://www.w3schools.com/jsref/jsref_op_assignment.asp)

JavaScript

The screenshot shows a browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window titled 'script.js' containing the following JavaScript 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
```

The console output on the right shows:

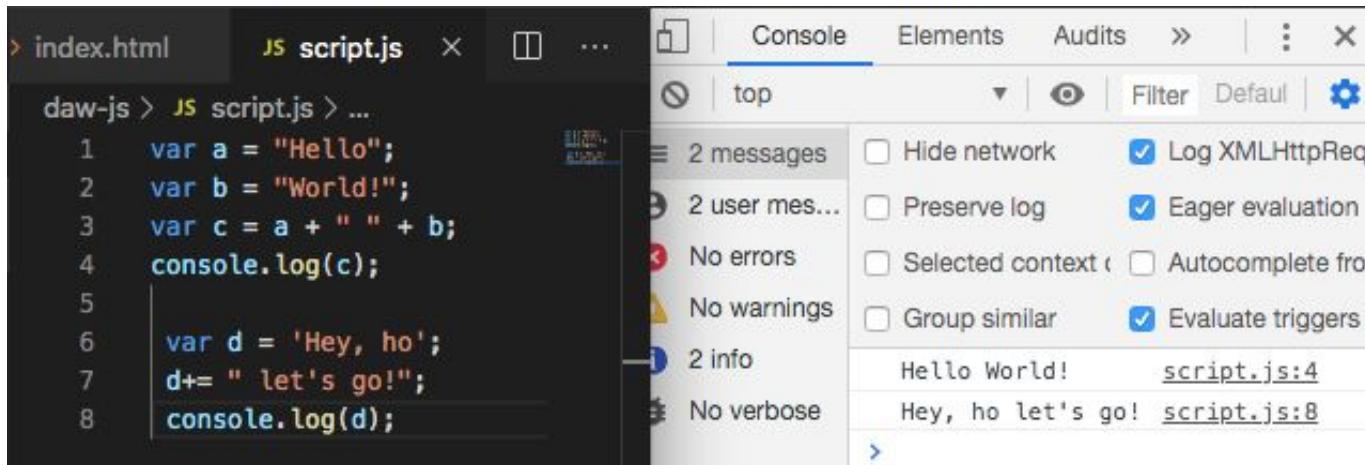
- 2 messages
- 2 user messages
- No errors
- No warnings
- 2 info
- No verbose

Below the messages, two log entries are listed:

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

On the far right of the console panel, there are several configuration checkboxes:
 Hide network traffic
 Log XMLHttpRequests
 Preserve logs
 Eager evaluation
 Selected context
 Autocomplete
 Group similar logs
 Evaluate template literals

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



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'script.js'. The 'script.js' tab is active, displaying the following code:

```
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);
```

On the right, the console output is shown:

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

The 'Info' section contains two entries:

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

Operadores de comparación

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

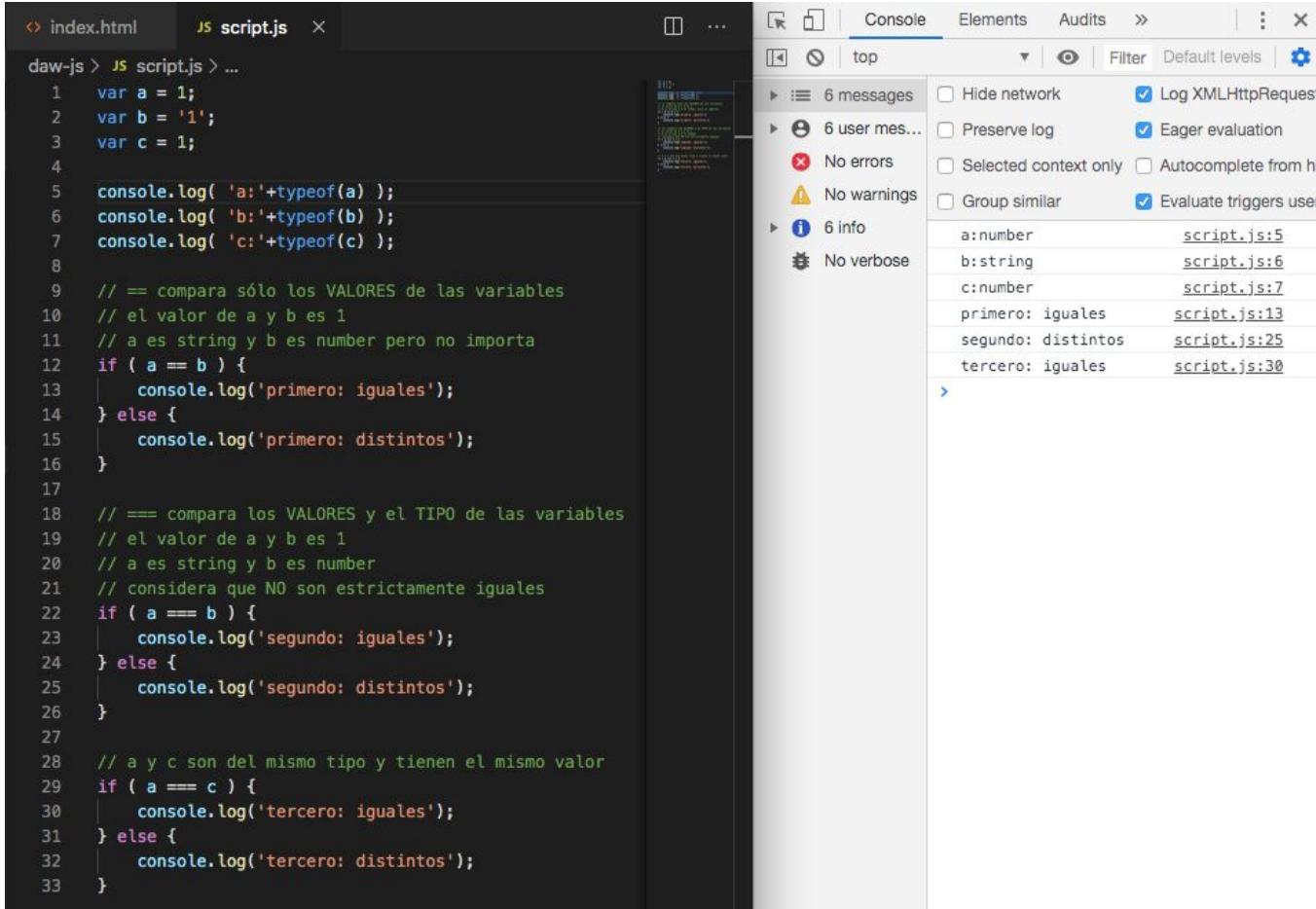
fuente: [w3schools.com](https://www.w3schools.com/js/js_operators.asp)

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

JavaScript



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'script.js'. The code in 'script.js' is as follows:

```
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 }
```

The right panel shows the console output with 6 messages:

- 6 user messages
- No errors
- No warnings
- 6 info

The 'Info' section shows the following log entries:

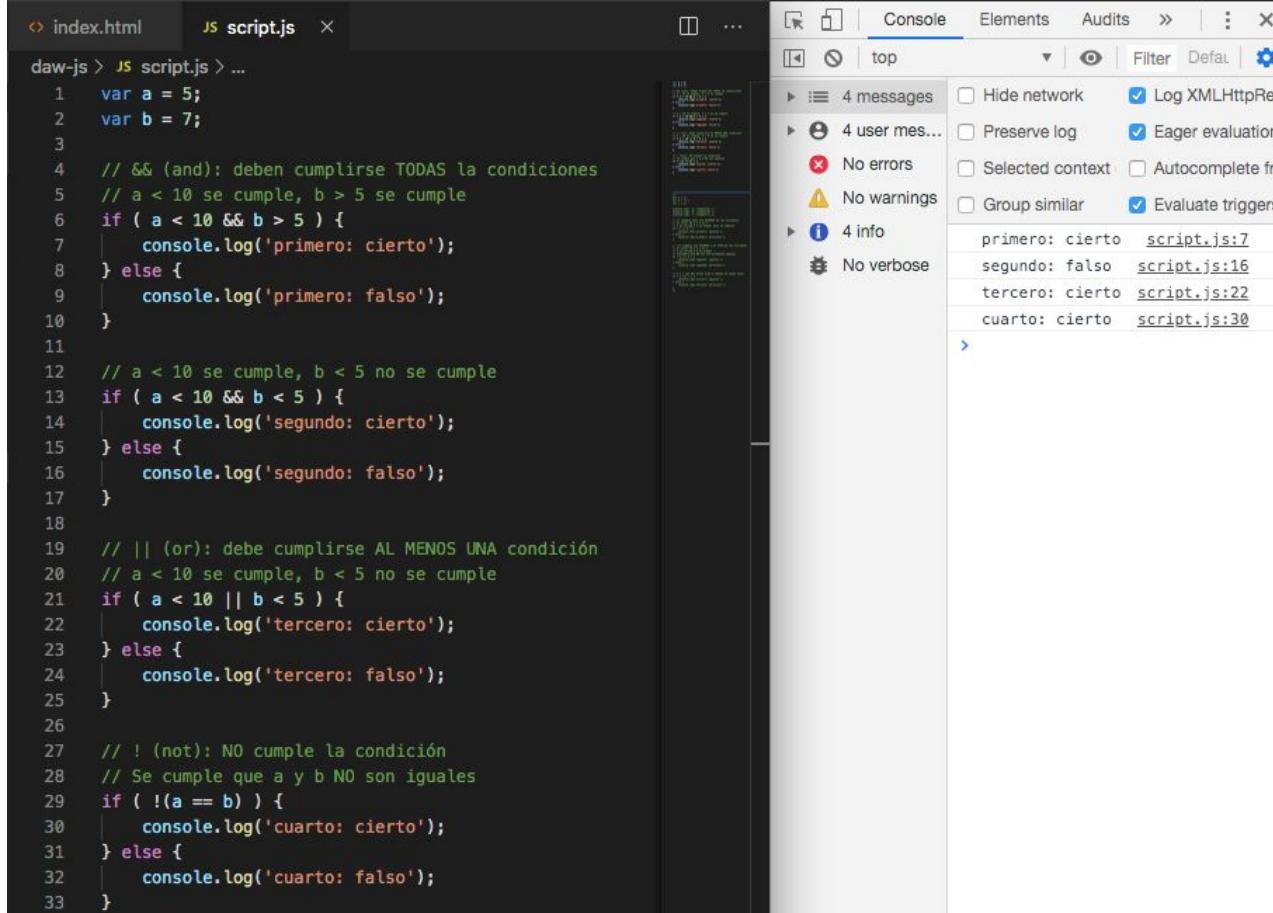
Message	Script
a:number	script.js:5
b:string	script.js:6
c:number	script.js:7
primero: iguales	script.js:13
segundo: distintos	script.js:25
tercero: iguales	script.js:30

Operadores lógicos

Operator	Description
&&	logical and
	logical or
!	logical not

fuente: w3schools.com

JavaScript



The screenshot shows a browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html' and 'script.js'. The 'script.js' tab is active, displaying the following code:

