La alegría de crecer juntos

Plaza de Castilla - Madrid
COLEGIO NUESTRA SEÑORA DEL PILAR
Salesianas

C.F.G.S.: DESARROLLO DE APLICACIONES WEB
Módulo: DESARROLLO WEB EN ENTORNO CLIENTE

02 JAVASCRIPT BÁSICO

Normas sintácticas:

- Sensible a mayúsculas/minúsculas
  - Sintaxis lowerCamelCase si están compuestas por varias palabras
- Comentarios
  - //
  - /* */
- Fin de la instrucción con ;
  Podemos omitir dicho signo si cada instrucción se encuentra en una línea independiente pero no es una buena práctica de programación.

- Espacios consecutivos se consideran un único espacio

Declaración de variables

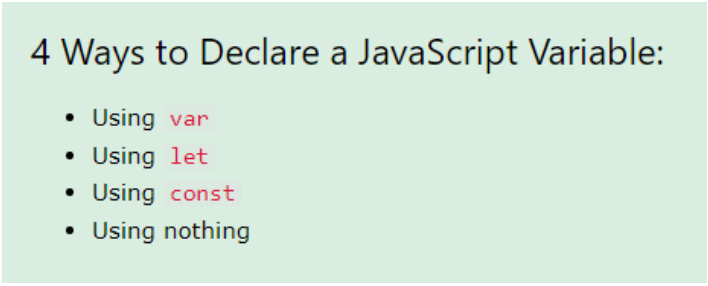
- Lenguaje de programación de "tipado débil", lo que quiere decir que el tipo de datos no es una propiedad de las variables, sino de los datos en sí, de modo que una misma variable puede contener datos de tipo diferente en distintos instantes.

```
a=25;
alert (typeof a ); // number
a= 'hola';
alert (typeof a ); // string
```

Ver ej 02 01

- Permite inicializar o dar valor a variables que en principio no han sido declaradas

```
a = 22;
otra = a + 8;
```



Fuente: https://www.w3schools.com/

- Permite declarar variables mediante **var,** que es la palabra reservada utilizada tradicionalmente para la definición de variables desde el inicio del lenguaje.

```
var a=7, b='pepe';
var una;
```

- No se puede utilizar una variable que no haya sido declarada o al menos inicializada.
- Cualquier variable usada sin haber sido declarada tiene alcance global (puede ser utilizadas en el interior de cualquier función o en cualquier otro código incluido en el mismo documento html)
- Una variable declarada con **var** dentro de una función tiene alcance local dentro de ella (más específicamente, pertenece al contexto de ejecución local de la función), es decir, será reconocida dentro de esa función y dentro de cualquier otra función definida dentro de ella, pero no fuera de la función.
- Una variable declarada con **var** fuera de una función tiene alcance global.
- Si existe una variable local en una función con el mismo nombre que una variable global, dentro de la función se asocia ese nombre a la variable local.

- A partir del JavaScript ECMAScript 6 aparecen dos nuevas palabras para la definición de variables: **let y const**, ambas tienen ámbito de bloque.

```
if (true){
        let a, b, c;
        a = 3;
        b = 8;
        c = a + b;
        const PI = 3.14;
}
```

- **const** se utiliza para declarar una constante que no podrá variar a lo largo del programa y solo puede utilizarse en el bloque de código en el que se ha declarado

- **let** permite definir una o varias variables que solo serán visibles y solo pueden utilizarse en el bloque en el que se han definido.

- El uso de var permite redefinir la misma variable en el mismo bloque, algo que no se puede hacer con let. Por ejemplo, el siguiente código funciona sin problemas:

```
var cont = 1;
var cont = 2;
```

