

UNIDAD 2. JS - Básico



3.1. Javascript

JavaScript, es uno de los más potentes e importantes lenguajes de programación en la actualidad, por tres enfoques claros: es útil, práctico y está disponible en cualquier navegador web.

JavaScript es creado por **Brendan Eich** y vio la luz en el año 1995 con el nombre de LiveScript, que luego fue nombrado JavaScript, nace como un lenguaje sencillo destinado a añadir algunas características interactivas a las páginas web. Sin embargo, hoy en día ha crecido de manera acelerada y es el lenguaje de programación que se utiliza en casi todos los sitios web en el mundo.

Algunas características de JavaScript:

1. Es Liviano.
2. Multiplataforma, ya que se puede utilizar en Windows, Linux o Mac o en el navegador de tu preferencia.
3. Prototipado, debido a que usa prototipos en vez de clases para el uso de herencia.
4. Orientado a objetos y eventos.
5. Es Interpretado, no se compila para poder ejecutarse.
6. No tipado (o tipado débil)

3.2. Incluir JS en un HTML

Yo conozco 4 formas de incluir JS en tu documento HTML:

- Utilizando las etiquetas "script":
 - `<script language="Javascript">"CÓDIGO JS"</script>` => obsoleto
 - `<script type="text/javascript">"CÓDIGO JS"</script>` => EN XHTML
 - `<script>"CÓDIGO JS"</script>` => EN HTML5
- En un archivo externo:
 - `<script type="text/javascript" src="myscripts.js"></script>` => XHTML
 - `<script src="myscripts.js"></script>` => HTML5
- Utilizando eventos. Por ejemplo:
`<p onclick="AQUÍ SE PONDRÍA EL CÓDIGO JS">Un párrafo de texto.</p>`
- Utilizando el atributo href de los enlaces:
`texto del enlace`

3.3. Hola Mundo.

En este caso también tenemos varias opciones de mostrar el mensaje “Hola Mundo”:

- Usando la función alert: `alert("hola mundo");`
- Usando el método write del objeto document: `document.write("hola mundo");`
- Usando el objeto console: `console.log("hola mundo");`

Este último método también se puede utilizar para depurar.

Actividad - Hola Mundo: *Crear nuestro primer fichero html con un script (javascript) que muestre ‘Hola mundo’, dentro de la etiqueta ‘body’.*

3.4. Etiqueta noscript

A veces un navegador no es capaz de interpretar JS (ya sea porque el navegador no tiene esa opción o porque el usuario ha desactivado dicha opción en las opciones del navegador).

Por esta razón, es aconsejable poner un mensaje indicando que nuestra página necesita JS para su correcto funcionamiento. Para esto podemos utilizar la etiqueta noscript:

```
<noscript>
  <p>La página que estás viendo requiere para su funcionamiento el uso de JavaScript.</p>
</noscript>
```

Actividad - noscript: *En el fichero de la actividad anterior, añadir la etiqueta noscript, y abrirlo desde el navegador. Prueba a ver como lo muestra el navegador, activando y desactivando javascript en el navegador.*

3.5. Sintaxis

La sintaxis de JavaScript es muy similar a la de otros lenguajes de programación como Java y C. Las normas básicas que definen la sintaxis de JavaScript son las siguientes:

- **No se tienen en cuenta los espacios en blanco y las nuevas líneas** (las podemos usar para poner más “ordenado” nuestro código)
- **Case sensitive** (se distinguen las mayúsculas y minúsculas)
- **Tipado blando o débilmente tipado** (No se define el tipo de las variables): al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.
- **No es necesario terminar cada sentencia con el carácter de punto y coma (;)**: pero es muy recomendable. [Algunas razones para hacerlo.](#)
- **Se pueden incluir comentarios**: con `//` para una línea o con `/*` y `*/` para múltiples líneas

Actividades - Hola Mundo (avanzado)

1. Generar un fichero html con un párrafo y un enlace en el body. El documento HTML, tiene que mostrar los siguientes mensajes con javascript:

- Hola desde alert en head
- Hola desde alert dentro de la etiqueta body
- Hola desde alert después de la etiqueta /body
- Hola desde un fichero externo de JS
- Hola desde document.write antes del párrafo
- Hola desde document.write después del párrafo
- Hola desde alert, lanzado al hacer click en el párrafo
- Hola desde alert, lanzado al hacer click en el enlace
- Hola desde el objeto console

Además, hay que añadir algunos comentarios que expliquen el funcionamiento del código, así como un mensaje de aviso para los navegadores que no tengan activado el soporte de JavaScript.

3.6. Variables

Una variable es un elemento que se emplea para **almacenar** y hacer referencia a otro valor.

En JS se crean con la palabra “reservada” **var** o **let**:

```
var variable1; // En esta línea se declara la variable “variable1”
variable1= 6; // En esta línea se inicializa la variable
let variable2 = 3; // En esta línea se declara la variable y se inicializa
let resultado = variable1 + variable2;
```

Además, en JS, **no es necesario declarar la variable**, aunque sí aconsejable:

```
variable1 = 5;
variable2 = 4;
resultado = variable1 + variable2;
```

El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:

- Sólo puede estar formado por letras, números y los símbolos \$ (dólar) y _ (guión bajo).
- El primer carácter **NO** puede ser un número.

La versión de javascript que vamos a utilizar es la ECMAScript 6, o más conocida como **ES6** (publicada en 2015). En este enlace podéis ver más información sobre las diferentes versiones de javascript:

<https://desarrolloweb.com/articulos/494.php>

A partir de esta versión, podemos usar **let** y **const** para declarar variables.

“let” y “var”, tienen la misma función, declarar variables, pero con algunas diferencias (que ya veremos cuando veamos el ámbito de las variables. Pero actualmente se **recomienda** usar “**let**”, frente a “var”.

En cuanto a “const”, nos sirve para declarar **constantes**, es decir, variables que no pueden cambiar de valor:

```
const pi = 3.1416;
pi = 1; // Esta línea produce el error: TypeError: Assignment to constant variable
```

3.7. Tipos de variables

¿¿¿Tipo de variables???, pero si hemos dicho que no hace falta indicar el tipo de variable al declararla.

Si, pero en función del valor que le asignemos, se genera un tipo de variable u otra. De esta forma podemos tener variables **numéricas** (entero o decimal), **cadenas de texto** (strings), **arrays**, **booleanos**, **objetos**, ...

En cuanto a los strings, podemos usar caracteres especiales utilizando el carácter de escape “\”:

\n	Una nueva línea
\t	Tabulador
\'	Comilla simple
\"	Comilla doble
\\	Barra inclinada

Para saber el tipo de variable, podemos utilizar la función nativa “typeof”, que devuelve una cadena indicando el tipo del operando. Se puede usar de estas dos formas:

```
typeof operando
typeof ( operando )
```

3.8. Funciones alert, prompt y confirm

Las 3 funciones muestran una pequeña ventana con un texto, aunque cada una tiene sus peculiaridades:

```
// Muestra un mensaje
alert(mensaje);
```

```
/*
Muestra un mensaje, un campo de texto y dos botones. Devuelve el string introducido en el campo de
texto (o NULL en caso de pulsar ‘Cancelar’)
*/
```

```
result = prompt(title[, default]);
https://www.anerbarrena.com/javascript-prompt-js-5509/
```

```
// Muestra un mensaje, y dos botones. Devuelve true o false en función del botón pulsado.
result = confirm(question);
```

3.9. Operadores

Operador de asignación

```
=      Asignación. Ej: var1 = 5
```

Operadores lógicos

! Negación. Ej: `var1 = true; alert(!var1);`
&& AND. Ej: `result = var1 && var2;`
|| OR. Ej: `result = var1 || var2;`

Operadores matemáticos

+ Suma
- Resta
* Multiplicar
/ Dividir
% Operador Módulo (calcula el resto de una división)
** Operador exponencial (`2 ** 3` => dos elevado a tres)

Se pueden combinar con el operador asignación. Por ejemplo, con la suma sería el operador `'+='`. A continuación, se muestran las equivalencias;

<code>var1 += 5;</code>	<code>=></code>	<code>var1 = var1 + 5;</code>
<code>var1 -= 5;</code>	<code>=></code>	<code>var1 = var1 - 5;</code>
<code>var1 *= 5;</code>	<code>=></code>	<code>var1 = var1 * 5;</code>
<code>var1 /= 5;</code>	<code>=></code>	<code>var1 = var1 / 5;</code>
<code>var1 %= 5;</code>	<code>=></code>	<code>var1 = var1 % 5;</code>

También tenemos los operadores de incremento y decremento:

<code>var1++;</code>	<code>=></code>	<code>var1 = var1 + 1;</code>
<code>var1--;</code>	<code>=></code>	<code>var1 = var1 - 1;</code>

Operador de concatenación

+ Concatena strings

Operadores relacionales

> Mayor
< Menor
>= Mayor o igual
<= Menor o igual
!= Distinto
== Igual

Ejemplo:

```
let num1 = 3;
let num2 = 5;
resultado = num1 > num2 ; // resultado = false
resultado = num1 < num2 ; // resultado = true
```

3.9.1 Operadores '=== y '!=='

Los operadores === y !== son los operadores de comparación estricta. Esto significa que, si los operandos tienen tipos diferentes, no son iguales. Es decir, no hace ninguna conversión al comparar. Por ejemplo,

```
1 === "1" // false
1 !== "1" // true
null === undefined // false
```

Sin embargo, los operadores == y != son los operadores de comparación relajada. Es decir, si los operandos tienen tipos diferentes, JavaScript trata de **convertirlos** para que fueran comparables. Por ejemplo,

```
1 == "1" // true
1 != "1" // false
null == undefined // true
```

3.9.2 Valor UNDEFINED

Cuando intentamos utilizar una variable que no está declarada, JS nos da un error, pero si intentamos usar una variable que no tiene asignado ningún valor, JS utiliza el valor 'undefined' para esa variable.

```
let var1 = var2 + 4; // Esto provoca el error 'var2 is not defined', ya que 'var2' no está declarada.
```

Si necesito comprobar si una variable está o no está definida:

```
let x;
if (x === undefined) {
    txt = "x is undefined";
} else {
    txt = "x is defined";
}
```

También puedo utilizar el operador '==':

```
let x;
if (x == undefined) {
    txt = "x is undefined";
} else {
    txt = "x is defined";
}
```

3.9.3. NaN

NaN, (del inglés, "Not a Number") JavaScript emplea el valor NaN para indicar un valor numérico no definido (por ejemplo, la división 0/0).

```
let numero1 = 0;
let numero2 = 0;
alert(numero1/numero2); // se muestra el valor NaN
```

O también:

```
let var2;  
let var1 = var2 + 4; // var2 es 'undefined', con lo cuál var1 toma el valor NaN
```

Actividades - Tipos de variables, operadores, typeof, ...