```
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 }
```

The right panel shows the 'Console' output with the following messages:

- 4 messages
 - No errors
 - No warnings
 - 4 info
 - primero: cierto script.js:7
 - segundo: falso script.js:16
 - tercero: cierto script.js:22
 - cuarto: cierto script.js:30
 - No verbose

There are also several checkboxes in the settings panel on the right, such as 'Log XMLHttpRequests', 'Eager evaluation', and 'Evaluate triggers', which are checked.

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 sola línea.

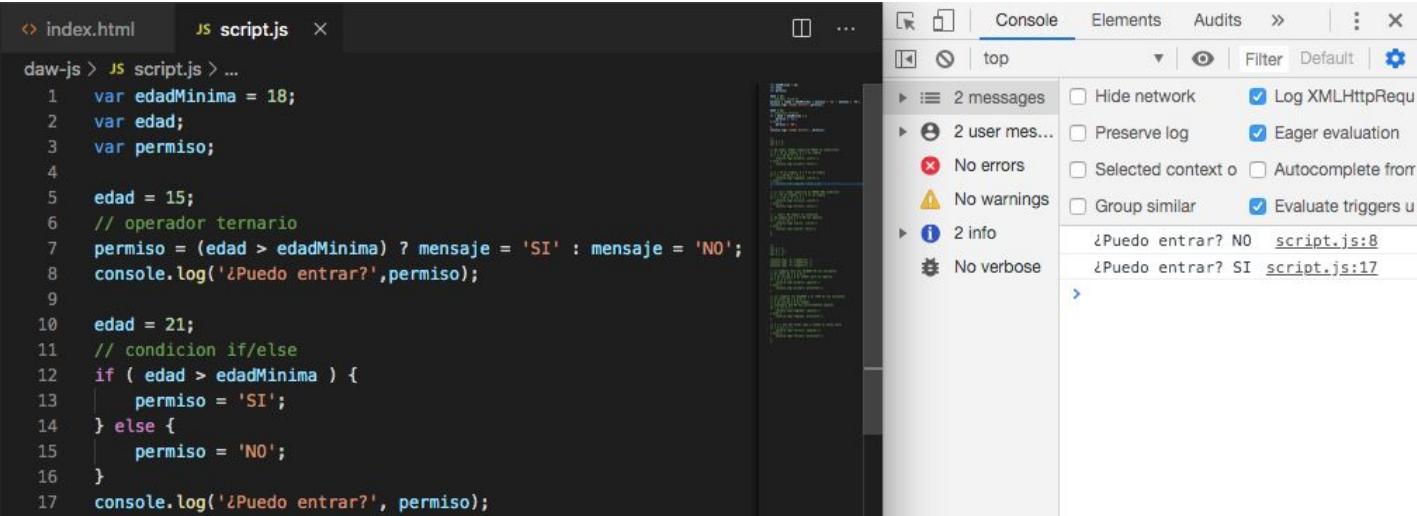
if/else:

```
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 browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'JS script.js'. The code in 'script.js' is as follows:

```
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);
```

The right panel shows the browser's developer tools interface with the 'Console' tab selected. It displays the following log entries:

- 2 messages
 - 2 user mes...
 - No errors
 - No warnings
 - 2 info
 - 2 info
 - ¿Puedo entrar? NO script.js:8
 - ¿Puedo entrar? SI script.js:17

There are several configuration checkboxes on the right side of the console panel, most of which are checked:

 - Hide network
 - Log XMLHttpRequests
 - Preserve log
 - Eager evaluation
 - Selected context
 - 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 dependender 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.

Alert

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

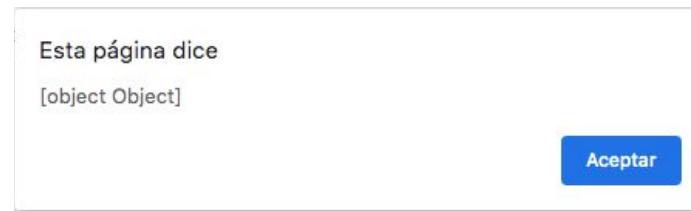
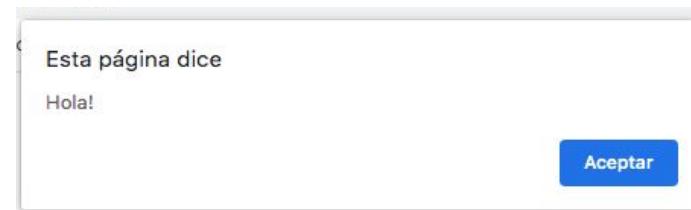
```
alert('Hello JavaScript!');
```

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);
```



Prompt

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.

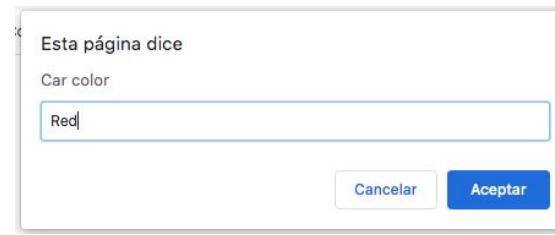
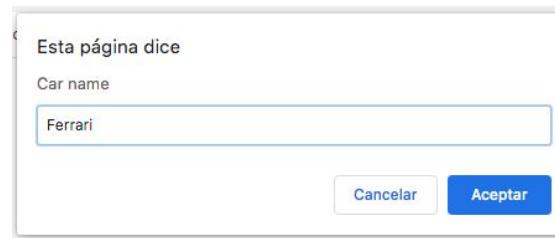
Prompt

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);
```



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.

Funciones y métodos

```
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.

Funciones y métodos

```
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
```

Funciones y métodos

Función (function)	Método (method)
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 recibirlas como argumento
Puede existir por sí misma	Está asociada al objeto dónde se ha declarado

Fuentes

[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/>](#)

Funciones

¿Sabías que...?

Algunas serpientes submarinas respiran a través de su piel



Funciones

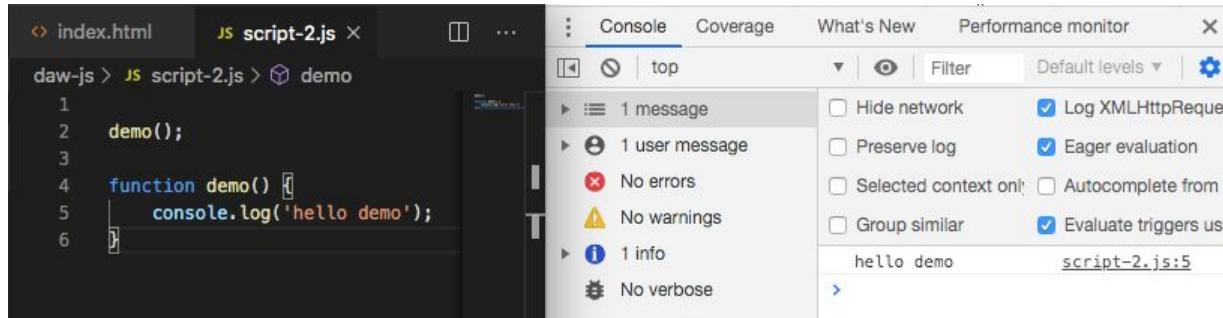
Hoisting

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



The screenshot shows a browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window showing a JavaScript file named 'script-2.js' with the following content:

```
1
2  demo();
3
4  function demo() [
5    console.log('hello demo');
6 ]
```

On the right, the 'Console' panel displays the following log message:

- 1 message
- 1 user message
 - No errors
 - No warnings
 - 1 info
 - hello demo script-2.js:5
 - No verbose

Below the message list are several configuration checkboxes:

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

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**.

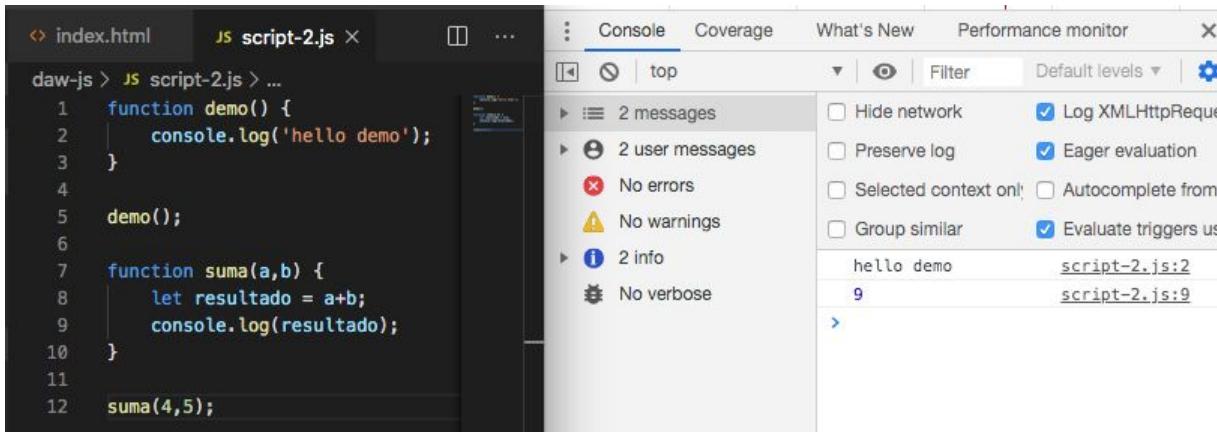
Para invocarla o llamarla, únicamente hay que escribir el nombre de la función seguido de los paréntesis (entre los que se pueden pasar parámetros, opcionalmente)

Funciones

```
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 browser's developer tools console tab. On the left, the code editor displays two functions: `demo()` which logs 'hello demo' to the console, and `suma(a,b)` which adds its arguments and logs the result. On the right, the console panel shows the output of these logs. It lists 2 messages (the demo log), 2 user messages (the suma log), and 2 info entries: 'hello demo' at line 2 of script-2.js and the result '9' at line 9 of script-2.js. The console interface includes various settings like 'Log XMLHttpRequests' and 'Eager evaluation'.

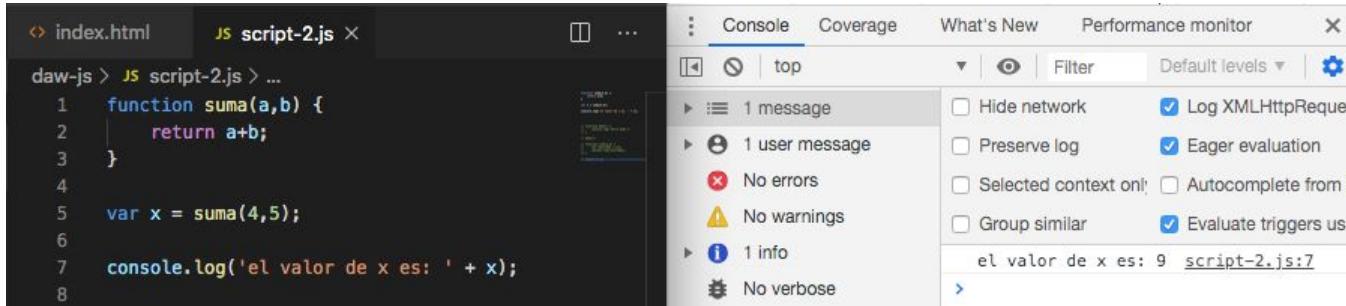
```
index.html    JS script-2.js × ...  
daw-js > JS script-2.js > ...  
1  function demo() {  
2      console.log('hello demo');  
3  }  
4  
5  demo();  
6  
7  function suma(a,b) {  
8      let resultado = a+b;  
9      console.log(resultado);  
10 }  
11  
12 suma(4,5);
```

Message	Level	File	Line
hello demo	info	script-2.js	2
9	info	script-2.js	9

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



The screenshot shows a browser developer tools interface with two tabs: "index.html" and "JS script-2.js". The "script-2.js" tab contains the following code:

```
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);
```

The "Console" tab displays the following output:

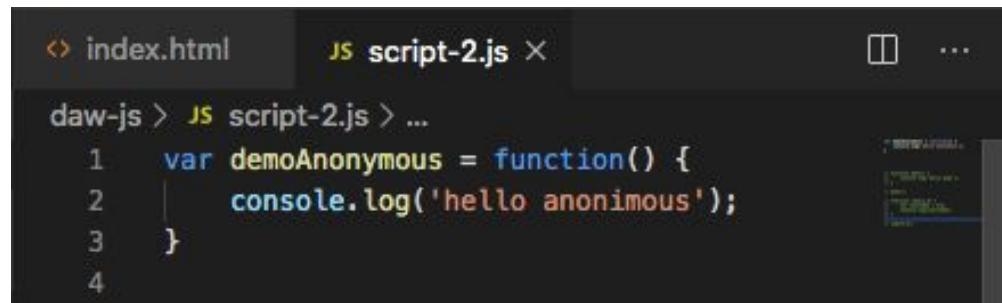
- 1 message
- 1 user message
 - No errors
 - No warnings
- 1 info
 - No verbose

The "el valor de x es: 9" message is highlighted in blue, indicating it was triggered by the "Evaluate triggers used" option in the settings.