Pero el siguiente daría error

```
let cont = 1;
let cont = 2;   ERROR!!
```

Sí es posible hacer lo siguiente:

```
let x = 1;

if (true) {
   let x = 2;
   console.log(x);
}
console.log(x);
```

que dará la salida 2

1

Ver ej 02 02

- Utilización en los bucles for

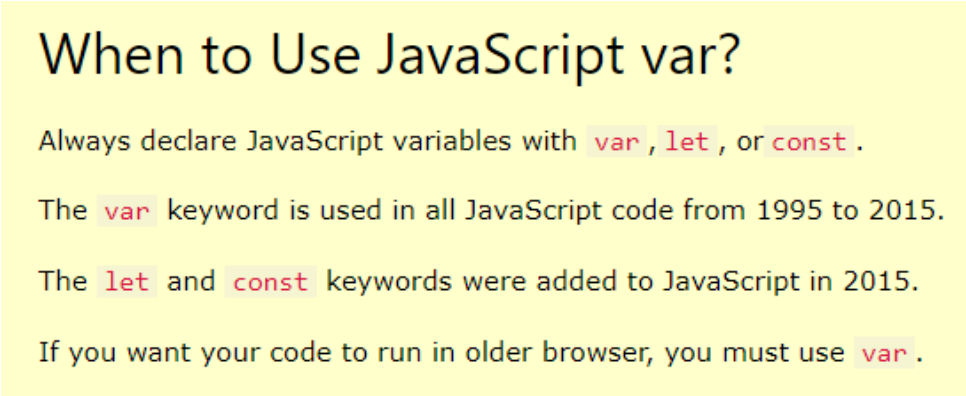
En los bucles for es habitual definir una variable sobre la que iterar. Esto se puede hacer tanto con let como con var. Aunque en el caso de let el alcance de las variables solamente es dentro del bucle. Lo que permite evitar la aparición de problemas.

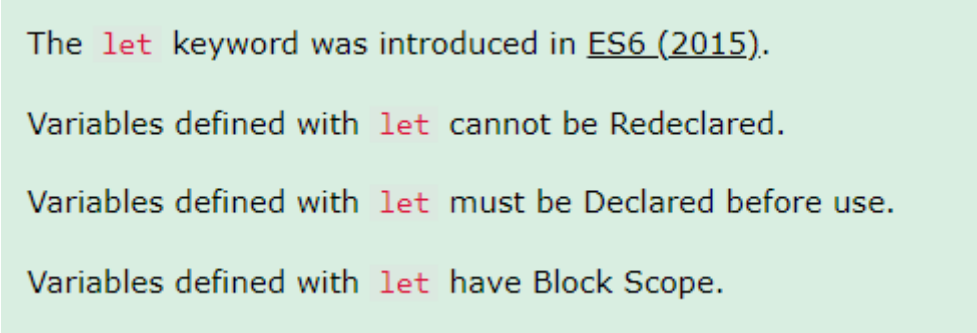
Un efecto secundario de esto es la posibilidad de anidar bucles dentro de otros con la misma variable para iterar con let.

```
for (let i=0; i < 2; ++i) {
        console.log('Primer', i);
        for (let i=0; i <2; ++i) {
            console.log('Segundo', i);
        }
}
```

que dará la salida

Primer 0
Segundo 0
Segundo 1
Primer 1
Segundo 0
Segundo 1





Fuente: https://www.w3schools.com/

- Identificadores de variables:

  Se pueden utilizar caracteres alfanuméricos y el carácter subrayado _

  No pueden comenzar por número.

  No pueden utilizarse palabras reservadas.

  Palabras reservadas: http://msdn.microsoft.com/es-es/library/0779sbks(v=vs.94).aspx

break	delete	if	this	while
case	do	in	throw	with
catch	else	instanceof	try	
continue	finally	new	typeof	
debugger	for	return	var	
default	function	switch	void	

Tipos de datos en Javascript

Los valores que se pueden almacenar en una variable se corresponden con los siguientes tipos:

- **number**

  Cualquier número entero o decimal (expresado en notación decimal, hexadecimal o científica) Correspondería al llamado double de Java o C

  let n1=4500;

  let n2=4500.00;

  let n3=0x1194; // notación hexadecimal

  let n4=4.5e3; // notación científica

- **string**

  Cualquier cadena de caracteres comprendida entre comillas dobles o simples

  let texto1 = "Hola caracola";

  let texto2 = 'Buenas…';

  let texto3 = 'Podemos incluir \\, \', \" en nuestras cadenas. \n...Pero con cuidado';