1. *Escribir el código para mostrar en la consola:
el producto de los números 12 y 13.
la diferencia (resta) entre 321 y 213.
el resultado de dividir 301 entre 3.
el resto de la división de 301 entre 3.*
2. *Repetir el ejercicio anterior, pero esta vez usando variables. Muestra también el resultado de typeof de cada una de las variables.
¿Que devuelve typeof si le pasas como parámetro una variable que no existe?*

3.10. Conversión de variables

¿Qué pasa si sumamos un string y un número?

```
let var1 = '1' + 2;  
console.log(var1);  
console.log( typeof(var1) );
```

En estos casos JS no da error, simplemente convierte una de las dos variables para poder realizar la operación.

¿Qué obtendremos con el siguiente código?

```
console.log( 7 + 7 + 7 ); // 21 (número)  
console.log( 7 + 7 + "7" ); // 147 (string)  
console.log( "7" + 7 + 7 ); // 777 (string)  
console.log( 10 - 2 - 4 ); // 4 (número)  
console.log( 10 - 2 - "4" ); // 4 (número)  
console.log( "10" - 2 - 4 ); // 4 (número)
```

Conclusión: *JS intenta convertir los números en cadena (para la suma).
JS intenta convertir las cadenas en números (excepto para la suma).*

Veamos ahora estos casos:

```
let foo = '5';  
console.log( typeof ( foo ) );  
console.log( typeof ( foo - 0 ) );  
console.log( typeof ( foo * 1 ) );  
console.log( typeof ( foo / 1 ) );  
console.log( typeof ( foo++ ) );
```


Vemos que tenemos distintas opciones en JS de convertir un string en un número. Esto también lo podemos conseguir usando las funciones 'parseInt' o 'parseFloat':

```
let foo = '5';
console.log( typeof parseInt( foo ) );
console.log( typeof parseFloat( foo ) );
```

Y para el caso contrario, el método 'toString':

```
let foo = 1;
console.log( typeof ( foo.toString() ) );
```

3.11. Depurando

La mejor forma de depurar es ejecutando el código paso a paso. Esa funcionalidad nos la dan los navegadores (al menos casi todos).

Al pulsar F12 en el navegador se nos abre un conjunto de herramientas, entre ellas un depurador. Otra forma de abrir estas herramientas es pulsando con el botón derecho en el navegador y elegir la opción "Inspeccionar".

Actividades - Operadores, Prompt, ...

1. Si nos fijamos en estas líneas de código

```
num1 = 5;
resultado = num1 = 3; // ¿Cuánto vale aquí 'resultado'?
resultado = num1 == 3; // ¿Cuánto vale aquí 'resultado'?
```

Nota: Vamos a aprovechar esta actividad para aprender a utilizar el depurador del navegador y 'observar' una variable.

2. Escribir el código necesario para que realice esta operación " $v1 * v2 + v3$ " y muestre el resultado por consola. Siendo los valores de $v1$, $v2$ y $v3$: $v1=2$ $v2=4$ $v3=5$

*A continuación, hacer lo mismo, pero con esta operación " $v1 + v2 * v3$ ". Antes de ejecutarlo, pensad qué resultado obtendremos. ¿Has acertado?*

¿Qué orden sigue JS para realizar las operaciones? ¿Cómo podemos cambiarlo?

3. ¿Cuál sería el resultado de las siguientes operaciones? $a=5$; $a/=2$; $a+=1$; $a*=3$; $a--$; $a++$;
4. Suma tres números tecleados por usuario y calcula la media, mostrando en la consola:
"La suma es: RESULTADO_SUMA"
"La media es: MEDIA"
5. Pide al usuario una cantidad de "millas náuticas" y muestra la equivalencia en metros.
6. Dado el siguiente código:

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

Y este otro:

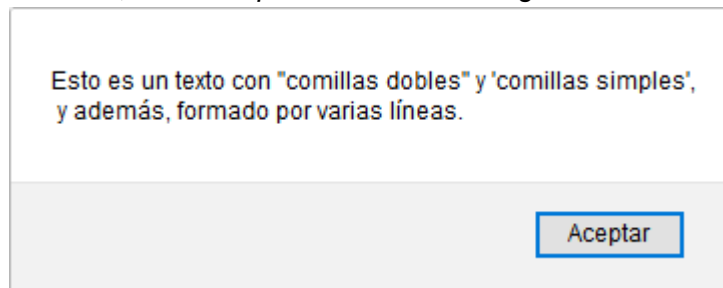
```
let a = 5;  
let b = ++a;
```

¿Tendrá la variable 'b' el mismo valor en ambos scripts? ¿Por qué?

7. ¿Que mostrará el siguiente código en la consola? ¿Por qué?

```
let foo = 'hola';  
console.log( parseInt(foo) );
```

8. Mostrar un mensaje con alert, como el que vemos en la imagen:



3.12. Estructuras de control

Estructura if

```
if (condición) {  
    ...  
}  
  
if (condición) linea_de_codigo;
```

Estructura if...else

```
if(condición) {  
    ...  
}  
else {  
    ...  
}  
  
// If/else “resumido” o “compacta”  
condición ? expr1 : expr2;  
  
// Ejemplo:  
let var1 = "La Cuota es de: " + (isMember ? "2 euros" : "10 euros")
```

Estructura if...else if...else

```
if(condición) {  
    ...  
}  
else if (condición){  
    ...  
}else {  
    ...  
}
```

Estructura for

```
for(inicialización; condición; actualización) {  
    ...  
}
```

Estructura for...in

```
for(indice in array) {  
    ...  
}
```

Estructura while

```
while(condición) {  
    ...  
}
```

Estructura do...while

```
do {  
    ...  
} while(condición);
```

Estructura switch

```
switch(variable) {  
    case valor1:  
        ...  
        break;  
  
    case valor2:  
        ...  
        break;  
  
    case valor3:  
    case valor4:  
        ...  
        break;  
  
    case valor5:  
        ...  
    case valor6:  
        ...  
        break;  
  
    case valorN:  
        ...  
        break;  
  
    default:  
        ...  
        break;  
}
```

Instrucciones break y continue

La instrucción **break**, 'salta fuera' del bucle (se sale del bucle).

La instrucción **continue**, 'salta una vuelta' del bucle.

3.13. Capturar errores

```
try {  
    funcionquenoexiste();  
}  
catch(err) {  
    alert(err.message);  
}
```

Actividades - Estructuras de control

De aquí en adelante, cosas que hay que tener en cuenta a la hora de realizar los ejercicios:

- Cuidado con lo que devuelve el prompt => es un **string**
- Si no funciona el código => mirar la consola (**F12**) para ver si hay errores de sintaxis.
- Si no funciona el código y no hay errores de sintaxis => depurar (**F12**)

1. Crea un programa que pida al usuario un número y diga si es par.
2. Crea un programa que pida al usuario dos números y diga cuál es el mayor de ellos.
3. Crea un programa que pida al usuario dos números y diga si el primero es múltiplo del segundo.
4. Crea un programa que pida al usuario dos números. Si el segundo no es cero, mostrará el resultado de dividir el primero entre el segundo. Por el contrario, si el segundo número es cero, escribirá "Error: No se puede dividir entre cero".
5. Crea un programa que pida al usuario dos números y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.
6. Necesitamos un programa que muestre los números del 10 al 20, ambos inclusive. Y a continuación, muestre los números del 20 al 10, es decir, de mayor a menor.
7. Crea un programa que escriba en pantalla los números del 1 al 50 que sean múltiplos de 3.
8. Crea un programa que muestre los números del 100 al 200 (ambos incluidos) que sean divisibles entre 7 y a la vez entre 3.
9. Crea un programa que muestre la tabla de multiplicar del 9:
 9 x 0 = 0
 9 x 1 = 9
 9 x 2 = 18
 ...
 9 x 10 = 90
10. Crea un programa que muestre los primeros ocho números pares: 2 4 6 8 10 12 14 16.
11. Crea un programa que muestre los números del 15 al 5, descendiendo.

12. Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y muestre un rectángulo formado por esa cantidad de asteriscos:

```
****
****
****
```

13. Lo mismo que el ejercicio anterior, pero esta vez mostrará una diagonal con otro carácter:

```
O***
*O**
**O*
```

14. Crea un programa que pida al usuario su identificador y contraseña de manera indefinida hasta que introduzca el identificador “alibaba” y la contraseña “sesamo”. Si al tercer intento no ha introducido los datos correctos, mostrará un mensaje diciendo que el identificador/contraseña no son correctos.