2. Expresiones (anonymous function)

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

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



```
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.

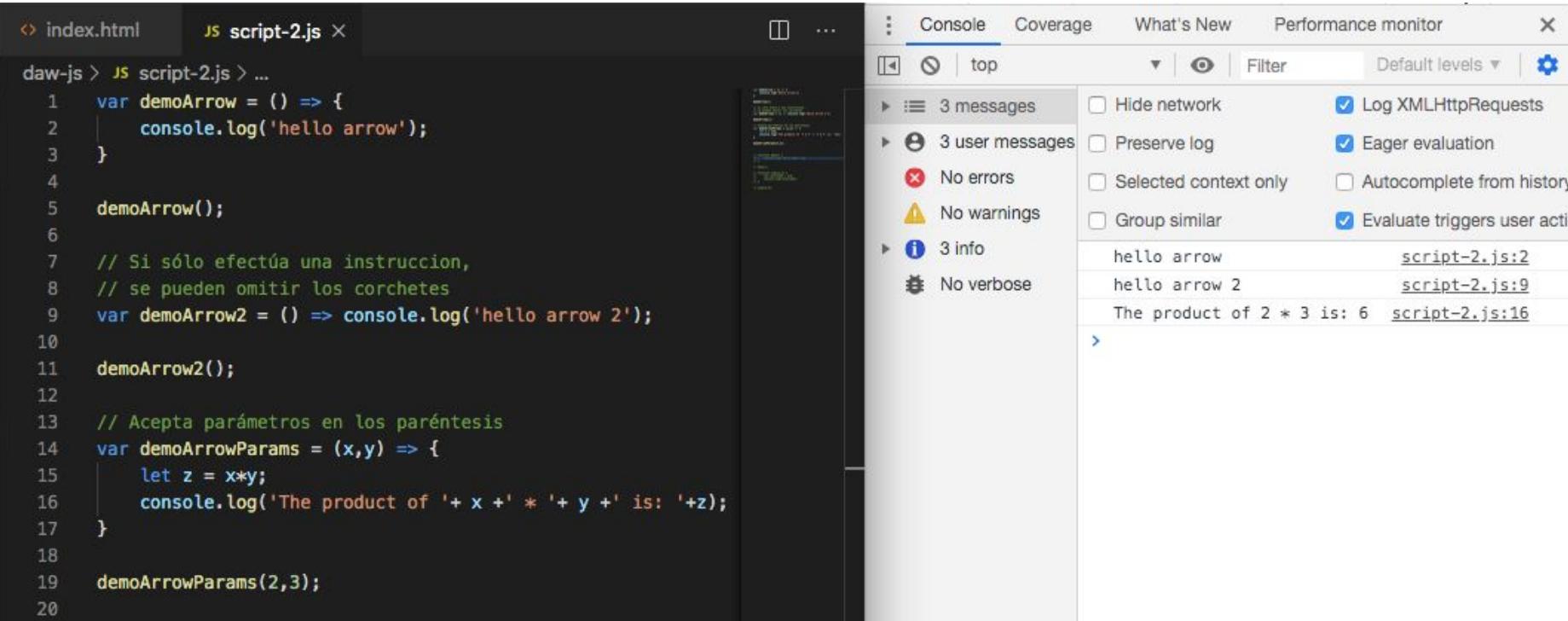
Se trata de una forma más corta de escribirlas (no hace falta la palabra **function**).

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 code editor with two tabs: "index.html" and "JS script-2.js". The "script-2.js" tab contains the following code:

```
daw-js > JS script-2.js > ...
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
```

To the right of the code editor is a browser window displaying the output of the console logs. The browser's developer tools are open, specifically the "Console" tab. The console output is as follows:

- 3 messages
 - No errors
 - No warnings
 - No verbose
- 3 user messages
 - hello arrow
 - hello arrow 2
 - The product of 2 * 3 is: 6
- 3 info
 - Autocomplete from history
 - Eager evaluation
 - Evaluate triggers user acti

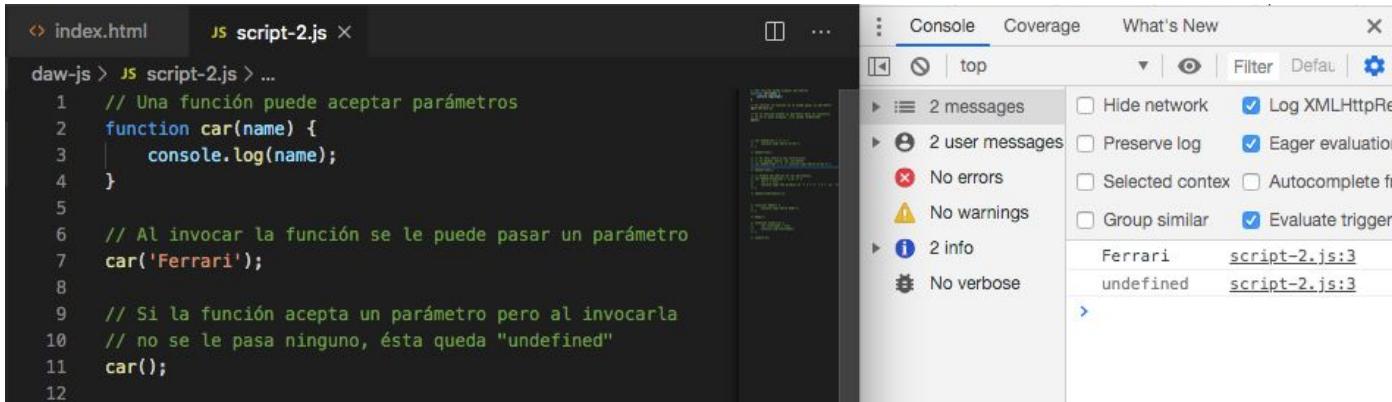
Below the list of messages, the individual log entries are shown:

- hello arrow (script-2.js:2)
- hello arrow 2 (script-2.js:9)
- The product of 2 * 3 is: 6 (script-2.js:16)

Parámetros

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

El parámetro queda **undefined**.



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there is a code editor window titled 'script-2.js' containing the following JavaScript code:

```
// Una función puede aceptar parámetros
function car(name) {
    console.log(name);
}

// Al invocar la función se le puede pasar un parámetro
car('Ferrari');

// Si la función acepta un parámetro pero al invocarla
// no se le pasa ninguno, ésta queda "undefined"
car();
```

On the right, the 'Console' tab displays the output of the script:

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

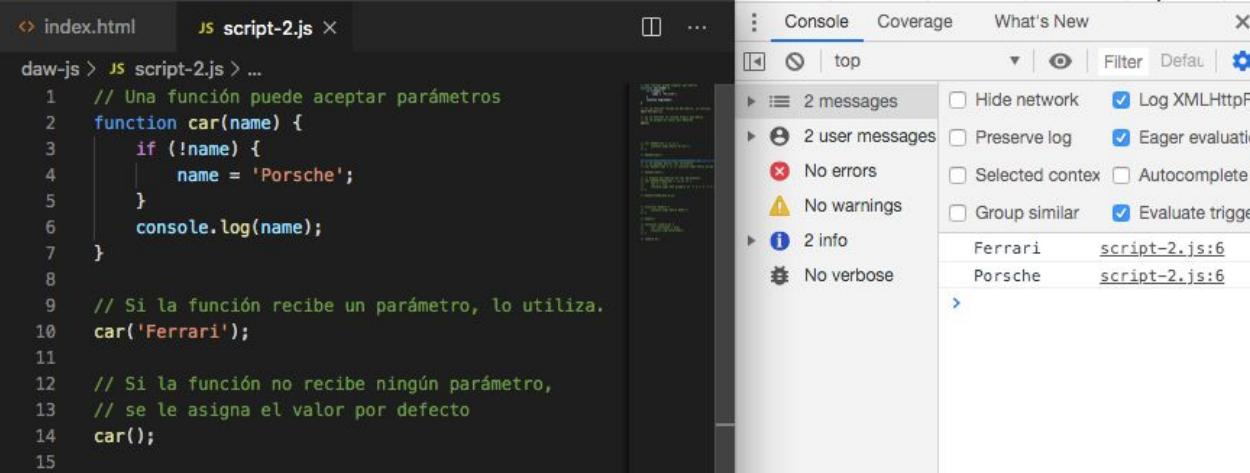
The 'Info' section shows two entries:

- Ferrari script-2.js:3
- undefined script-2.js:3

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.



The screenshot shows a browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window titled 'script-2.js' containing the following JavaScript code:

```
// Una función puede aceptar parámetros
function car(name) {
  if (!name) {
    name = 'Porsche';
  }
  console.log(name);
}

// Si la función recibe un parámetro, lo utiliza.
car('Ferrari');

// Si la función no recibe ningún parámetro,
// se le asigna el valor por defecto
car();
```

On the right, the browser's developer tools Console panel displays the following log output:

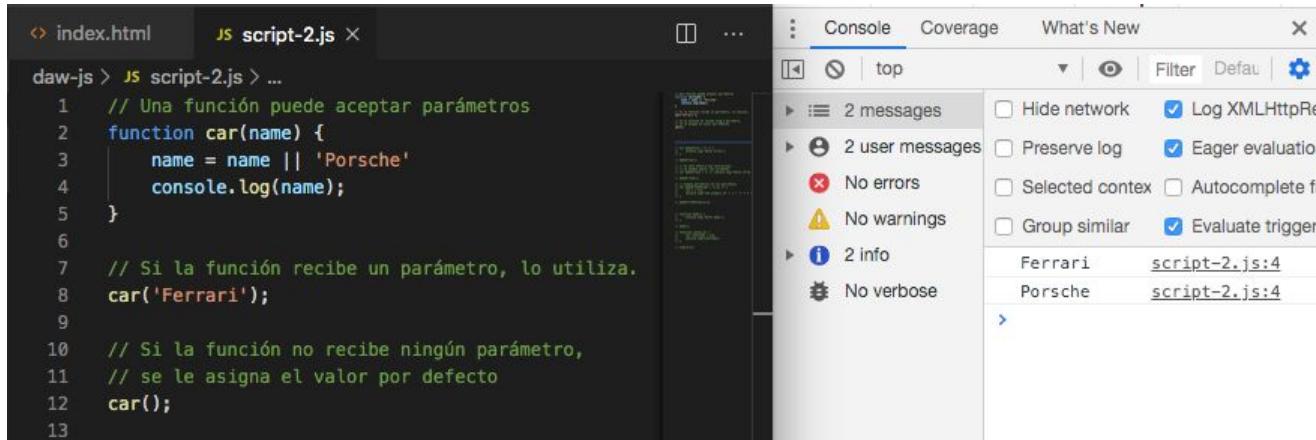
- 2 messages
 - 2 user messages
 - No errors
 - No warnings
 - 2 info
 - No verbose

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



The screenshot shows a browser developer tools interface with two main panes. On the left is the code editor for 'script-2.js' containing the following JavaScript:

```
// Una función puede aceptar parámetros
function car(name) {
  name = name || 'Porsche'
  console.log(name);
}

// Si la función recibe un parámetro, lo utiliza.
car('Ferrari');