Ver ej 02 03

Secuencias de escape:

http:// http://msdn.microsoft.com/es-es/library/ie/2yfce773(v=vs.94).aspx

También podemos incluir cualquier carácter Unicode de 16 bits utilizando la fórmula \u*nnnn*, donde *nnnn* es el valor hexadecimal del carácter. En http://unicode.org/charts/ pueden consultarse todos los caracteres Unicode, como los símbolos matemáticos, los caracteres de otros idiomas o los signos Braille. En la siguiente tabla se recogen los códigos Unicode de algunos caracteres especiales del castellano, y en la dirección http://www.rishida.net/tools/conversion/ dispone de un completo conversor:

á	é	í	ó	ú	Á	É	Í
\u00e1	\u00e9	\u00ed	\u00f3	\u00fa	\u00c1	\u00c9	\u00cd
Ó	Ú	ü	Ü	ñ	Ñ	¡	¿
\u00d3	\u00da	\u00fc	\u00dc	\u00f1	\u00d1	\u00a1	\u00bf

var texto4 = 'Y utilizar \u00e1';

- **boolean**

  Admite los datos false y true

- **undefined**

  Sólo admite el dato undefined. Valor asignado implícitamente a todas las variables declaradas a las que no se les ha asignado dato aun.

- **null**

Este tipo de datos sólo admite el dato null.

Se utiliza para inicializar una variable que posteriormente vamos a referir a un objeto pero aún no tiene objeto asignado.

```
let automovil=null;
```

Si utilizamos typeof para ver el tipo de automóvil nos dirá que es un object.

También se utiliza null para preguntar si una variable no tiene valor.

```
let texto;
if (texto == null)
        alert ('texto no tiene valor asignado');
if (texto == undefined)
        alert ('texto no tiene valor asignado');
```

El valor **null** se comporta como el número 0, mientras que **undefined** se comporta como el valor especial **NaN** (no es un número).

Si comparas un valor **null** con un valor **undefined**, son iguales.

```
let a;
b=null;
alert (a); //undefined
alert (b); //
alert (a+2); // NaN
alert (b+2);  // b lo considera como 0 y da 2
```

- **object**

Este tipo de datos puede interpretarse como una colección de otros datos, sin ninguna restricción en cuanto a su tipo, en la que cada uno de ellos está identificado por un nombre.

Para crear un dato del tipo object podemos utilizar la expresión new Object()

```
let automovil = new Object();
```

Ver ej 02 04

JavaScript incluye algunos **objetos predefinidos**, denominados objetos del núcleo, como **Number, String, Boolean, Array, Function, Date y Math**, que veremos más adelante.

Cada vez que en una expresión interviene un dato de tipo number, string o boolean, en segundo plano se crea para él un objeto de tipo Number, String o Boolean, respectivamente, de modo que podríamos acceder a las propiedades y métodos de ese objeto. Por ejemplo, sabiendo que el objeto String posee una propiedad llamada length que contiene el número de caracteres de la cadena, el siguiente código podría servirnos para mostrar cuántos caracteres tiene la cadena.

```
let texto = “Hola”;
alert (texto.length);
```

OPERADORES

En esta sección se presentan tablas de los distintos tipos de operadores disponibles en JavaScript.

Operadores aritméticos

Operador	Nombre	Descripción
++	Incremento unario	Pude usarse en modo prefijo o sufijo. Por ejemplo, a = ++b; incrementa en una unidad el valor de b y se lo asigna a a; por el contrario a = b++; asigna a a el valor de b y después incrementa en una unidad el valor de b. Se denomina unario porque actúa sobre un único operador.
--	Decremento unitario	Pude usarse en modo prefijo o sufijo. Por ejemplo, a = --b; decrementa en una unidad el valor de b y se lo asigna a a; por el contrario a = b--; asigna a a el valor de b y después decrementa en una unidad el valor de b. Se denomina unario porque actúa sobre un único operador.
+	Suma	Por ejemplo a = 5 + 3.2; //a = 8.2
-	Resta	Por ejemplo a = 5 - 3.2; //a = 1.8
*	Producto	Por ejemplo, a = 5 * 3.2; // a = 16
/	Cociente	Por ejemplo, a = 5 / 3.2; //a = 1.5625
%	Resto o módulo	Por ejemplo, a = 5 % 3.2; //a = 1.7999999999999998