15. Crea un "calculador de cuadrados": pedirá al usuario un número y mostrará su cuadrado. Se repetirá mientras el número introducido no sea cero.

16. Escribe un código que genere una excepción (por ejemplo, llamar a una función que no exista). Capturar el error y mostrar un mensaje de aviso.

17. Crea un programa que contenga un bucle sin fin que muestra el mensaje "Hola" con alert. Cuando llegue a 10 vueltas, se detendrá.

18. A partir del siguiente código html:

```
<!DOCTYPE html><html lang="es">
<head><meta charset="utf-8"><title>Tabla</title>
<style> table td{ border: 1px solid #555; padding: 10px; } </style>
<body></body></html>
```

Genera el código necesario (usando dos “for” anidados) para generar esta tabla dentro del **body** (no hace falta ponerles borde a las celdas, ya lo tiene puesto en los estilos):

<https://parzibyte.me/blog/2019/08/22/dibujar-tabla-html-dinamica-javascript/>

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							
6							
7							
8							

3.14. Arrays

Un Array (vector o matriz) es una zona de almacenamiento contiguo que contiene una serie de **elementos del mismo tipo**, que forman los elementos de la matriz.

Para crear un array, podemos utilizar varias sintaxis distintas:

```
let array_name = [item1, item2, ...];
let array_name = new Array(item1, item2, ...);
let array_name = Array(item1, item2, ...);
```

Sin embargo, en JS no existen los arrays como tal, aunque tenemos el objeto “Array” que nos permite usarlos, pero con algunas diferencias, por ejemplo, la diferencia más visible, es que podemos crear un array en JS con **elementos de distinto tipo**. Por ejemplo, en JS, puedo hacer:

```
let miArray = ["Juan", 4, "Badajoz"];
```

Para acceder a un elemento, utilizamos el “**índice**”. El primer elemento tiene índice ‘cero’:

```
miArray[0] => estoy accediendo al primer elemento del array ‘miArray’, en este caso sería “Juan”
```

Para saber el número de elementos de un array, usamos el método ‘length’:

```
miArray.length => devolverá el valor 3
```

Añadir un elemento, al final del array:

```
miArray[3] = "Soltero";           // si ya conocemos el número de elementos
miArray.push("Soltero");          // usando el método push
miArray[miArray.length] = "Soltero"; // o usando el elemento .length
```

Para recorrer un array, podemos usar la estructura for:

```
for(let i=0;i<miArray.length;i++){
    console.log("El elemento " + i + " toma el valor" + miArray[i]);
}
```

Para recorrer un array, podemos usar la estructura for ... in:

```
for(let indice in miArray)
{
    console.log("El elemento " + indice + " toma el valor" + miArray[indice]);
}
```

En el siguiente enlace podemos ver más métodos del objeto 'Array':

<https://devcode.la/tutoriales/manejo-de-arrays-en-javascript/>

Array multidimensionales (no es más arrays dentro de arrays):

```
let miArrayFila1 = [ "fila1 col1", "fila1 col2" ]; // Array unidimensional
let miArrayFila2 = [ "fila2 col1", "fila2 col2" ]; // Array unidimensional
let miArrayTabla = [ miArrayFila1, miArrayFila2 ]; // Array de dos dimensiones
```

Para acceder a un elemento de un array multidimensional:

miArrayTabla[1][0] => estaría accediendo a la columna 1 de la fila 2

Array asociativos

EN JAVASCRIPT NO EXISTEN LOS ARRAY ASOCIATIVOS, SE PUEDE IMITAR SU FUNCIONALIDAD USANDO OBJETOS, PERO SABRIENDO LO QUE SE HACE, YA QUE POR EJEMPLO ".length" NO FUNCIONARÁ. PERO HAY OTROS MÉTODOS PARA RECORRER EL ARRAY.

WARNING !!

If you use named indexes, JavaScript will redefine the array to a standard object. After that, some array methods and properties will produce incorrect results.

Example:

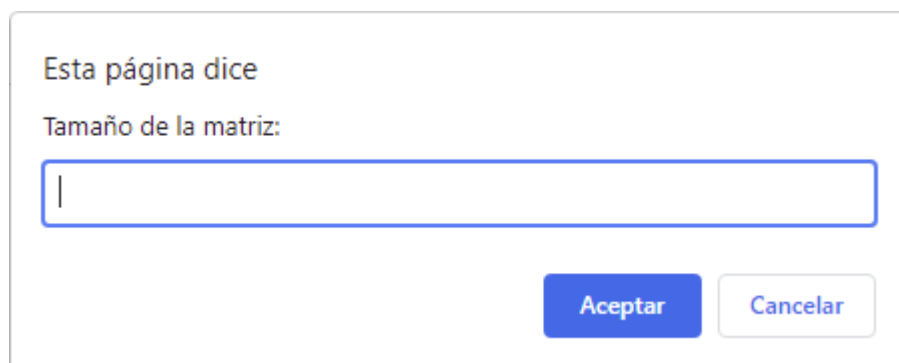
```
let person = [ ];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
console.log(person["firstName"]);
let x = person.length; // person.length will return 0
let y = person[0];     // person[0] will return undefined
```

Sin embargo, podemos recorrer el array asociativo (objeto) sin tener que recurrir al .length


```
for (let indice in person){  
  console.log("El índice " + indice + " vale " + person[indice]);  
}
```

Actividades - Arrays

1. Crea un programa que pida al usuario 6 números y luego los muestre en orden inverso (pista: usa un array para almacenarlos y "for" para mostrarlos).
2. Crea un programa que pregunte al usuario cuantos números va a introducir (por ejemplo, 5), le pida todos esos números, los guarde en un array y finalmente calcule y muestre la media de esos números.
3. Crea un array de 100 elementos. El valor del primer elemento será 1, valor del segundo elemento será 2, y así sucesivamente, de tal forma que el último elemento del array tendrá el valor 100. Calcula la media de todos los elementos del array.
4. Un programa que pida al usuario 10 números, calcule su media y luego muestre los que están por encima de la media.
5. Desarrolla un programa que pedirá nombres al usuario hasta que se introduzca un nombre vacío, momento en el que dejarán de pedirse más nombres y se mostrará en pantalla la lista de los nombres que se han introducido ordenados alfabéticamente.
6. Escribe un script para generar un array de dos dimensiones (matriz). Se le pedirá al usuario primero el tamaño de la matriz (que será cuadrada):

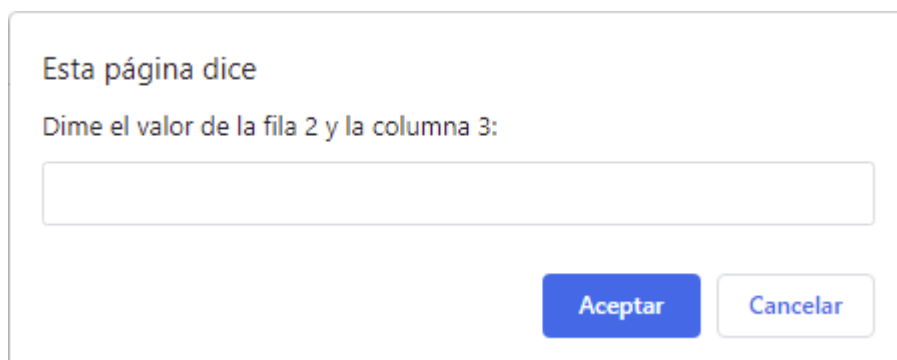


Esta página dice

Tamaño de la matriz:

Aceptar Cancelar

Y a continuación le irá pidiendo el valor de cada celda.



Esta página dice

Dime el valor de la fila 2 y la columna 3:

Aceptar Cancelar

Por último, mostrar en el body una tabla cuadrada mostrando en cada celda el valor correspondiente a su posición en la matriz. Por ejemplo, si el tamaño de la matriz es de 3x3 y en cada celda se ha indicado el valor "Celda fila-columna", debería aparecer una tabla como esta:

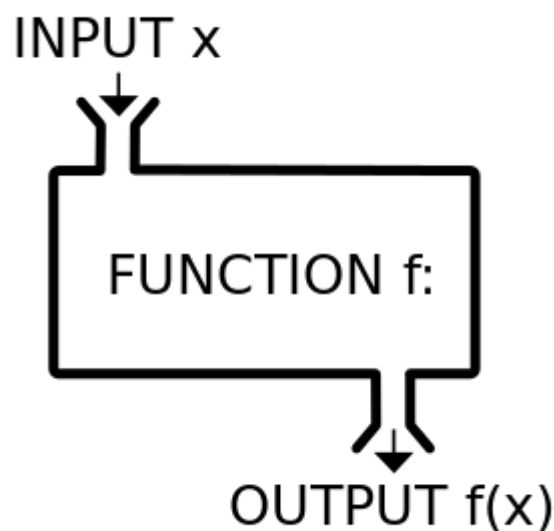
Celda 1-1	Celda 1-2	Celda 1-3
Celda 2-1	Celda 2-2	Celda 2-3
Celda 3-1	Celda 3-2	Celda 3-3

7. **[OPCIONAL]** Dado un array con elementos repetidos, escribe el código para que saber cuál es el elemento que más veces aparece en el array.
8. **[OPCIONAL]** Modificar el ejercicio anterior, de tal forma que muestra en la consola el número de veces que aparece cada ítem en el array, pero ordenamos de mayor a menor.

Para esta actividad es probable que necesitéis saber cómo eliminar un elemento de un array asociativo. Con el método "slice", en array normales se puede hacer, pero, ¿funciona igual en array asociativos? En caso de que no funcione, ¿se puede eliminar un elemento de un array asociativo? ¿como?