// Si la función no recibe ningún parámetro,
// se le asigna el valor por defecto
car();
```

On the right is the 'Console' tab of the developer tools, showing the following log output:

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

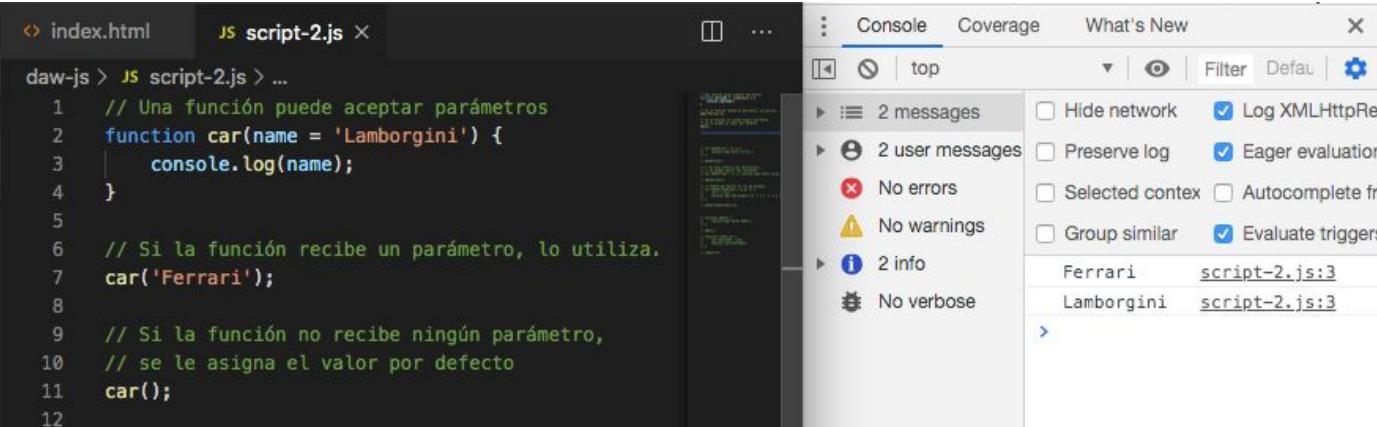
The log entries are:

- 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 ("Lamborgini" en el ejemplo). En cambio, si recibe algún parámetro al ser invocada, utilizará éste.



The screenshot shows a browser developer tools interface with two tabs: "index.html" and "JS script-2.js". The "script-2.js" tab contains the following code:

```
// Una función puede aceptar parámetros
function car(name = 'Lamborgini') {
    console.log(name);
}

// Si la función recibe un parámetro, lo utiliza.
car('Ferrari');

// Si la función no recibe ningún parámetro,
// se le asigna el valor por defecto
car();
```

The "Console" tab on the right displays the following log entries:

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

Specific log entries shown are:

- Ferrari script-2.js:3
- Lamborgini 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 browser developer tools interface with the 'Console' tab selected. On the left, there is a code editor window titled 'script-2.js' containing the following JavaScript code:

```
// declaración
function demo() {
    console.log('función invocada')
}
// invocación
demo();

// función autoinvocada
(function () {
    console.log('función autoinvocada')
})()
```

On the right, the 'Console' panel displays the following log entries:

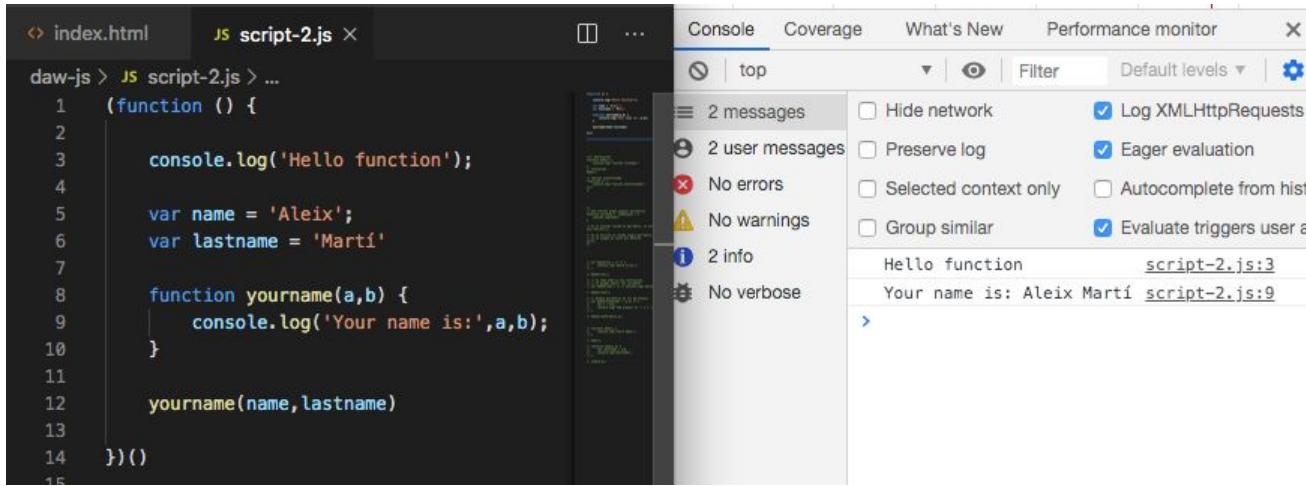
- 2 messages
 - 2 user messages
 - No errors
 - No warnings
 - 2 info
 - No verbose

Details of the log entries:

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

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 browser developer tools console window. The tabs at the top are 'index.html' and 'JS script-2.js'. The code in 'script-2.js' is:

```
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 })()
```

The console output shows:

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

Details of the info message:

- Hello function script-2.js:3
- Your name is: Aleix Martí script-2.js:9

Callback

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

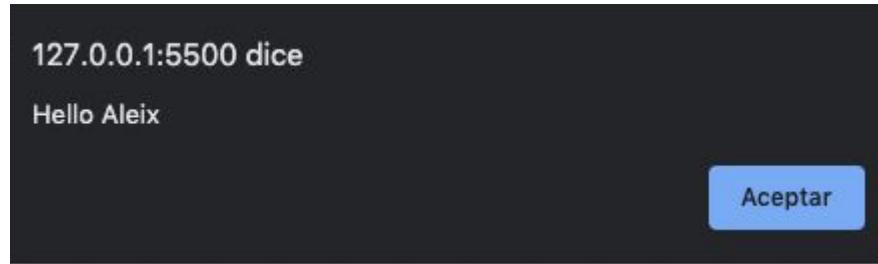
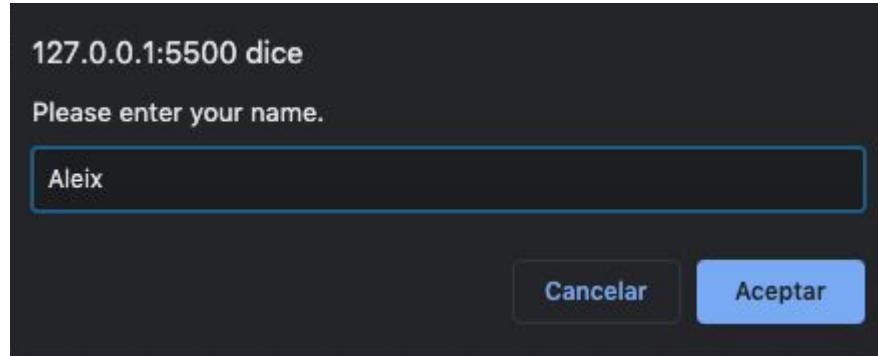
Esta técnica permite que una función llame a otra función.

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

Funciones

```
function greeting(name) {  
    alert('Hello ' + name);  
}  
  
function processUserInput(mycb) {  
    var yourname = prompt('Please enter your name.');?>  
    mycb(yourname);  
}  
  
processUserInput(greeting);
```

Funciones



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
```

Fuentes

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

https://www.w3schools.com/js/js_callback.asp

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

ES5 vs ES6

```
),removeEvent,a.fn.scrollspy=d>this},a(window).on("load.bs.  
y),+function(a){use strict;function b(b){return this.each(function()  
&e[b]())}var c=function(b){this.element=a(b)};c.VERSION="3.3.7",c.  
dropdown-menu"),d=b.data("target");if(d||(d=b.attr("href")),d=d&&d.re  
st a),f=a.Event("hide.bs.tab",{relatedTarget:b[0]}),g=a.Event("sh  
faultPrevented()){var h=a(d);this.activate(b.closest("li"),c),this.  
trigger({type:"shown.bs.tab",relatedTarget:e[0]}))}}},c.prototype.  
u>.active).removeClass("active").end().find('[data-toggle="tab"]  
aria-expanded",!0),h?(b[0].offsetWidth,b.addClass("in")):b.removeCl  
().find('[data-toggle="tab"]').attr("aria-expanded",!0),e&&e()}{var  
de)||!ld.find(">.fade").length);g.length&&h?g.one("bsTransitio  
;var d=a.fn.tab;a.fn.tab=b,a.fn.tab.Constructor=c,a.fn.tab.noConfi  
"show");a(document).on("click.bs.tab.data-api",'[data-toggle="tab  
use strict;function b(b){return this.each(function(){var d=a(this  
=typeof b&&e[b]())}var c=function(b,d){this.options=a.extend({},  
,",a.proxy(this.checkPosition,this)).on("click.bs.affix.data-api",  
null,this.pinnedOffset=null,this.checkPosition());c.VERSION="3.3.  
tState=function(a,b,c,d){var e=this.$target.scrollTop(),f=this.$e  
"bottom"==this.affixed) return null!=c?!(e+this.unpin<=f.top)&&"b  
!-=c&&e<=c?"top":null!=d&&i+j>=a-d&&"bottom"},c.prototype.getPin  
.RESET).addClass("affix");var a=this.$target.scrollTop(),b=this.  
WithEventLoop=function(){setTimeout(a.proxy(this.checkPosition,t  
ent.height(),d=this.options.offset,e=d.top,f=d.bottom  
peof e&&(e=d.top(this.$element)),"c  
ent.css("top","","
```

¿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 var	Las variables se pueden definir con var , let o const
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 function y return para ser definidas	Aparece una nueva forma de declarar funciones, las arrow functions
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.

JavaScript

The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'script.js'. The code in 'script.js' is as follows:

```
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();
```

The console output on the right shows three messages:

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

Message	Location
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 líneas 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 variables, se puede usar la interpolación. Todo el conjunto debe escribirse entre `` (ácento 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

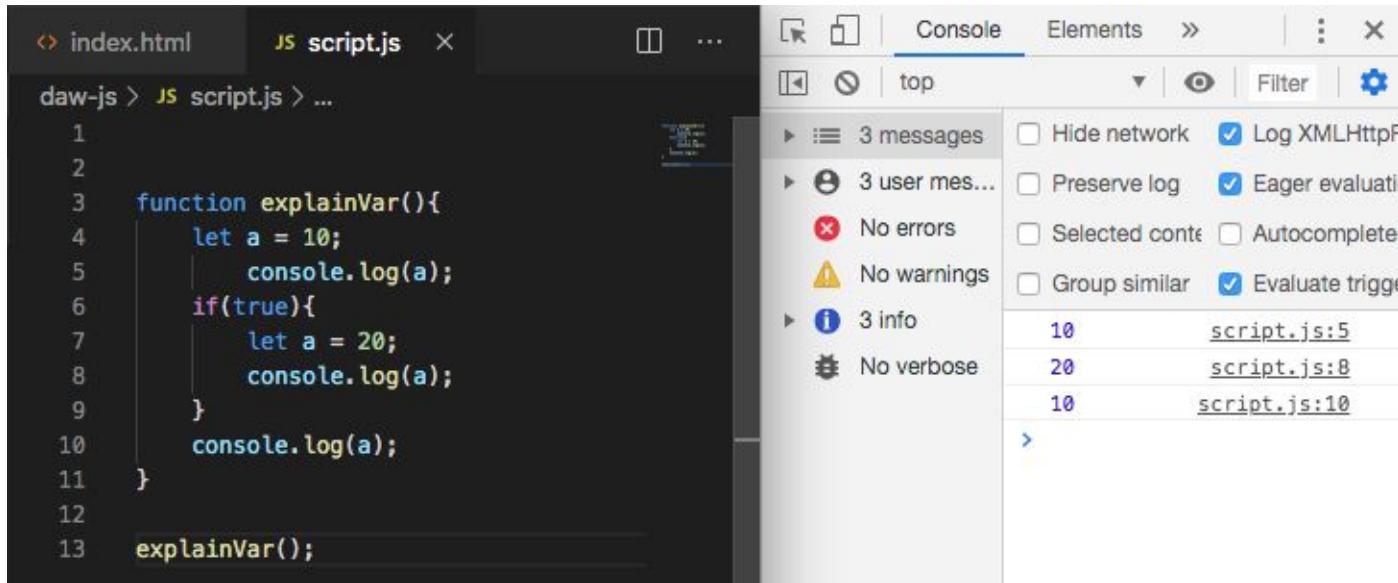
Las variables **let** son más recientes y pueden evitar fácilmente el problema de ES5 de sobreescribir variables var con el mismo nombre.