Operadores booleanos o lógicos

Operador	Nombre	Descripción
!	Negación	Por ejemplo, a = !false; // a = true
&&	Y	Por ejemplo, a = true && false; //a = false
||	O	Por ejemplo a = true || false; //a = true

Nota: En JavaScript los operadores booleanos && y || son de tipo cortocircuito y su resultado es el último operando evaluado. ¿Qué quiere decir esto? Que si el operando de la izquierda es suficiente para conocer el resultado de la operación, no se evalúa el de la derecha y se devuelve directamente el de la izquierda como resultado de la operación. Por el contrario, si el operando de la izquierda no es suficiente para conocer el resultado de la operación se devuelve el de la derecha. Por ejemplo, false && 'casa' devuelve false, pero true && 'casa' devuelve true. Otro ejemplo con ||: false || 'casa' devuelve 'casa', pero true || 'casa' devuelve true.

Operadores de comparación

Operador	Nombre	Descripción

==	Igual que	Por ejemplo, 5 == 3 + 2; //true
		Pero 0.3 == 0.1 + 0.2; //false
		Y false == "false"; //false
		Pero false == "0"; //true
		Y "2" == 2; //true

!=	Distinto que	Por ejemplo, 5 != 3 + 2// false
===	Idéntico a (incluso en tipo)	Por ejemplo, "2" === 2; //false
!==	Diferente a (en dato y/o tipo)	Por ejemplo, "2" !== 2;//true

<	Menor que (números), o código ASCII anterior a (cadenas)	Por ejemplo, 5 < 3; //false
		Y "casa" < "tos"; //true
		Pero "Tos" < "casa"; //true

<=	Menor o igual que, o código anterior o igual que	Por ejemplo, 5 <= 3; //false
>	Mayor que (números), o código ASCII posterior a (cadenas)	Por ejemplo, 5 > 3; //true
>=	Mayor o igual que, o código posterior o igual que	Por ejemplo, 5 >= 3; //true

Operadores de asignación

Operador	Nombre	Descripción
=	Asignación simple	Por ejemplo, nombre = 'pepe';
+=	Suma/concatenación y asignación	Por ejemplo, nombre += apellido;
-=	Resta y asignación	Por ejemplo, credito-=1;
*=	Producto y asignación	Por ejemplo, iva*=1.1;
/=	Cociente y asignación	Por ejemplo, salario /= 1.1;
%=	Resto y asignación	Por ejemplo, dni =% 23;

Otros operadores de interés

Operador	Nombre	Descripción
?:	Operador condicional	Si la expresión anterior al signo ? se evalúa como true el resultado es el dato anterior al signo :, y en caso contrario el dato posterior al signo :.

		Nota=a>=5?'aprobado':'suspenso'; Por ejemplo, temperatura>=37.2?'enfermo':'sano'
typeof	Tipo de	Calcula el tipo del dato. Por ejemplo, typeof 'casa' alert(typeof 'casa'); → string alert(typeof 23); → number alert(typeof null); → object
+	Concatenación de cadenas	Sirve para concatenar dos cadenas de caracteres.
( )	Agrupación de operaciones	Se utiliza para agrupar operaciones estableciendo en qué orden deben efectuarse.

Precedencia de los operadores

```
( )
++ -- !
* / %
+ -
< > <= >=
== !=
&&
||
= += -= *= /= %=
```

Conversión de tipos implícita

- Cuando en una operación con el operador + uno de los operandos es una cadena, el otro se convertirá también en una cadena.

alert (2+2); → 4
alert (2+'2'); → 22

Nota:

- *true se convierte en 1 antes de intervenir en una operación de comparación.*
  alert ((5==2+3)==2); →false
  alert ((5==2+3)==1); →true

- *false se convierte en 0 antes de intervenir en una operación de comparación*
  alert ((5==2+2)==0); → true

- *Al comparar una cadena con un número se convertirá la cadena en un número antes.*
  alert (23=="23"); → true

- *En una comparación, cualquier cadena no vacía que pueda leerse como un número expresado en decimal (incluida notación científica) o hexadecimal se convertirá en ese número*

Se recomienda:

Evitar mezclar tipos de datos en las operaciones cuando sea posible

Recurrir a alguno de los **métodos de conversión explícito o funciones de forzado de tipo (*casting*)** que se recogen en la siguiente tabla cuando no lo sea.