3.15. Funciones

Una **función** es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor. Además, las **funciones** pueden tomar parámetros que modifiquen su funcionamiento.



Creación de funciones propias

En cuanto se desarrolla una aplicación un poco compleja, la utilización de funciones es indispensable. En cualquier aplicación vamos a encontrarnos con trozos de código que se repiten una y otra vez. La solución es la utilización de funciones.

Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar todas las veces que se quiera.

Sintaxis:

```
function NOMBRE_DE_LA_FUNCION {  
    /* conjunto de instrucciones*/  
}
```

Ejemplo:

```
function printDosEspacios(){  
    document.write('&nbsp;&nbsp;');  
}
```

También podemos utilizar argumentos (o parámetros) para pasarle a la función:

Ejemplo:

```
function printEspacios(numItems){  
    for (var i=1; i<=numItems; i++){  
        document.write('&nbsp;');  
    }  
}  
  
function suma(num1, num2){  
    document.write(num1+num2);  
}
```

En javascript todos los valores se pasan **por valor**. Los únicos que se pasan por **referencia** son los objetos, arrays y funciones (que en realidad son objetos del tipo function).

A veces, nos interesa devolver un valor, con “**return**” :

```
function calculaPrecioTotal(precio) {  
    let impuestos = 1.21;  
    let gastosEnvio = 10;  
    let precioTotal = ( precio * impuestos ) + gastosEnvio;  
    return precioTotal;  
}  
  
let precioTotal = calculaPrecioTotal(23.34);
```

Funciones y propiedades básicas de JavaScript

A continuación, se pueden ver algunas de las funciones (o métodos) nativas de javascript más utilizadas.

Funciones para cadenas de texto:

length: devuelve el número de caracteres de un string. Ej: *let cad = “hola”; console.log(cad.length)*

+ : concatenar cadenas

concat(): concatenar cadenas:

```
let mensaje1 = "Hola";
let mensaje2 = mensaje1.concat(" Mundo"); // mensaje2 = "Hola Mundo"
```

toUpperCase(): transforma todos los caracteres en mayúsculas:

```
let mensaje1 = "Hola";
let mensaje2 = mensaje1.toUpperCase(); // mensaje2 = "HOLA"
```

toLowerCase(): transforma todos los caracteres en minúsculas:

charAt(posicion): obtiene el carácter que se encuentra en la posición indicada:

```
let mensaje = "Hola";
let letra = mensaje.charAt(0); // letra = H
letra = mensaje.charAt(2);    // letra = l
```

indexOf(caracter o string):. calcula la primera posición en la que se encuentra el carácter (o string) indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor -1:

```
let mensaje = "Hola";
let posicion = mensaje.indexOf('a'); // posición = 3
posicion = mensaje.indexOf('b');    // posición = -1
```

lastIndexOf(carácter o string): calcula la **última** posición en la que se encuentra el carácter (o string) indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor -1:

```
let mensaje = "Hola";
let posicion = mensaje.lastIndexOf('a'); // posición = 3
posicion = mensaje.lastIndexOf('b');    // posición = -1
```

substring(inicio, final): extrae una porción de una cadena de texto. El segundo parámetro es opcional. Si sólo se indica el parámetro inicio, la función devuelve la parte de la cadena original correspondiente desde esa posición hasta el final:

```
let mensaje = "Hola Mundo";
let porcion = mensaje.substring(2); // porcion = "la Mundo"
porcion = mensaje.substring(5);    // porcion = "Mundo"
porcion = mensaje.substring(-2); // porcion = "Hola Mundo" !!!!! OJO con esto
porcion = mensaje.substring(1, 8); // porcion = "ola Mun"
porcion = mensaje.substring(5, 0); // porcion = "Hola " (si fin<inicio, JS les da la vuelta)
porcion = mensaje.substring(0, 5); // porcion = "Hola " (OJO, hace lo mismo que la anterior)
```

split(separador): convierte una cadena de texto en un array de cadenas de texto, utilizando el 'separador' como elemento divisor.

```
let mensaje = "Hola Mundo, soy una cadena de texto!";
let palabras = mensaje.split(" ");
// palabras = ["Hola", "Mundo,", "soy", "una", "cadena", "de", "texto!"];
```

Si separador es igual a una cadena vacía (""), devuelve un array con cada uno de los caracteres:

```
let palabra = "Hola";
let letras = palabra.split(""); // letras = ["H", "o", "l", "a"]
```

replace(searchvalue, newvalue): sustituye una cadena de texto por otra. También admite expresiones regulares (que ya se verán más adelante). Por defecto solamente reemplaza la primera coincidencia.

```
let str = "Hola a todos!";  
let res = str.replace("todos", "todo el mundo"); // res => Hola a todo el mundo!
```

replaceAll(searchvalue, newvalue): sustituye una cadena de texto por otra. También admite expresiones regulares (que ya se verán más adelante). Reemplaza todas las coincidencias.

```
let str = "El script es útil. El script es interesante.";  
let res = str.replaceAll("script", "código"); // res => El código es útil. El código es interesante.
```

padStart(targetLength [,padString]): rellena la cadena actual con una cadena dada (esta cadena puede repetirse varias veces), de modo que la cadena resultante alcance una longitud dada. El relleno es aplicado desde el inicio (izquierda) de la cadena actual. Por ejemplo:

```
'abc'.padStart(10, "R"); // "RRRRRRRRabc" (rellena con un carácter hasta llegar a 10)  
'abc'.padStart(10); // "      abc" (si no se pasa el segundo parámetro, usa el espacio en blanco)  
'abc'.padStart(10, "foo"); // "foofoofabc" (también se puede rellenar con más de un carácter)  
'1'.padStart(2, "0"); // "01" (este ejemplo es útil para mostrar segundos, minutos, ...)
```

A veces necesitamos usar el método **"toString()"** para poder aplicar estos métodos.

Ejercicio: crear una función que tenga como parámetro de entrada un DNI y le ponga un 0 o no al principio según corresponda (si tiene 7 caracteres, hay que ponerle un 0 delante).

Funciones para los arrays

length, calcula el número de elementos de un array

concat(), se emplea para concatenar los elementos de varios arrays

```
let array1 = [1, 2, 3];  
array2 = array1.concat(4, 5, 6); // array2 = [1, 2, 3, 4, 5, 6]
```

join(separador), es la función contraria a split(). Une todos los elementos en una cadena de texto.

```
let array = ["hola", "mundo"];  
let mensaje = array.join(""); // mensaje = "holamundo"  
mensaje = array.join(" "); // mensaje = "hola mundo"
```

pop(), elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en 1 elemento.

```
let array = [1, 2, 3];  
let ultimo = array.pop();  
// ahora array = [1, 2], ultimo = 3
```

push(), añade un elemento al final del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
let array = [1, 2, 3];  
array.push(4);  
// ahora array = [1, 2, 3, 4]
```

shift(), elimina el primer elemento del array y lo devuelve. El array original se ve modificado y su longitud disminuida en 1 elemento.

```
let array = [1, 2, 3];
let primero = array.shift();
// ahora array = [2, 3], primero = 1
```

unshift(), añade un elemento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
let array = [1, 2, 3];
array.unshift(0);
// ahora array = [0, 1, 2, 3]
```

reverse(), modifica un array colocando sus elementos en el orden inverso a su posición original:

```
let array = [1, 2, 3];
array.reverse();
// ahora array = [3, 2, 1]
```

sort(), ordena los elementos de un array en orden ascendente

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

También se le puede pasar una función de comparación

```
array.sort(compareFunction)
```

Por ejemplo:

```
let points = [40, 100, 1, 5, 25, 10];
points.sort( function(a, b){return a-b} ); // ordena de menor a mayor (usando una función anónima)
/*
```

Cuando el método `sort()` compara dos valores, envía los valores a la función a comparar y ordena los valores según el valor que retorna (negativo, cero, positivo)

Example:

Cuando compara 40 y 100, el método `sort` llama a la función de comparación `function(40,100)`.

La función calcula $40-100$, y retorna -60 (un valor negativo).

La función `sort` ordenará 40 como un valor menor que 100.

*/

```
/* También se puede hacer sin usar funciones anónimas */
```

```
points.sort(ordena);
```

```
function ordena(a,b){
  return a-b;
}
```

Funciones para los números

NaN, (del inglés, "Not a Number") Ya sabemos que JavaScript emplea el valor NaN para indicar un valor numérico no definido (por ejemplo, la división $0/0$).

```
let numero1 = 0;
let numero2 = 0;
alert(numero1/numero2); // se muestra el valor NaN
```

isNaN(), permite saber si una variable o resultado es NaN

```

let numero1 = 0;
let numero2 = 0;
if(isNaN(numero1/numero2)) {
  alert("La división no está definida para los números indicados");
}
else {
  alert("La división es igual a => " + numero1/numero2);
}

```

Infinity, hace referencia a un valor numérico infinito y positivo (también existe el valor **-Infinity** para los infinitos negativos)

```

let numero1 = 10;
let numero2 = 0;
alert(numero1/numero2); // se muestra el valor Infinity

```

isFinite(), permite saber si una variable o resultado es “finito”.

```

if (isFinite(x)) {
  return 'Number is NOT Infinity.';
}

```

toFixed(dígitos), devuelve el número original con tantos decimales como los indicados por el parámetro dígitos y realiza los redondeos necesarios.

```

let numero1 = 4564.34567;
numero1.toFixed(2); // 4564.35
numero1.toFixed(6); // 4564.345670
numero1.toFixed(); // 4564

```

parseInt(cadena,base): Recibe una cadena y una base. Devuelve un valor numérico resultante de convertir la cadena en un número en la base indicada. Ejemplo: **parseInt("0xF", 16)** devuelve un 15.

parseFloat(cadena): Convierte la cadena en un número y lo devuelve.