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 browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'JS script.js'. The 'script.js' tab is active, displaying the following 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 output on the right shows three messages:

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

The 'info' messages are:

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

On the far right, there are several configuration checkboxes:
 Hide network Log XMLHttpRequest
 Preserve log Eager evaluation
 Selected context Autocomplete
 Group similar Evaluate triggered
A scroll bar is visible on the right side of the console output area.

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.

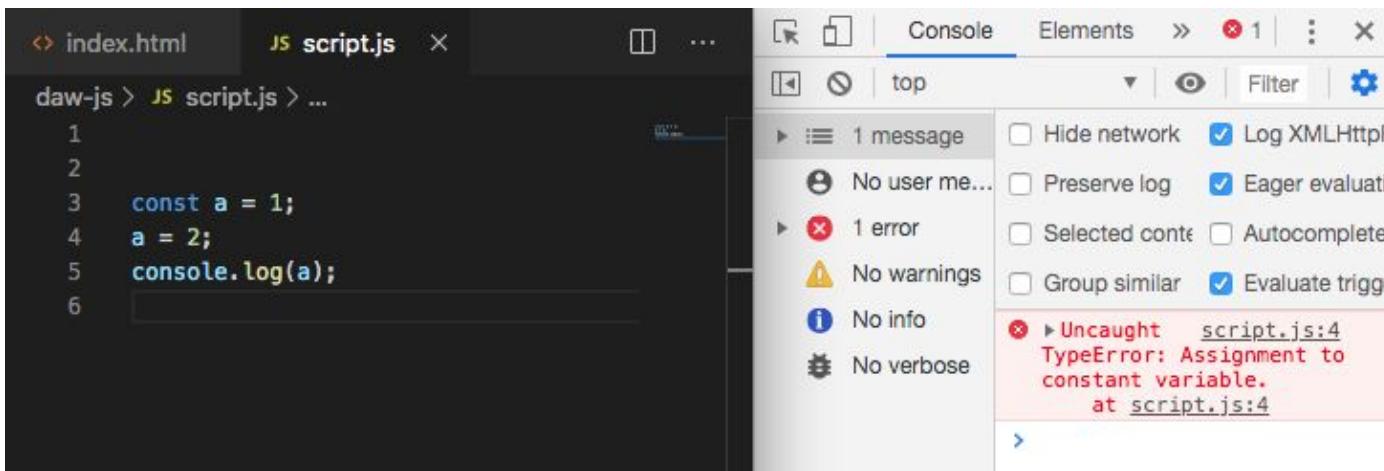
Si que puede cambiar su valor si se trata de un objeto (veremos el concepto más adelante), pero para entenderlo, sería un conjunto de información de tipo clave:valor

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



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there are tabs for 'index.html' and 'JS script.js'. The code in 'script.js' is:

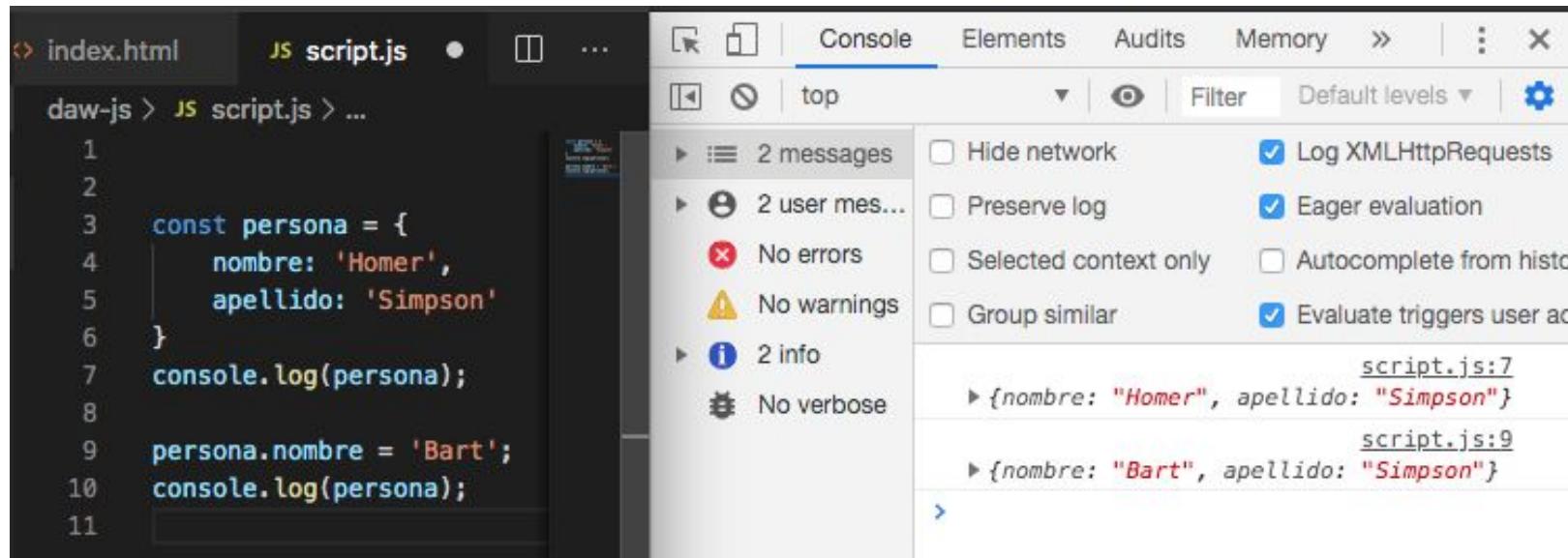
```
1
2
3  const a = 1;
4  a = 2;
5  console.log(a);
6
```

In the console output area, there is one message: 'No user message'. Below it, there is one error:

- 1 error
 - Uncaught script.js:4
TypeError: Assignment to constant variable.
at script.js:4

The error message is highlighted with a red border.

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



The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, the code in 'script.js' is visible:

```
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 console output on the right shows two log statements:

- script.js:7 ▶ {nombre: "Homer", apellido: "Simpson"} (highlighted in red)
- script.js:9 ▶ {nombre: "Bart", apellido: "Simpson"} (highlighted in red)

This demonstrates that while the variable was declared with `const`, its value can still be modified.

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

permiten **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')  
  
})
```

Fuentes

<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/>

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
	car.name = Fiat car.model = 500 car.weight = 850kg car.color = white	car.start() car.drive() car.brake() car.stop()

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');
```

Objetos

```
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:

Demanar 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

Fuentes

https://www.w3schools.com/js/js_objects.asp

https://www.w3schools.com/js/js_object_display.asp

https://www.w3schools.com/js/js_object_constructors.asp

String, Number

```
a.fn.scrollspy=d>this},a(window).on("load.bs.  
y),+function(a){"use strict";function b(b){return this.each(function()  
&e[b]())}var c=function(b){this.element=a(b)};c.VERSION="3.3.7",c.1  
dropdown-menu"),d=b.data("target");if(d||(d=b.attr("href")),d=d&&d.re  
st a),f=a.Event("hide.bs.tab",{relatedTarget:b[0]}),g=a.Event("sho  
faultPrevented()){var h=a(d);this.activate(b.closest("li"),c),this.  
trigger({type:"shown.bs.tab",relatedTarget:e[0]}))}}},c.prototype.  
u>.active").removeClass("active").end().find('[data-toggle="tab"]  
'ia-expanded",!0),h?(b[0].offsetWidth,b.addClass("in")):b.removeClass  
().find('[data-toggle="tab"]').attr("aria-expanded",!0),e&&e()}{var  
de)||!!d.find(">.fade").length);g.length&&h?g.one("bsTransitionE  
;var d=a.fn.tab;a.fn.tab=b,a.fn.tab.Constructor=c,a.fn.tab.noConfig=1  
"show");a(document).on("click.bs.tab.data-api",'[data-toggle="tab"]  
use strict";function b(b){return this.each(function(){var d=a(this)  
=typeof b&&e[b]())}var c=function(b,d){this.options=a.extend({},  
",a.proxy(this.checkPosition,this)).on("click.bs.affix.data-api",  
null,this.pinnedOffset=null,this.checkPosition());c.VERSION="3.3.  
tState=function(a,b,c,d){var e=this.$target.scrollTop(),f=this.$e  
"bottom"==this.affixed){return null!=c?!(e+this.unpin<=f.top)&&"b  
!-=c&&e<=c?"top":null!=d&&i+j>=a-d&&"bottom"},c.prototype.getPin  
.RESET).addClass("affix");var a=this.$target.scrollTop(),b=this.  
WithEventLoop=function(){setTimeout(a.proxy(this.checkPosition,t  
ent.height(),d=this.options.offset,e=d.top,f=d.bottom,  
peof e&&(e=d.top(this.$element)),"c  
ent.css("top","";
```

¿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!
```

Strings

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 carácter 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 Tíirri 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
```

Numbers

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.

Numbers

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
```

Numbers

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
```

Numbers

`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
```

Fuentes

https://www.w3schools.com/js/js_strings.asp

https://www.w3schools.com/js/js_string_methods.asp

https://www.w3schools.com/jsref/jsref_obj_string.asp

https://www.w3schools.com/js/js_string_search.asp

```
a.fn.scrollspy=d>this},a(window).on("load.bs.  
y),+function(a){use strict;function b(b){return this.each(function(e,[b])}{var c=function(b){this.element=a(b)};c.VERSION="3.3.7",c.  
dropdown-menu"),d=b.data("target");if(d||(d=b.attr("href")),d=d&&d.re  
st a),f=a.Event("hide.bs.tab",{relatedTarget:b[0]}),g=a.Event("show.  
faultPrevented(){var h=a(d);this.activate(b.closest("li"),c),this.  
trigger({type:"shown.bs.tab",relatedTarget:e[0]}))}}},c.prototype.  
u>.active").removeClass("active").end().find('[data-toggle="tab"]  
.aria-expanded",!0),h?(b[0].offsetWidth,b.addClass("in")):b.removeClass()  
().find('[data-toggle="tab"]').attr("aria-expanded",!0),e&&e()}{var  
de)||!l.d.find(">.fade").length);g.length&&h?g.one("bsTransitionE  
;var d=a.fn.tab;a.fn.tab=b,a.fn.tab.Constructor=c,a.fn.tab.noConfi  
"show");a(document).on("click.bs.tab.data-api",'[data-toggle="tab"  
use strict;function b(b){return this.each(function(){var d=a(this  
=typeof b&&e[b]())}{var c=function(b,d){this.options=a.extend({},  
",a.proxy(this.checkPosition,this)).on("click.bs.affix.data-api",  
null,this.pinnedOffset=null,this.checkPosition());c.VERSION="3.3.  
tState=function(a,b,c,d){var e=this.$target.scrollTop(),f=this.$e  
"bottom"==this.affixed){return null!=c?!(e+this.unpin<=f.top)&&"b  
!=c&&e<=c?"top":null!=d&&i+j>=a-d&&"bottom"},c.prototype.getPin  
.RESET).addClass("affix");var a=this.$target.scrollTop(),b=this.  
WithEventLoop=function(){setTimeout(a.proxy(this.checkPosition,t  
ent.height(),d=this.options.offset,e=d.top,f=d.bottom  
peof e&&(e=d.top(this.$element)),"c  
ent.css("top","")};
```

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      // Raiz cuadrada de 2
Math.SQRT1_2    // Raiz 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;  
}
```

Fuentes

https://www.w3schools.com/js/js_object_maps.asp

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

https://www.w3schools.com/jsref/jsref_obj_date.asp

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

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/js/js_math.asp

https://www.w3schools.com/js/js_strings.asp

https://www.w3schools.com/js/js_string_methods.asp

¿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.