Función/Método	Descripción

Number(*dato*)	Función que devuelve el resultado de convertir *dato* en un número (que puede ser NaN Not a Number).
	true se convierte en 1, y false se convierte en 0.
	Cualquier cadena que pueda ser interpretada como un número ('12.45', '0x3f', '314e-2') se convierte en ese número.
	Una cadena vacía se convierte en 0.
	Cualquier otro dato se convierte en NaN.
	`a='23';`
	`alert (a+5); // 235`
	`a=Number (a);`
	`alert (a+5); // 28`
	`alert (Number("1.23")+4); // 5.23`
	`alert(Number('casa')); // NaN`
	`alert(Number('')); // 0`

isNaN(valor)	La función **isNaN**(valor) devuelve true cuando valor vale NaN y false en caso contrario

parseInt(*cadena*)	Función que analiza *cadena* carácter a carácter empezando por la izquierda intentando componer un número entero expresado en decimal o hexadecimal. En cuanto encuentra un carácter que no puede formar parte de un número entero detiene el análisis y devuelve el resultado del análisis hasta ese punto. Si el análisis se detiene en el primer carácter o *cadena* está vacía devuelve NaN (obsérvese que este comportamiento es diferente al del Number(), que devuelve 0).
	`a='23';`
	`alert (a+5); // 235`
	`a=parseInt (a);`
	`alert (a+5); // 28`
	`alert (parseInt("1.23")+4); // 5`
	`alert (parseInt("1.83")+4); // 5`
	`alert (parseInt("1pa83")+4); // 5`
	`alert(parseInt('casa')); // NaN`
	`alert(parseInt('')); // NaN`

parseFloat(*cadena*) Similar a la función anterior, pero intentando componer un número de coma flotante expresado en decimal o notación científica.

```
alert (parseFloat ("1.23")+4); // 5.23
alert (parseFloat ("1.83")+4); // 5.83
alert (parseFloat ("1,83")+4); // 5
alert (parseFloat ("1pa83")+4); // 5
```

Boolean(*dato*) Función que devuelve el resultado de convertir *dato* a un valor booleano (true o false).

Cualquier cadena no vacía y cualquier número distinto de 0 se convierte en true, incluso "false". Una cadena vacía y el número 0 se convierten en false.

String(*dato*) Función que devuelve el resultado de convertir *dato* en una cadena de caracteres.

```
alert (23.567+5); // 28.567
alert (String(23.567)+5);  // 23.5675
```

RESUMEN

Number(dato) Función que devuelve el resultado de convertir *dato* en un número (si no puede convertirlo devuelve NaN).

isNaN(**valor**) devuelve true cuando valor vale NaN y false en caso contrario

parseInt(*cadena*) Convierte cadena en entero**,** si no puede devuelve NaN

parseFloat(*cadena*) Convierte cadena en número decimal**,** si no puede devuelve NaN

BIFURCACIONES

- **If**

La sintaxis básica de la instrucción if es la siguiente:

if (*expresiónDeControl*){

instrucciones a ejecutar si la expresión de control es verdadera;

}else{

instrucciones a ejecutar si la expresión de control es falsa;

}

```
let a = 5, b= 3;
if (a<b) {
alert (b+' es mayor que '+a);}
else
        if (a==b) {
                alert ('son iguales');}
        else
                alert (b+' es menor que '+a);
```

- **switch case**

Las bifurcaciones de tipo switch…case comparan el valor de una expresión de control con diferentes "casos" posibles, y ejecutan el código asociado al caso correspondiente, o el bloque correspondiente al caso default (por defecto). Es muy importante recordar que cada bloque deberá terminar con una instrucción break si no queremos que se ejecuten automáticamente las instrucciones del siguiente caso (aunque su valor no coincida con el de la expresión de control), y así sucesivamente hasta que se encuentre un break o se alcance el final de la instrucción switch…case.