Number(cadena): Convierte la cadena en un número y lo devuelve.

```

Number("10");      // returns 10
Number("10.33");   // returns 10.33
Number("10,33");   // returns NaN
Number("10 20 30"); // returns NaN
Number("10 years"); // returns NaN
Number("years 10"); // returns NaN
Number(" 10");     // returns 10
Number("10 ");     // returns 10
Number(" 10 ");    // returns 10

```

```

parseInt("10");     // returns 10
parseInt("10.33");  // returns 10
parseInt("10,33");  // returns 10

```

```

parseInt("10 20 30"); // returns 10
parseInt("10 years"); // returns 10
parseInt("years 10"); // returns NaN
parseInt(" 10"); // returns 10
parseInt("10 "); // returns 10
parseInt(" 10 "); // returns 10

parseFloat("10"); // returns 10
parseFloat("10.33"); // returns 10.33
parseFloat("10,33"); // returns 10
parseFloat("10 20 30"); // returns 10
parseFloat("10 years"); // returns 10
parseFloat("years 10"); // returns NaN
parseFloat(" 10"); // returns 10
parseFloat("10 "); // returns 10
parseFloat(" 10 "); // returns 10

```

Funciones para números con el objeto Math

Métodos

abs(x)	Retorna el valor absoluto de x
ceil(x)	Retorna x redondeado al entero más cercano por arriba
floor(x)	Retorna x redondeado al entero más cercano por abajo
round(x)	Redondea x al entero más cercano
trunc(x)	Retorna la parte entera del número x
cos(x)	Retorna el coseno de x (x es en radianes)
sin(x)	Retorna el seno de x (x es en radianes)
exp(x)	Retorna el valor de E(número de Euler) elevado a x
log(x)	Retorna el logaritmo de x
max(x,y,..., n)	Retorna el número con el mayor valor
min(x,y,..., n)	Retorna el número con el menor valor
pow(x,y)	Retorna el valor de x elevado a y
random()	Retorna un número aleatorio entre 0 y 1
sqrt(x)	Retorna la raíz cuadrada de x
tan(x)	Retorna la tangente de un ángulo

Constantes

E	Retorna el número de Euler (aproximadamente 2.718)
PI	Retorna el número PI (aproximadamente 3.14)

Ejemplos:

```

let x = Math.PI; // Retorna PI
let y = Math.sqrt(16); // Retorna la raíz cuadrada de 16

```

[Aquí](#) podéis ver el listado completo de los métodos y constantes del objeto Math.

Funciones para fechas. Objeto Date.

```
let fecha = new Date();
```

- Si no se le pasa ningún parámetro, coge la fecha actual.

- Si se le pasa 1 parámetro, se indica el número de milisegundos que han transcurrido desde el 1 de Enero de 1970.
- También se le puede pasar un solo parámetro indicando una cadena de texto indicando la fecha (aunque la cadena de texto tiene que tener un formato específico).
- Si se le pasa 3 parámetros: (año, mes, día)
- Si se le pasan 6 parámetros: (año, mes, día, hora, minuto, segundo)

Ejemplos:

```
let fecha = new Date(); // "La fecha y hora actuales"
let fecha = new Date(10000000000000); // "Sat Nov 20 2286 18:46:40"
let fecha = new Date(2009, 5, 1); // 1 de Junio de 2009 (00:00:00)
let fecha = new Date(2009, 5, 1, 19, 29, 39); // 1 de Junio de 2009 (19:29:39)
```

Una vez creado el objeto, tenemos los siguientes métodos para trabajar con él:

getDate()	Retorna el día del mes (de 1-31)
getDay()	Retorna el día de la semana (de 0-6), siendo 0 el domingo.
getFullYear()	Retorna el año
getHours()	Retorna la hora (de 0-23)
getMilliseconds()	Retorna los milisegundos (de 0-999)
getMinutes()	Retorna los minutos (de 0-59)
getMonth()	Retorna el mes (de 0-11), donde 0 es Enero.
getSeconds()	Retorna los segundos (de 0-59)
getTime()	Retorna el número de milisegundos desde el 1 de Enero de 1970 hasta la fecha indicada en el objeto.
getYear()	DEPRECADA. Usar getFullYear() en su lugar.
Date.now()	Retorna el número de milisegundos desde el 1 de Enero de 1970 hasta ahora.
toString()	Convierte una fecha en una cadena.

Ejemplo:

```
let d = new Date();
alert(d.getTime());
alert(d.toString()); //Thu Oct 18 2018 12:10:37 GMT+0200 (Hora de verano romance)
```

Otras funciones:

eval(string): Esta función recibe una cadena de caracteres y la ejecuta como si fuera una sentencia de Javascript.

```
let miTexto1 = "hola ";
let miTexto2 = "cara ";
let miTexto3 = "bola ";
for (let i=1;i<=3;i++){
    eval("document.write(miTexto"+ i +"")");
}
```

Funciones anónimas

A veces nos interesa crear una función anónima, que no es más que una función, pero que no tiene nombre.

```
let mifuncion = function (a,b){
    return a+b;
};
```

Este tipo de función la usaremos en los siguientes temas.

Funciones 'flecha'

A la hora de definir una función **anónima**, tenemos una opción que es más 'compacta'. Es lo que llamamos funciones flecha.

Por ejemplo, la siguiente función:

```
// Función anónima tradicional
function (a){
    return a + 100;
}
```

Se puede definir también como:

```
// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y la llave de apertura.
(a) => {
    return a + 100;
}
```

Y, además:

```
// 2. Quita las llaves del cuerpo y la palabra "return" — el return está implícito.
(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos
a => a + 100;
```

En el caso de tener más de un argumento, no se podrían quitar los paréntesis:

```
// Función anónima tradicional
function (a, b){
    return a + b + 100;
}
```

```
// Función flecha
(a, b) => a + b + 100;
```

El equivalente a una función normal (que no es anónima);

```
// función tradicional
function suma ( a, b ) {
```

```
    return a + b;  
}
```

```
// su equivalente en función flecha  
let suma = ( a, b ) => a + b;
```

OJO, hay algunas funcionalidades que **NO** admiten las funciones flechas, comparándolas con las funciones normales. Por ejemplo, no tiene sus propios enlaces a **'this'** (ya veremos qué es esto cuando lleguemos a los eventos).

Actividades - Funciones:

- 1.- Escribe un script con, al menos, estos valores: 3, 3.5, "3", "3.5", "7aaa", "aaa7" y "hola". Recorre el array y muestra por consola el resultado de aplicar las funciones parseInt, parseFloat, isNaN, Number y String, a cada uno de los elementos del array. ¿Hay algún resultado que no esperabas?
- 2.- Crea una función llamada "DibujarLinea3", que dibuje una línea con 3 asteriscos en la pantalla.
- 3.- Crea una función llamada "DibujarCuadrado3x3", que dibuje un cuadrado (en la pantalla) formado por 3 filas con 3 asteriscos cada una (esta función deberá llamar a la función creada en el ejercicio anterior).
- 4.- Crea una función llamada "DibujarLinea", que dibuje una línea con X asteriscos (en la pantalla). La función tendrá un parámetro de entrada con el número de asteriscos a mostrar.
- 5.- Crea una función llamada "DibujarCuadrado", que dibuje un cuadrado formado por X filas con X asteriscos cada una (esta función deberá llamar a la función creada en el ejercicio anterior). El script pedirá al usuario el tamaño del cuadrado, y llamará a la función con ese dato.
- 6.- Crea una función "DibujarRectangulo" que dibuje en pantalla un rectángulo del ancho y alto que se indiquen como parámetros. El script pedirá al usuario el tamaño del rectángulo, y llamará a la función con ese dato.
- 7.- Crea una función "Cubo" que devuelva el cubo de un número real que se indique como parámetro. La función tendrá que comprobar que el parámetro es un número (o que se pueda convertir a un número), en caso contrario, devolverá false.
Prueba esta función para calcular el cubo de 3.2 y el de 5, y también prueba a pasarle "hola" como parámetro. ¿Qué ocurre?
- 8.- Crea una función "Menor" que devuelva el menor de dos números que recibirá como parámetros.
- 9.- Crear 4 funciones: PideNumero, EsPositivo, CalculaMitad y HazTodo

La función PideNumero, pedirá al usuario que introduzca un número y devuelve el número introducido. Las funciones EsPositivo y CalculaMitad, no imprimen nada en la consola, simplemente devuelven los valores correspondientes.

La función HazTodo no tendrá ningún parámetro, llamará a las otras tres funciones y mostrará la siguiente información en la consola:

```
'El número X es POSITIVO/NEGATIVO'  
'La mitad de X es Y'
```

Fuera de las funciones, solo habrá una llamada a la función HazTodo.

10.- ~~¿Para que se usan las funciones nativas “escape” y “unescape”? Haz un ejemplo donde las uses.~~

11.- Escribe una función que dado un string, devuelva un array con las palabras que forman el string. ¿Qué pasa si el string tiene 3 espacios en blanco seguidos? ¿Se puede solucionar?

12.- Escribe una función que se le pase un nombre completo (string) y devuelve el nombre abreviado (es decir, el nombre y la primera letra del primer apellido, seguido de un punto). Escribe también el código necesario para probarla.

13.- Crea una función que oculte parte de una dirección de email. Por ejemplo, si se le pasa “1234567890@gmail.com”, devolverá “12345...@gmail.com”, es decir, sustituye la 2ª mitad del email (antes de la @) por “...”. Escribe también el código necesario para probarla.

14.- Define una función que sustituya los espacios en blanco por un guión y, además, convierta todo el texto en minúsculas. Escribe también el código necesario para probarla.