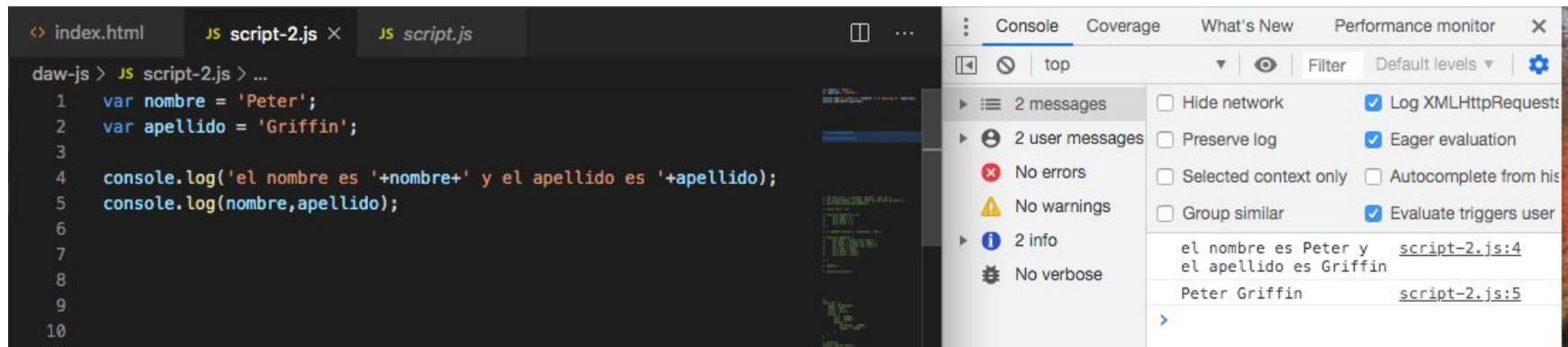
Hasta ahora hemos utilizado **console.log()**, muy útil para representar valores simples o strings. Permite encadenar strings y variables usando +

```
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 browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html', 'script-2.js', and 'script.js'. The code in 'script-2.js' is as follows:

```
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
```

The console output on the right shows the following messages:

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

The 'Info' section contains two entries:

- el nombre es Peter y el apellido es Griffin
- Peter Griffin

Below the 'Info' section, there is a link labeled '>'.

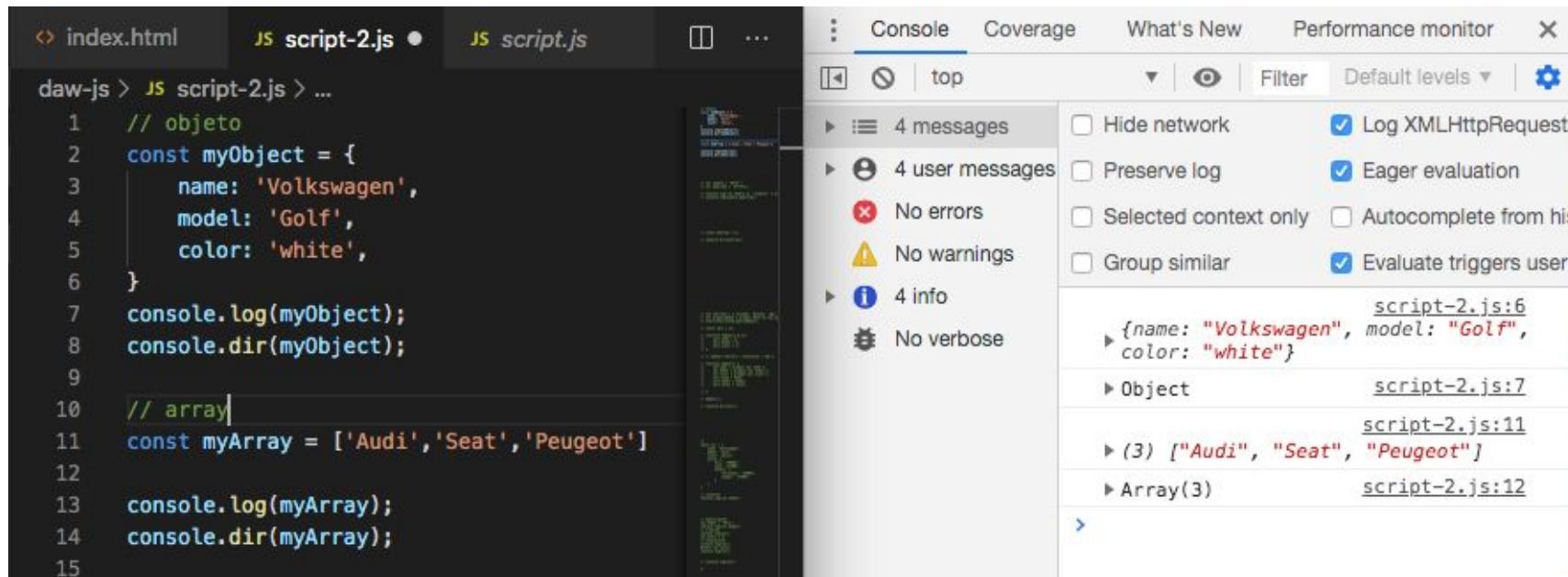
Hay muchas formas de [mostrar contenido en la consola](#). El más “famoso” es `console.log`, pero otra forma muy útil es `console.dir()`. Es especialmente útil cuando queremos mostrar objetos o arrays.

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 browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html', 'script-2.js' (which is active), and 'script.js'. The code in 'script-2.js' is as follows:

```
// objeto
const myObject = {
    name: 'Volkswagen',
    model: 'Golf',
    color: 'white',
}
console.log(myObject);
console.dir(myObject);

// array
const myArray = ['Audi','Seat','Peugeot']
console.log(myArray);
console.dir(myArray);
```

The console output on the right shows the results of the logging statements:

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

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 browser's developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html' and 'script-2.js'. The code in 'script-2.js' is:

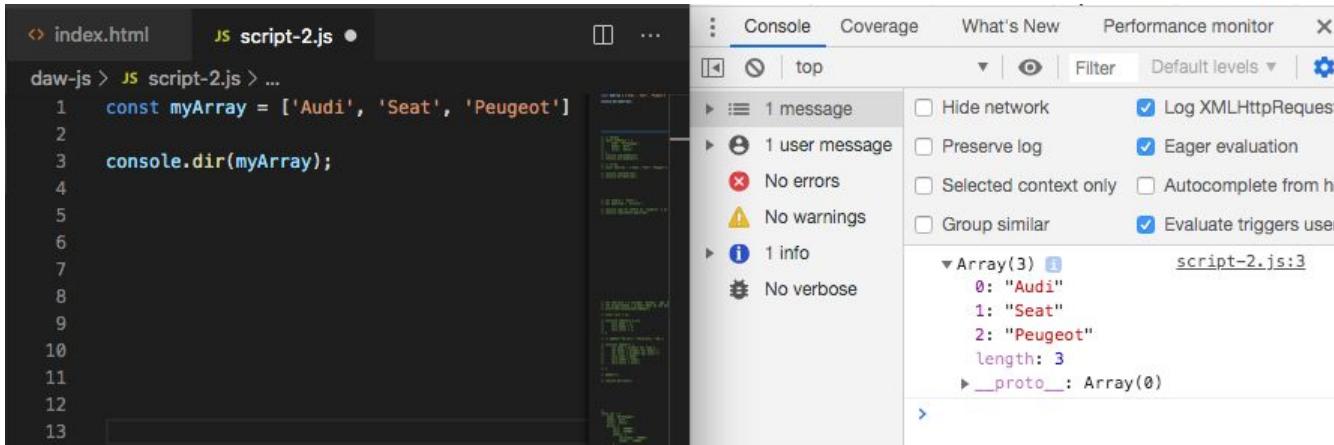
```
1 const miArray = [];
2
3 console.dir(miArray);
```

The console output on the right shows:

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

On the far right, there are several filter checkboxes:
 Hide network Log XMLHttpRequest
 Preserve log Eager evaluation
 Selected context Autocomplete
 Group similar Evaluate triggers
▶ Array(0) [script-2.js:3](#)

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



The screenshot shows the Chrome DevTools Console tab with the following code in 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 output shows:

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

Details for the info message:

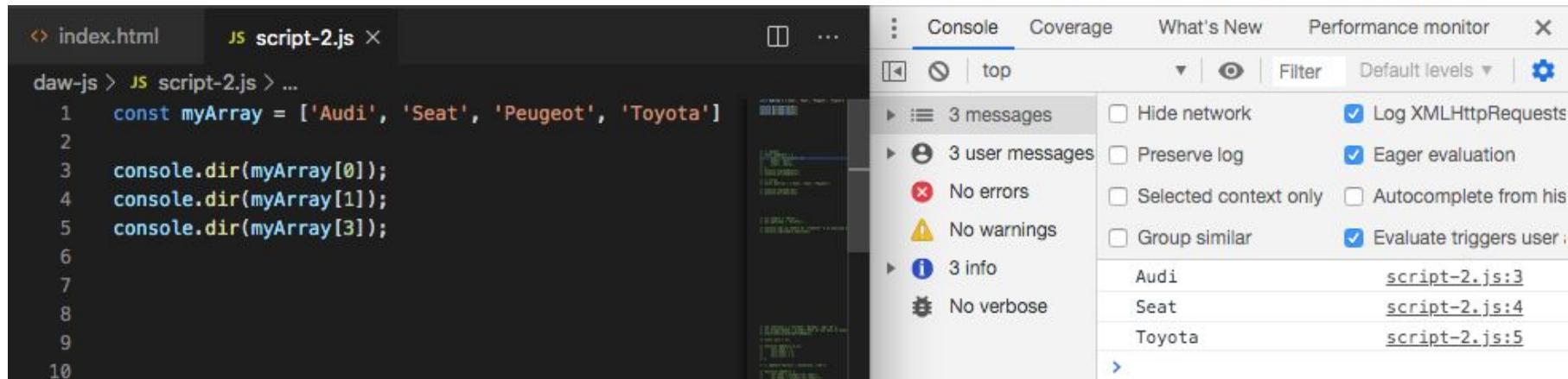
- Array(3) script-2.js:3
 - 0: "Audi"
 - 1: "Seat"
 - 2: "Peugeot"
 - length: 3
 - __proto__: Array(0)

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 browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html' and 'JS script-2.js'. The code in 'script-2.js' is as follows:

```
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
```

The console output on the right shows the results of the `console.dir` calls:

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

Message	Line Number
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 browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html' and 'JS script-2.js'. The code in 'script-2.js' is as follows:

```
daw-js > JS script-2.js > ...
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 output on the right shows the state of the 'fruits' array after the modification:

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

Array(4) [script-2.js:3](#)

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

length: 4

__proto__: Array(0)

Checkmarks in the settings sidebar indicate the following options are enabled: Log XMLHttpRequests, Eager evaluation, and Evaluate triggers user.