15.- Escribe una función que reciba dos parámetros, string1 y string2, y que devuelva true si string1 contiene a string2, o false en caso contrario.

16.- Crea una función que genere un número aleatorio, entre dos valores dados. Utiliza esta función para simular el funcionamiento de un dado.

17.- Define una función que devuelva una cadena de texto con el día actual. Además, hay que pasarle el separador. Suponiendo que el separador es “-”, el formato de salida será “dd-mm-yyyy”

18.- En una empresa las facturas vencen a los 20 días. Crear una función que tendrá como parámetro una fecha con el formato 'dd-mm-YYYY' y devolverá la fecha de vencimiento (con el mismo formato). Hay que tener en cuenta que, si la fecha de vencimiento cae en fin de semana, habrá que mostrar la fecha del viernes anterior.

19. Hacer una función que reciba dos parámetros (mes y año). Con esos datos tiene que mostrar la siguiente información (por ejemplo, si le pasamos el mes de Octubre y el año 2020:

Octubre (2020)

Semana 1: 1(Ju) 2(Vi) 3(Sa) 4(**Do**)
Semana 2: 5(Lu) 6(Ma) 7(Mi) 8(Ju) 9(Vi) 10(Sa) 11(**Do**)
Semana 3: 12(Lu) 13(Ma) 14(Mi) 15(Ju) 16(Vi) 17(Sa) 18(**Do**)
Semana 4: 19(Lu) 20(Ma) 21(Mi) 22(Ju) 23(Vi) 24(Sa) 25(**Do**)
Semana 5: 26(Lu) 27(Ma) 28(Mi) 29(Ju) 30(Vi) 31(Sa)

20.- **[OPCIONAL]** Crear un calendario del estilo al que aparece en la siguiente imagen:

2020**Ene**

		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Feb

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	

Mar

						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Abril

		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

Mayo

				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Jun

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Jul

		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Ago

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Sep

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Oct

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Nov

						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Dic

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Con solo cambiar una variable (por ejemplo “/et anio”), mostrará el calendario del año indicado. El código deberá tener al menos la función ‘muestraMes’.

21.- [OPCIONAL] Calendario V2.0 (para el que tenga tiempo):

2020**Ene**

30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Feb

27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	1
2	3	4	5	6	7	8

Mar

24	25	26	27	28	29	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Abril

30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

Mayo

27	28	29	30	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Jun

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5
6	7	8	9	10	11	12

Jul

29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Ago

27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

Sep

31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11

Oct

28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

Nov

26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

Dic

30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

22.- **[OPCIONAL]** Continuación del ejercicio 5. Crear una función DibujarCuadradoHueco, que dibuje un cuadrado, pero mostrando solo el borde. Por ejemplo, si indicamos un tamaño de 5, saldría esto:

```
*****  
*      *  
*      *  
*      *  
*      *  
*****
```

3.16. Ámbito de las variables

El ámbito de una variable (llamado "scope" en inglés) es la **zona del programa en la que se define** la variable. JavaScript define dos ámbitos para las variables: global y local.

El siguiente ejemplo ilustra el comportamiento de los ámbitos:

```
function creaMensaje() {  
  var mensaje = "Mensaje de prueba";  
}  
creaMensaje();  
alert(mensaje);
```

Las variables definidas fuera de cualquier función, automáticamente se transforman en **variables globales** y están disponibles en cualquier punto del programa (incluso dentro de cualquier función).

Si una variable se declara fuera de cualquier función, automáticamente se transforma en variable global independientemente de si se define utilizando la palabra reservada var/let o no.

Sin embargo, las variables definidas dentro de una función pueden ser globales o locales. Si en el interior de una función, las variables se declaran mediante let se consideran locales y las variables se han declarado mediante var, se transforman automáticamente en variables globales.

¿Qué se mostraría en los siguientes ejemplos:

```
function creaMensaje() {  
  let mensaje = "Mensaje de prueba";  
  alert(mensaje);  
}  
creaMensaje();
```

```
let mensaje = "Mensaje1";  
function creaMensaje() {  
  let mensaje = "Mensaje de prueba";  
  alert(mensaje);  
}  
creaMensaje();  
alert(mensaje);
```

```
let mensaje = "Mensaje de prueba";  
function muestraMensaje() {  
  alert(mensaje);  
}  
muestraMensaje();
```

```
let mensaje = "Mensaje1";
function creaMensaje() {
    mensaje = "Mensaje de prueba";
}
alert(mensaje);
creaMensaje();
alert(mensaje);
```

```
function creaMensaje() {
    mensaje = "Mensaje de prueba";
}
creaMensaje();
alert(mensaje);
```

```
function DibujaLinea3(){
    for(i=1;i<=3;i++){
        document.write('*');
    }
    document.write('<br/>');
}
```

```
function DibujaCuadrado3(){
    for(i=1;i<=3;i++){
        DibujaLinea3();
    }
}
DibujaCuadrado3()
```

```
var i = 5;
for (var i=1;i<=10;i++){
    document.write(i);
}
alert(i);
```

```
let i = 5;
for (let i=1;i<=10;i++){
    document.write(i);
}
alert(i);
```

La recomendación general es definir como variables locales todas las variables que sean de uso exclusivo para realizar las tareas encargadas a cada función. Las variables globales se utilizan para compartir variables entre funciones de forma sencilla.

A partir de la versión **ES6** de JS, podemos usar **let** para declarar una variable. [Aquí](#) podemos ver la diferencia con **var**.

¿Cómo funcionarían los códigos anteriores usando **'let'** en vez de **'var'**?

3.17. Operador typeof

El operador **typeof** devuelve una cadena que indica el **tipo** del operando sin evaluarlo. La sintaxis es:

```
typeof (operando )
```

Los paréntesis son opcionales.

Suponga que define las siguientes variables:

```
let miFuncion = new Function("5+2")
let forma="redonda"
let tamano=1
let hoy=new Date()
```

El operador typeof devuelve los siguientes resultados para estas variables

```
typeof miFuncion === 'function'
typeof forma === 'string'
typeof tamano === 'number'
typeof hoy === 'object'
typeof noExiste === 'undefined'
```

3.18. Operador instanceof

Es importante no confundir typeof con instanceof, ya que **instanceof** valida sobre la cadena de prototipos, de un constructor. La sintaxis es:

```
objeto instanceof constructor
```

Donde:

objeto: Objeto a verificar.

constructor: función contra la que se hará la verificación.

En el siguiente ejemplo podemos ver claramente su funcionamiento:

```
color1=new String("verde")
color1 instanceof String // devuelve verdadero (true)

color2="coral"
color2 instanceof String // devuelve falso (color2 no es un objeto String)
```

3.19. Objetos

En Javascript, casi todo es un objeto, por ejemplo, si hacemos console.log(typeof []), no nos devuelve que el tipo es Array sino que es de tipo Objeto.

Lo que no es un objeto en Javascript, es un tipo primitivo:

```
console.log( typeof "hello world" )
```

```
console. log ( typeof 100 )  
console. log( typeof false )
```

Pero como en Javascript es muy importante el concepto de objeto, incluso los tipos primitivos pueden convertirse en objetos.:

```
console.log ( typeof new Number (100) )  
console. log ( typeof new String("hello world") )  
console. log (typeof new Boolean (true) )
```

Pero... ¿para qué vale entonces que Javascript tenga el objeto String si yo uso comillas dobles y funciona igual?. Pues porque los objetos tienen una cosa que se llama métodos, que son funciones para poder interactuar con los datos, por ejemplo, al hacer “hola”. toUpperCase lo que en realidad está sucediendo es que “hola” lo mete en un objeto String para poder acceder al método toUpperCase.

3.19.1. POO

Vamos a ver primero las características de la POO.

Clase: Define las características del Objeto.

Objeto: Una instancia de una Clase.

Propiedad (o atributo): Una característica del Objeto, como el color. Lo que viene a ser una variable.

Método: Una capacidad del Objeto, como caminar. Lo que viene a ser una función.

Constructor: Es un método llamado en el momento de la creación de instancias.

Herencia: Una Clase puede heredar características de otra Clase.

Encapsulamiento: Ocultamiento de los datos de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto.

Abstracción: Una clase abstracta es aquella sobre la que no podemos instanciar objetos. Por ejemplo, dada la clase Animal y la clase Perro (hija de Animal), no instanciamos animales como tal en el mundo, sino que instanciamos especímenes de un tipo de animal concreto.

Polimorfismo: Diferentes Clases podrían definir el mismo método o propiedad (polimorfismo de sobrecarga, paramétrico y de inclusión)

3.19.2. POO en JS

En JavaScript, la orientación a objetos es conceptualmente muy diferente a la orientación a objetos de Java y otros lenguajes. Es una programación orientada a objetos, pero, **basada en prototipos**.

La programación basada en prototipos es un estilo de programación orientada a objetos en la que las clases no están presentes y la reutilización de comportamiento (conocido como herencia en lenguajes basados en clases) se lleva a cabo a través de un proceso de decoración de objetos existentes que sirven de prototipos.

Aunque prototipo y objeto, no son lo mismo, a efectos prácticos vamos a considerar que si son lo mismo.

En JavaScript no existen las clases. En JavaScript sólo existen objetos. Pero estos objetos son muy flexibles y se les puede añadir atributos y métodos en el momento de la creación, justo después de haberlos creado o en cualquier momento a lo largo de la ejecución del programa. Esta es la clave de la programación orientada a objetos en JavaScript. En JavaScript los objetos son **dinámicos**, es decir,

podemos crear **instancias** de un prototipo (equivalente a crear objetos de una clase), y cualquier cambio en ese prototipo, se aplicará a todas las instancias donde esté **delegado**.

Es muy habitual encontrarnos (o crear) una **cadena de prototipos**. No es más que una **composición** de prototipos (u objetos) con otros prototipos (u objetos). Es decir, un objeto es hijo de otro, que a su vez puede ser hijo de otro, y así sucesivamente.

El prototipo básico que tienen todos los objetos es **Object.prototype**. Si creamos un objeto vacío, y lo depuramos en el navegador, veremos que tiene este prototipo.

```
let objeto_de_prueba = {}; // Esto es una forma de crear un objeto en JS
```

Además, cualquier objeto, tiene la propiedad **'__proto__'**, que te muestra (o te permite acceder) al prototipo delegado que tiene ese objeto.

De todas formas, a partir de la versión **ES6**, ya podemos trabajar como en Java (usando la palabra reservada **class**)

Podemos utilizar un objeto si necesitamos devolver en una función una serie de valores, por ejemplo:

```
const usuario = {  
  nombre: "John",  
  apellido: "Doe",  
  age: 30,  
  hobbies: ["leer", "programar", "quedarme el Viernes a última"],  
  direccion: {  
    calle: "c/ lacalle 5",  
    ciudad: "Badajoz"  
  }  
}
```

3.19.3. Objetos nativos

JS tiene muchos objetos nativos, en el punto anterior ya hemos visto muchos de ellos: Math, Object, Array y String.

```
alert (Math.random ());
```

El ejemplo anterior muestra cómo utilizar el **objeto** Math para obtener un número al azar mediante el uso de su **método** random().

3.19.4. Objetos personalizados

JS también nos permite crear nuestros propios objetos.

Clase

Como ya hemos comentado, JavaScript es un lenguaje basado en prototipos, pero a partir de la versión ES6 ya tenemos una declaración de clase, al igual que en C++, Java, php, ...

En el ejemplo siguiente se define una nueva clase llamada **Persona**.

```
class Persona {  
}
```

Objeto

Para crear un objeto, utilizamos la declaración “**new obj**”.

En el siguiente ejemplo se define una clase llamada Persona y creamos dos instancias (persona1 y persona2).

```
class Persona {  
  
let persona1 = new Persona();  
let persona2 = new Persona();
```

También hay otras formas de crear objetos que las veremos más abajo.

El constructor

El método **constructor** es un método especial para crear e inicializar un objeto creado con una clase.

```
class Persona {  
  constructor() {  
    alert('Una instancia de Persona');  
  }  
}  
  
let persona1 = new Persona();  
let persona2 = new Persona();
```

Atributos (o propiedades)

Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada “**this**”, que se refiere al objeto actual.

Lo que sí es diferente con respecto a otros lenguajes es que, una vez creado un objeto, a este le podemos añadir más propiedades de forma dinámica (mira el ejemplo siguiente).

```
class Persona {  
  constructor(primerNombre) {  
    this.primerNombre = primerNombre;  
  }  
}  
  
let persona1 = new Persona("Alicia");  
let persona2 = new Persona("Sebastian");  
  
// Creamos un propiedad dinámicamente  
persona2.otraPropiedad = 'Otra propiedad';
```

```
// Muestra el primer nombre de persona1
alert ('persona1 es ' + persona1.primerNombre); // muestra "persona1 es Alicia"
alert ('persona2 es ' + persona2.primerNombre); // muestra "persona2 es Sebastian"

// Muestra la propiedad creada dinámicamente
alert ('persona1 NO tiene otra propiedad: ' + persona1.otraPropiedad); // muestra "undefined"
alert ('persona2 tiene otra propiedad: ' + persona2.otraPropiedad); // muestra "Otra propiedad"
```

Aquí hemos usado la palabra reservada `this`, para indicar que hacemos referencia al mismo objeto y por tanto, podemos hacer referencia a sus métodos y propiedades de la forma: `this.edad` o `this.lafuncion()`

Al poder crear propiedades dinámicamente, Javascript tiene un método llamado “keys” que devuelve en un array todas las propiedades del objeto:

```
const usuario = {
  nombre: "John",
  apellido: "Doe",
  edad: 30,
  showName ( )
}
```

```
console.log (Object.keys (usuario)) //Devuelve un array: ["nombre", "apellido", "edad", "showName"]
```

Para mostrar los valores en vez del nombre de las propiedades, usar `Objet.values(usuario)` y devolvería: `["John", "Doe", 30, [Function: showName]]`

Métodos

Los métodos siguen la misma lógica que el método constructor.

```
class Persona {
  constructor(primerNombre) {
    this.primerNombre = primerNombre;
  }

  diHola () {
    alert ('Hola, Soy ' + this.primerNombre);
  }
}

let persona1 = new Persona("Alicia");
let persona2 = new Persona("Sebastian");

// Llamadas al método diHola de la clase Persona.
persona1.diHola(); // muestra "Hola, Soy Alicia"
persona2.diHola(); // muestra "Hola, Soy Sebastian"
```

También le podemos pasar parámetros a los métodos:

```
class Persona {
  constructor(primerNombreParametro) {
    this.primerNombre = primerNombreParametro;
  }
}
```

```

        diHola (saludo) {
            alert (saludo + ', soy ' + this.primerNombre);
        }
    }
}

```

```

let persona1 = new Persona("Alicia");
let persona2 = new Persona("Sebastian");

// Llamadas al método diHola de la clase Persona.
persona1.diHola('Hello'); // muestra "Hello, soy Alicia"
persona2.diHola('Bienvenido'); // muestra "Bienvenido, soy Sebastian"

```

Otra de las cosas que puede hacer Javascript con los objetos es añadir funciones a objetos ya creados, por ejemplo, si tenemos en cuenta la clase Persona anterior, podemos añadirle una función dinámicamente que retorne “soy una persona”:

```

persona1.saludo = function() {
    return "Soy una persona llamada " + this.primerNombre
}

```

Otras formas de crear objetos en JS

También podemos crear un objeto de otras maneras:

```

let persona1 = {
    primerNombre: 'Paco',
    diHola: function(saludo){
        alert (saludo + ', soy ' + this.primerNombre);
    }
}
persona1.diHola('Hola');

let persona2 = new Object({
    primerNombre: 'Juan',
    diHola: function(saludo){
        alert (saludo + ', soy ' + this.primerNombre);
    }
});
persona2.diHola('Bienvenidos');

```

En las últimas versiones de Javascript, el método dentro de una clase se puede acortar de la siguiente forma:

```

diHola: function(saludo){
    alert (saludo + ', soy ' + this.primerNombre);
}

```

Se quedaría así:

```

diHola(saludo){
    alert (saludo + ', soy ' + this.primerNombre);
}

```

A la hora de definir una función, ésta no tiene por qué estar implementada dentro del objeto, podemos ponerla fuera y hacer referencia a ella:

```
function showFullName ( ) { return "John Doe" }
const usuario = {
  nombre: "John",
  apellido: "ray",
  edad: 30,
  showFullName: showFullName
}
```

Para llamar a la función de la clase se usaría:

```
usuario.showFullName () //Retorna una cadena, habría que asignarla a algún sitio para usarla
```

Además, se suele emplear 'const' para definir un objeto. La declaración de una constante crea una referencia de sólo lectura. No significa que el valor que tiene sea inmutable, sino que el identificador de variable no puede ser reasignado, por lo tanto, en el caso de que la asignación a la constante sea un objeto, el objeto sí que puede ser alterado.

```
// const tambien funciona en objetos
const MY_OBJECT = {'key': 'valor'};
```

```
// Intentando sobrecribir el objeto nos lanza un error.
// No podemos darle otro valor a la variable MY_OBJET
MY_OBJECT = { };
```

```
// Pero si podemos modificar el objeto en sí, la siguiente sentencia se ejecutará sin problema
MY_OBJECT.key = 'otro valor';
```

En el caso de que queramos hacer un objeto inmutable, es decir, que no se pueda modificar, tendremos que usar `Object.freeze()`:

```
Object.freeze(MY_OBJECT );
MY_OBJECT.key = 'cualquier otro valor'; // No da error, pero no afecta en nada al objeto
```

Propiedades y métodos utilizando 'prototype'

La propiedad de prototipo de JavaScript también le permite agregar nuevos métodos y atributos a los **constructores** de objetos.

```
class Persona {
  constructor(primerNombreParametro) {
    this.primerNombre = primerNombreParametro;
    alert('Una instancia de Persona');
  }
}
Persona.prototype.edad = 0; // añadimos una nueva propiedad al constructor.
Persona.prototype.diHola = function() { alert ('Hola, Soy ' + this.primerNombre); }
```

Pero también podemos añadir más métodos o propiedades a los objetos nativos. Por ejemplo, IE8 no soporta el método 'trim' del objeto String, pero se lo podemos añadir:

```

<!DOCTYPE html>
<html>
<body>

<h2>JavaScript String.trim()</h2>

<p>IE 8 does not support String.trim(). To trim a string you can use a polyfill.</p>

<script>
if (!String.prototype.trim) {
  String.prototype.trim = function () {
    return this.replace(/^\s\uFEFF\xA0+|[\s\uFEFF\xA0]+$/g, "");
  };
};
let str = "      Hello World!      ";
alert(str.trim());
</script>

</body>
</html>

```

Herencia

La herencia es una manera de crear un objeto como una “versión especializada” de uno o más objetos. Desde ES6 usamos la palabra reservada **'extends'** para crear herencias. Además, hay que ejecutar la función **'super'** en el constructor de la clase hija.

```

// Definimos el objeto
class Persona {
  constructor(primerNombreParametro) {
    this.prop1 = "uno";
    this.prop2 = "dos";
  }

  milog () {
    console.log(this.prop1 + this.prop2);
  }
  milog2 () {
    console.log("Estamos en la función milog2 del padre: " + this.prop1 + this.prop2);
  }
}

// Definimos otro objeto, que será hija de la anterior.
class Estudiante extends Persona {
  constructor() {
    super();
  }

  // Podemos sobrescribir los métodos del padre.
  milog2 () {

```



```

        super.milog2(); // Aquí estamos llamando al método milog2 del padre.
        console.log("Estamos en la función milog2 del hijo: " + this.prop1 + this.prop2);
    }
}

// creamos el padre (instanciamos la clase)
let persona1 = new Persona;
persona1.milog();
persona1.milog2();

// creamos el objeto (hijo)
let estudiante1 = new Estudiante;
estudiante1.prop1 = "111111";
estudiante1.milog(); // Aquí llamamos al método del padre.
estudiante1.milog2(); // Aquí llamamos al método del hijo.