Añadir

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

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

Arrays

The screenshot shows a browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window for 'script-2.js' containing the following JavaScript code:

```
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 right side of the interface is the 'Console' panel, which displays the output of the `console.dir(fruits);` command. The output shows an array with five elements: 'Banana', 'Orange', 'Apple', 'Mango', and 'Lemon'. The 'length' property is shown as 5, and the prototype is an empty array.

Setting	Value
Hide network	<input type="checkbox"/>
Preserve log	<input type="checkbox"/>
Selected context only	<input type="checkbox"/>
Group similar	<input type="checkbox"/>
Log XMLHttpRequests	<input checked="" type="checkbox"/>
Eager evaluation	<input checked="" type="checkbox"/>
Autocomplete from his	<input type="checkbox"/>
Evaluate triggers user	<input checked="" type="checkbox"/>

script-2.js:3

```
0: "Banana"
1: "Orange"
2: "Apple"
3: "Mango"
4: "Lemon"
length: 5
__proto__: Array(0)
```

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 browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window for 'script-2.js' containing the following code:

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

The console output on the right shows the state of the 'fruits' variable after the deletion:

- 1 message
- 1 user message
 - No errors
 - No warnings
 - 1 info
 - No verbose
- Array(4) script-2.js:3
 - 0: "Banana"
 - 1: "Orange"
 - 3: "Mango"
 - length: 4
- __proto__: Array(0)

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 () { . . . }
```

Propiedades

```
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"]
```

Fuentes

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

https://www.w3schools.com/jsref/jsref_obj_array.asp

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

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 → 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 browser developer tools interface with the 'Console' tab selected. On the left, there's a code editor window for 'script-2.js' containing the following code:

```
1  for (i = 0; i < 5; i++) {  
2      console.log('El valor de i es:' + i)  
3  }
```

The console output on the right lists five messages, all of which are user messages. Each message is a log statement from the script, showing the value of 'i' at each iteration: 0, 1, 2, 3, and 4. There are also summary sections for errors, warnings, and info levels.

Level	Message	File
user	El valor de i es:0	script-2.js:2
user	El valor de i es:1	script-2.js:2
user	El valor de i es:2	script-2.js:2
user	El valor de i es:3	script-2.js:2
user	El valor de i es:4	script-2.js:2

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 browser developer tools interface with the 'Console' tab selected. On the left, there are tabs for 'index.html' and 'script-2.js'. The code in 'script-2.js' is as follows:

```
1 const cars = ['Audi', 'Seat', 'Peugeot', 'Ferrari']
2
3 for (i = 0; i < cars.length; i++) {
4     console.log(cars[i])
5 }
6
7
```

The console output on the right lists four messages, each corresponding to one of the car names from the array, with the file path 'script-2.js:4' indicated for each:

- Audi
- Seat
- Peugeot
- Ferrari

Each entry includes a blue link labeled 'script-2.js:4'.

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

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 browser developer tools interface with two main panels: a code editor and a console tab.

Code Editor: The left panel displays a file named "script-html.js". The code contains a loop that logs three car brands to the console. It also includes a log statement with a separator and a forEach loop.

```
JS script-html.js ×
daw-js > JS script-html.js > ...
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
```

Console Tab: The right panel is the "Console" tab. It lists 7 messages, all of which are user messages. The output shows the three car brands: Audi, Seat, and Peugeot, each followed by the file name "script-html.js" and the corresponding line number (29 or 35).

Message Type	Content	File	Line
7 messages	Audi	script-html.js	29
7 user messages	Seat	script-html.js	29
No errors	Peugeot	script-html.js	29
No warnings	Audi	script-html.js	32
7 info	Seat	script-html.js	35
No verbose	Peugeot	script-html.js	35

Fuentes

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/Operators/Remainder>

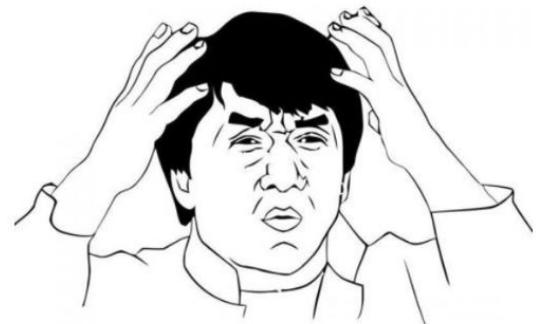
DOM

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.

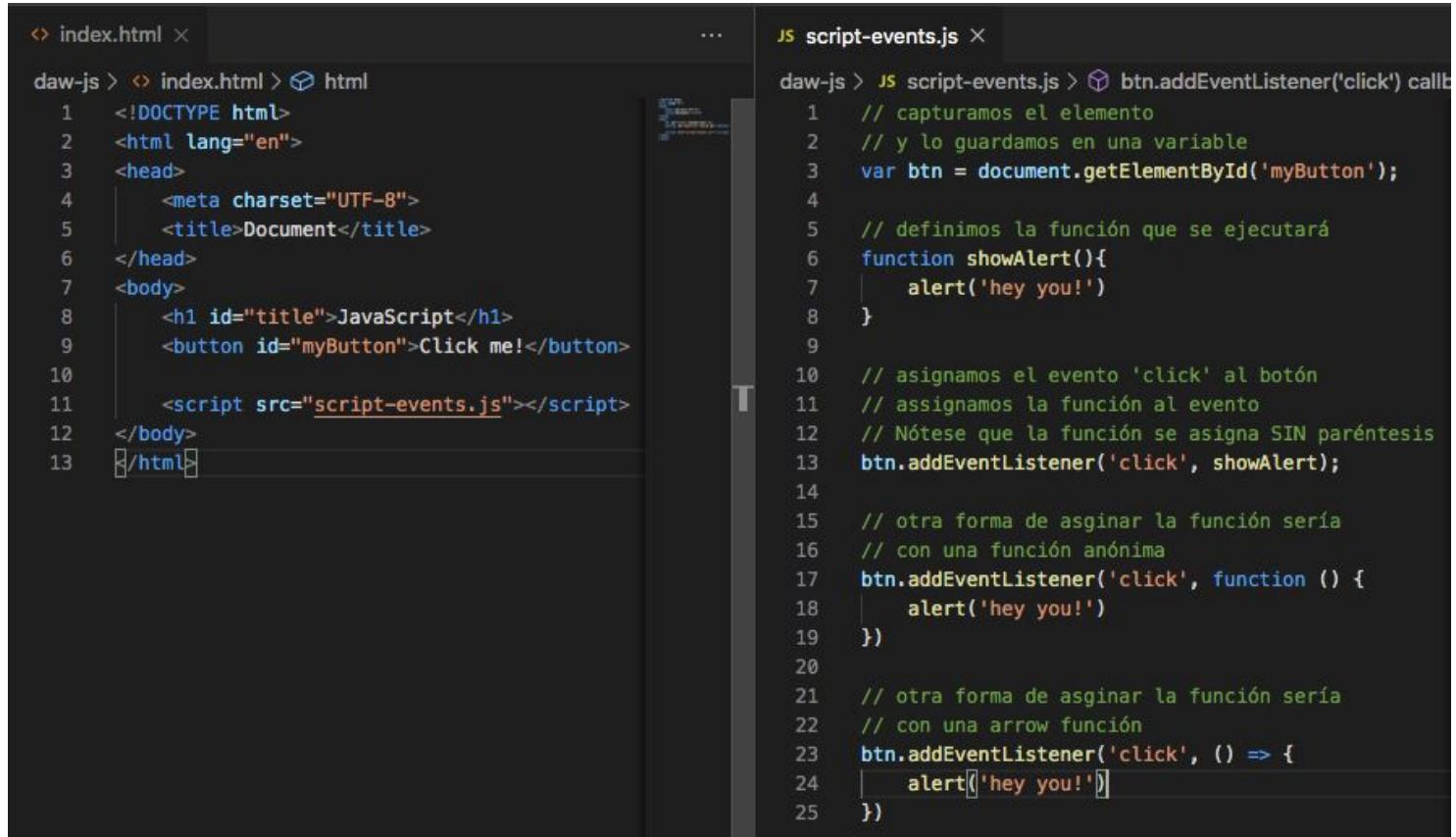
```
var myElement = document.getElementById('myButton');
```

Event Listener: addEventListener()

Un **event listener** se utiliza 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()**



The image shows a code editor interface with two files open:

- index.html**:
A simple HTML document with a title, a button, and a script tag linking to `script-events.js`.

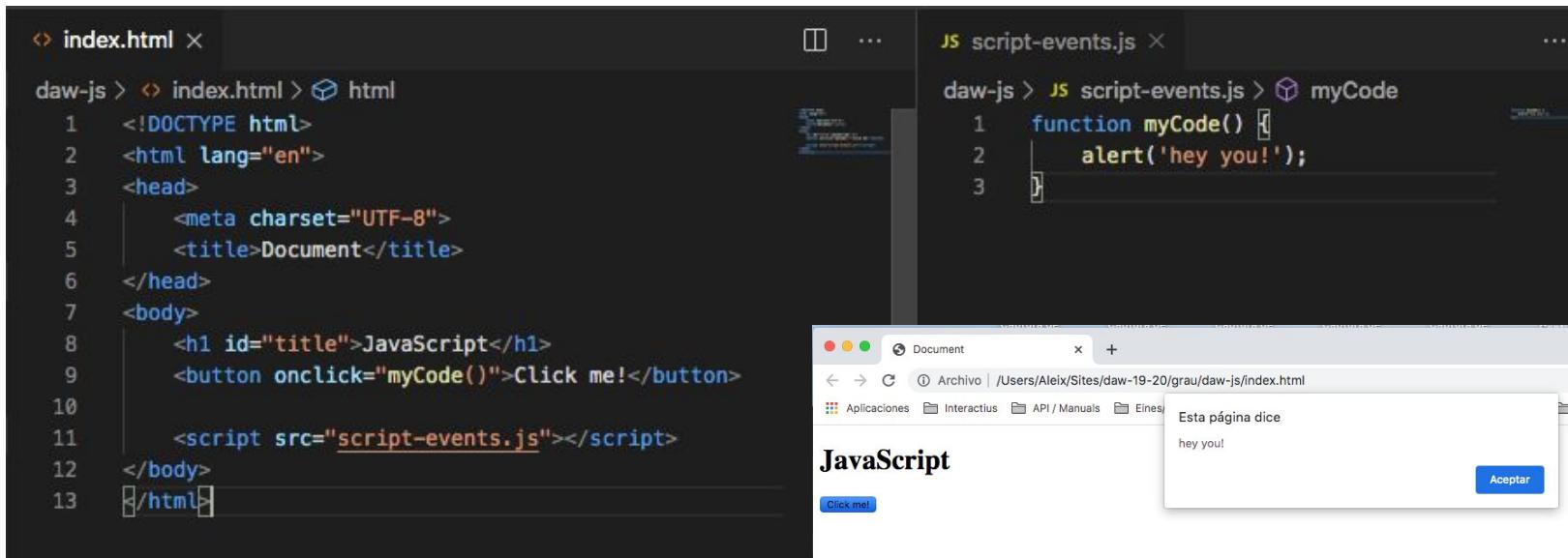
```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6   </head>
7   <body>
8     <h1 id="title">JavaScript</h1>
9     <button id="myButton">Click me!</button>
10    <script src="script-events.js"></script>
11  </body>
12 </html>
```
- script-events.js**:
A JavaScript file containing three examples of adding an event listener to the button.

```
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 // assignamos 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 and a browser window displaying a JavaScript alert.

index.html:

```
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 onclick="myCode()">Click me!</button>
10
11     <script src="script-events.js"></script>
12 </body>
13 </html>
```

script-events.js:

```
daw-js > JS script-events.js > myCode
1  function myCode() [
2      alert('hey you!');
3  ]
```

Browser Window:

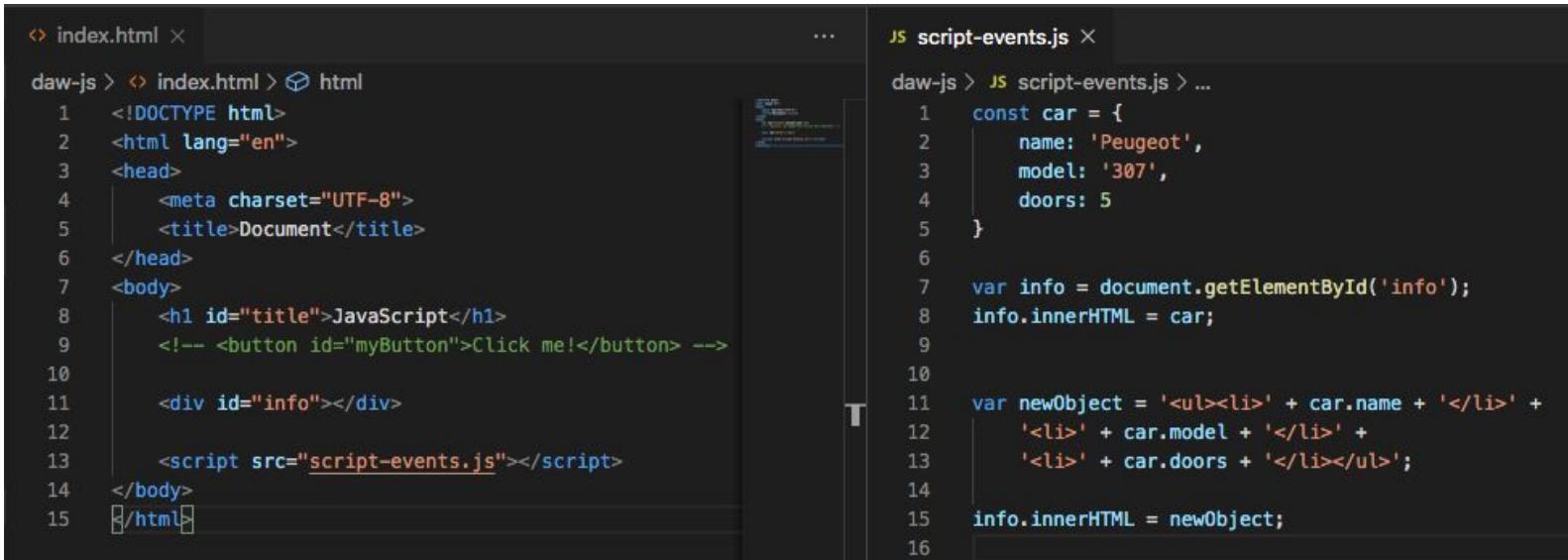
A Mac OS X style window titled "Document". The address bar shows "/Users/Aleix/Sites/daw-19-20/grau/daw-js/index.html". The main content area displays a "JavaScript" page with a "Click me!" button. A JavaScript alert dialog is open, displaying the message "Esta página dice" followed by "hey you!". There is an "Aceptar" (Accept) button at the bottom right of the dialog.

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.



The image shows a code editor interface with two files open:

- index.html**:
A simple HTML document structure. It includes a title, a heading, a button, and a div element. A script tag at the bottom points to `script-events.js`.

```
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>
```
- script-events.js**:
A JavaScript file that defines a variable `car` containing object properties for a Peugeot 307. It then creates a new string `newObject` which is an ul list item containing the car's name, model, and doors. Finally, it updates the innerHTML of the `info` div with this new object.

```
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;
```

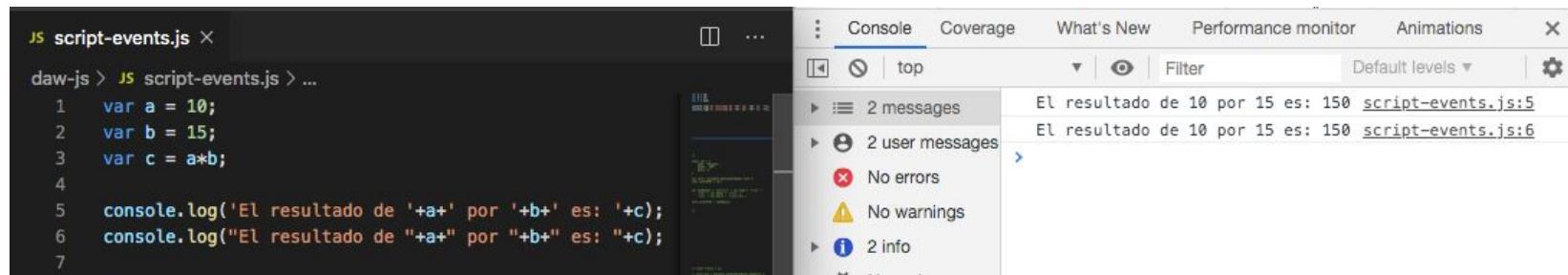
Interpolación de variables

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 screenshot shows a browser's developer tools interface with the "Console" tab selected. On the left, there is a code editor window displaying a JavaScript file named "script-events.js". The code contains the following lines:

```
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
```

On the right, the console output pane shows two log statements:

- Line 5: El resultado de 10 por 15 es: 150 script-events.js:5
- Line 6: El resultado de 10 por 15 es: 150 script-events.js:6

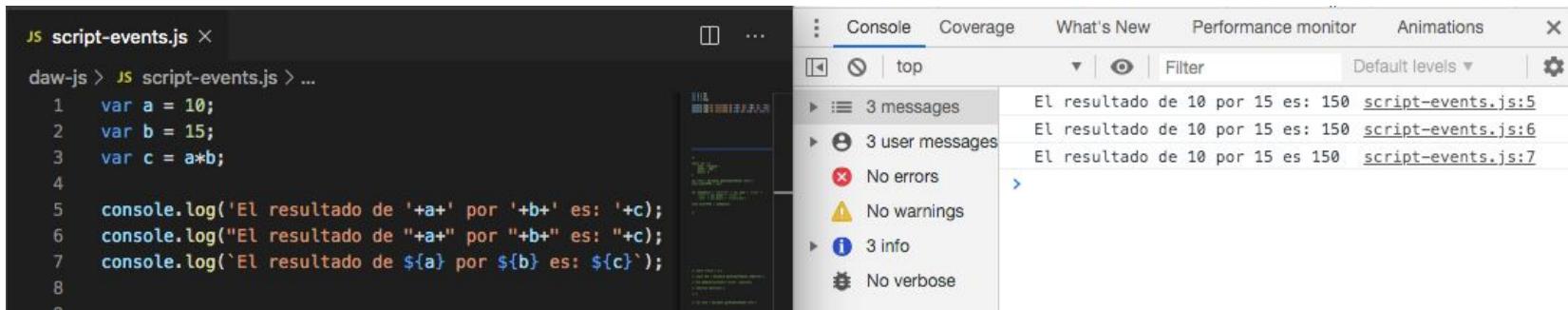
The console pane also includes a sidebar with message counts and a settings gear icon.

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 browser's developer tools with the "Console" tab selected. On the left, there is a code editor window titled "script-events.js" containing the following JavaScript code:

```
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 console.log(`El resultado de ${a} por ${b} es: ${c}`);
8
```

The right side of the interface shows the console output. It lists three messages, all of which are user messages (indicated by a person icon). 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
- El resultado de 10 por 15 es 150 script-events.js:7

Below the messages, there are several status indicators:

- 3 messages
- 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 donde 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 sobreescribe la variable, sió 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 // sobreescrivimos 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 sobreescribimos 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() {
    // sobreescrivimos 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.

IMPORTANTE: 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
<div id="myDiv">	document.getElementById('myDiv')
<input type="radio" name="colors">	document.getElementsByName('colors')
<label>	document.getElementsByTagName('label')
<button class="cta">	document.getElementsByClassName('cta')
<h1 class="title">	document.querySelector('.title')
...	document.querySelectorAll('ul li')

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.getElementById...`) y añadírselo:

```
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) ;
```

```
childElementCount: 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").classListadd("miclase");
```

classList.remove

elimina una clase del elemento

```
document.getElementById("id_elemento").classListremove("miclase");
```

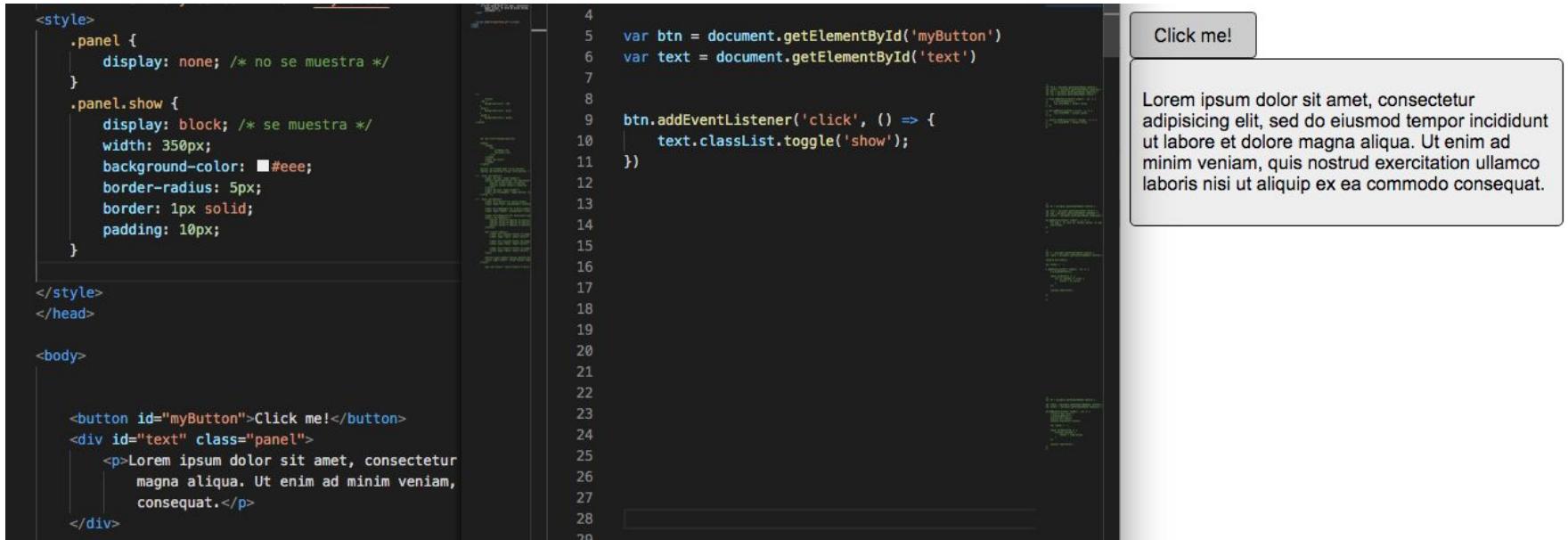
classList.toggle

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



The image shows a screenshot of a web browser displaying a simple HTML page. On the left, the browser's developer tools are visible, showing the DOM tree and the CSS styles applied to the elements.

```
<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>
</body>
```

The code includes a CSS block that defines a class `.panel` with `display: none` and a class `.panel.show` with styling like `width: 350px`, `background-color: #eee`, and `border-radius: 5px`. The body contains a button with the ID `myButton` and a `div` with the ID `text` and class `panel`. The `div` contains a single `p` tag with placeholder text.

On the right, the browser window shows the button "Click me!" and the panel below it, which is currently hidden (display: none). To the right of the panel, there is a large text area containing placeholder text: "Lorem ipsum dolor sit amet, consectetur adipisicing 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](#)