```

En el caso de que el constructor del padre, tuviera algún parámetro, sería así:

```

class Persona {
    constructor(nombre) {
        this.name = nombre;
    }
}

class Estudiante extends Persona {
    constructor(nombre, notas) {
        super(nombre);
        this.notas = notas;
    }
}

```

Veamos el ejemplo siguiente:

```

class Persona {
    constructor(nombre) {
        this.name = nombre;
    }
    diHola(){
        alert ("Hola, soy " + this.name);
    }
}

Persona.prototype.diHola2 = function() {
    alert ("Hola2, soy " + this.name);
}

class Estudiante extends Persona {
    constructor(nombre) {
        super(nombre);
    }
}

```

```

    }
  }

  let persona1 = new Persona("Paco");
  persona1.diHola();
  persona1.diHola2();

  let estudiante1 = new Estudiante("Francisco");
  console.log(estudiante1.name);
  estudiante1.diHola();
  estudiante1.diHola2();

```

En el ejemplo anterior, Estudiante no tiene que saber cómo se aplica el método *diHola()* de la clase Persona, pero, sin embargo, puede utilizar ese método. La clase Estudiante no tiene que definir explícitamente ese método, **a menos que queramos cambiarlo**. Lo mismo ocurre con el método *diHola2()*, que se ha añadido dinámicamente (con prototype) a la clase Persona.

API fluida en JS

JS permite la implementación de la API fluida, que es una implementación de una API orientada a objetos que tiene como objetivo proporcionar un código más legible, permitiendo **encadenar** las llamadas de métodos. Por ejemplo, suponiendo que la clase Persona tenga definido los métodos *diHola* y *diAdios*, al usar la API fluida podríamos usarlo así:

```

let persona1 = new Persona();
persona1.diHola().diAdios();

```

Para conseguir esto, los métodos tienen que devolver el propio objeto, lo cual se consigue con: **"return this"**

Por ejemplo:

```

class Persona {
  constructor(nombre) {
    this.name = nombre;
  }
  diHola(){
    alert ("Hola, soy " + this.name);
    return this;
  }
  diAdios(){
    alert ("Adios");
  }
}

let persona1 = new Persona("Paco");
persona1.diHola().diAdios();

```

En este ejemplo, como el método *diAdios*, NO devuelve el objeto, la siguiente línea daría error:

```
persona1.diAdios().diHola(); //ESTO DARÍA ERROR
```

Un buen ejemplo de esto es el framework jQuery.

Herencia por Composición

Ya hemos visto que la herencia es una manera de crear un objeto como una “versión especializada” de uno o más objetos.

En Javascript, otra forma de usar la herencia es la composición.

Hay varias formas de usar la composición y una de ellas es la **delegación**.

En este caso vamos a usar **Object.create**, al que le tenemos que pasar el prototipo en el que delega el objeto a crear.

Por ejemplo:

```
let a = { }
```

Es equivalente a:

```
let a = Object.create(Object.prototype);
```

También podemos crear un objeto sin “Object.prototype”:

```
let a = Object.create(null)
```

Para crear un objeto que delegue en otro (distinto de Object.prototype), es tan sencillo como:

```
let a = { }
```

```
let b = Object.create(a);
```

Con esto conseguimos que el objeto **b** delegue en el objeto **a**, o dicho de otro modo, que el objeto **b** herede de **a**.

Sabiendo esto podemos hacer esto:

```
// Definimos el objeto
```

```
let Persona = {
```

```
  prop1: "uno",
```

```
  prop2: "dos",
```

```
  milog: function(){
```

```
    console.log(this.prop1 + this.prop2);
```

```
  },
```

```
  milog2: function(){
```

```
    console.log("Estamos en la función milog2 del padre: " + this.prop1 + this.prop2);
```

```
  }
```

```
}
```

```
// Definimos otro objeto, que será hija de la anterior.
```

```
let Estudiante = Object.create(Persona);
```

```
Estudiante.milog2 = function(){
```

```
  console.log("Estamos en la función milog2 del hijo: " + this.prop1 + this.prop2);
```

```
}
```

```
// creamos el padre (instanciamos la clase)
```

```

let persona1 = Persona;
persona1.milog();
persona1.milog2();

// creamos el objeto (hijo)
let estudiante1 = Estudiante;
estudiante1.prop1 = "11111";
estudiante1.milog(); // Aquí llamamos al método del padre.
estudiante1.milog2(); // Aquí llamamos al método del hijo.
estudiante1.__proto__.milog2(); // Aquí llamamos al método del padre.

```

Concatenación

Es otro tipo de composición. En el apartado anterior vimos la delegación, y ahora la concatenación. Para esto usamos **Object.assign**, este método copia todas las propiedades enumerables de uno o más objetos fuente a un objeto destino, y además devuelve el objeto destino.

En nuestro ejemplo, el primer parámetro usamos un objeto vacío, para que a dicho objeto le concatene los demás objetos pasados como parámetros.

```

const datosPersonales = { nombre: 'Paco', edad: 2 };
const notas = { ssii: 10, diw: 10, dwec: 10 };

const alumno1 = Object.assign({}, datosPersonales, notas);
const alumno2 = Object.assign({}, datosPersonales, notas);

alumno1.nombre = 'Paula';
alumno1.ssii = 8;
alumno2.nombre = 'Juan';
alumno1.ssii = 7.5;

```

RECORDATORIO

En Javascript **no existe** el concepto de *clases*. Aún en ES6 y la llegada de la palabra reservada **class** sigue sin existir como tal. JavaScript es un lenguaje **prototipado**. Aunque uses la nueva sintaxis para crear "*clases*", estarás creando **funciones** con un prototipo.

3.19.5. Espacio de nombres

Un **espacio de nombres** (namespace) es un contenedor que permite asociar toda la funcionalidad de un determinado objeto con un nombre único. En JavaScript un espacio de nombres es un objeto que permite a métodos, propiedades y objetos asociarse. La idea de crear espacios de nombres en JavaScript es simple: Crear un único **objeto global** para las variables, métodos, funciones convirtiéndolos en propiedades de ese objeto.

El uso de los namespace permite minimizar el conflicto de nombres con otros objetos haciéndolos únicos dentro de nuestra aplicación.

Actividades - Objetos:

1. Crea una clase llamada *Persona*. Esta clase deberá tener una propiedad "nombre", de tipo string. También tendrá un método "Saludar", que escribirá en la pantalla "Hola, soy " seguido de su nombre. Se crearán dos objetos de tipo *Persona*, se les asignará un nombre a cada uno y se les pedirá que saluden.
2. Para guardar información sobre libros, vamos a comenzar por crear una clase "*Libro*", que contendrá propiedades "autor", "título", "ubicación" (todos ellos strings) y métodos Get y Set adecuados para leer su valor y cambiarlo. Se creará un objeto de la clase *Libro*, le daremos valores a sus tres propiedades (usando los set) y luego los mostrará por pantalla (usando los get).
3. Crea una clase "*Vehículo*", con la propiedad "cilindrada" (número entero). Creamos los Get y Set para las propiedades. También tendrá un método **InfoCilindrada** para mostrar la cilindrada por pantalla.

Define otra clase "*Coche*" que será "hija" de la clase "*Vehículo*", con las propiedades "marca" (texto) y "modelo", y un método **InfoCompleta** (que mostrará por pantalla la marca, el modelo y la cilindrada).

El constructor de la clase "*Coche*", recibirá como parámetros la marca, el modelo y la cilindrada. Por último, crea un objeto *Coche* con los parámetros que prefieras y llama a los métodos "InfoCilindrada" e "InfoCompleta".

4. ¿Hay alguna forma de declarar las propiedades de las clases como "privadas"? Haz un ejemplo y compruébalo.

Bibliografía

- Google
- Wikipedia
- <https://devcode.la/blog/que-es-javascript/>
- <https://librosweb.es/libro/javascript/>
- <http://www.etnassoft.com/2011/01/27/tipado-blando-en-javascript/>
- https://www.w3schools.com/js/js_mistakes.asp
- https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model
- <http://lineadecodigo.com/javascript/crear-un-objeto-con-metodos-en-javascript/>
- [https://es.wikipedia.org/wiki/Convenci%C3%B3n_de_nombres_\(programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Convenci%C3%B3n_de_nombres_(programaci%C3%B3n))
-
-

Avanzado:

- <http://www.tufuncion.com/funciones-javascript>
- <https://medium.com/codeine-labs/funciones-nativas-de-javascript-1b0f19e8f7b>
- <https://desarrolloweb.com/articulos/herencia-clases-javascript-ecmascript.html>

- <https://code.tutsplus.com/es/tutorials/understanding-fluent-apis-in-javascript--cms-24429>
- <https://www.campusmvp.es/recursos/post/JavaScript-Variables-y-funciones-privadas.aspx>
-