Su sintaxis es la siguiente:

switch (*expresiónDeControl*){

case *valor1*:
bloque de instrucciones a ejecutar si expresiónDeControl coincide con valor1;
break;
case *valor2*:
b*loque de instrucciones a ejecutar si expresiónDeControl coincide con valor2;*
break;
…
default:
bloque de instrucciones a ejecutar si expresiónDeControl no coincide con ningún caso;
}

```
switch (a){
        case 1:
                alert ('a vale 1');
                break;
        case 2:
                alert ('a vale 2');
                break;
        case 5:
                alert ('a vale 5');
                break;
        default:
                alert ('no vale nada');
}
```

Ver ej 02 05

BUCLES

- **while**

La sintaxis de while es la siguiente:

while (*expresiónDeControl*){

bloque de instrucciones a ejecutar en cada iteración;

}

```
let a=1;
while (a<=5){
        document.write ("<br> a vale "+a);
        a++;
}
```

- **do while**

La sintaxis de do … while es la siguiente:

do{

bloque de instrucciones a ejecutar en cada iteración;

}while (*expresiónDeControl*);

```
let a=1;
do{
        document.write ("<br> a vale "+a);
        a++;
}
while (a<5);
```

- **for**

La sintaxis de for es la siguiente:

for(*instrucciónDeInicialización*;*expresiónDeControl*;*instrucciónTrasCadaIteración*){

bloque de instrucciones a ejecutar en cada iteración;

}

```
for (a=1;a<=5;a++)
        document.write ("<br> a vale "+a);
```

break → Para forzar la salida de un bucle

continue → Para abandonar la iteración sin ejecutar las instrucciones restantes y evaluar de nuevo la expresión de control para decidir si debe realizar otra iteración o no

Ver ej 02 06

FUNCIONES

JavaScript nos ofrece la posibilidad de implementar funciones. Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.

Su sintaxis es la siguiente:

function *nombreFunción* (*argumento1*, …, *argumentoN*){

…bloque de instrucciones de la función, incluido el return opcional…

}

```
function cuadrado (x){
return x*x;
}
alert (cuadrado (3));
```

- **Argumentos**
  - El número de argumentos puede ser variable. Podemos definir la función con n y luego llamarla con m argumentos (n>=m ó n<=m)

```
function sumar2 (x,y){
return x+y;
}
alert (sumar2(2,3));  // 5
alert (sumar2(2,3,4,5));  // 5
alert (sumar2(2));   // NaN
```

  - Los argumentos de la llamada se asignan por orden a los de la declaración.
  - Al llamar a una función se crea una variable llamada **arguments** que es de tipo array, con alcance local dentro de la función y que contiene una lista de todos los argumentos enviados por la llamada. Esto será muy útil cuando a una función le lleguen un número variable de argumentos.

```
function suma(){
var resultado = 0;
for (i=0;i<arguments.length;i++)
        resultado+=arguments[i];
return resultado;
```

```
}
alert (suma (1,2,3,4,5));
```

Ver ej 02 07

- En JavaScript **la asignación de todos los datos se realiza por valor excepto la de los objetos, que se realiza por referencia**. Por ejemplo, en el siguiente código enviamos un array en la llamada a la función y modifica los números del array cambiándolos por sus cuadrados

```
let numeros = new Array (1,2,3,4,5);
cuadrados(numeros);
alert(numeros);
function cuadrados (num)
{ for (i=0;i<num.length;i++)
        num[i]=num[i]*num[i];
}
```

Ver ej 02 08

- Se pueden enviar una función como argumento a otra función
- Se pueden llamar funciones a sí mismas recursivamente

```
function factorial (n){
if (n==1)
        return 1;
else
        return n*factorial (n-1);
}
alert (factorial(4));
```

- Se pueden anidar definiciones de funciones. Una función defina dentro de otra función sólo podrá ser utilizada dentro de la función en la que ha sido definida.

```
function dividir (x)
{
function par (y){
if (y%2==0)
        return true;
else
        return false;
```

```
}
if (par(x))
        return x/2;
else
        return x;
}
// dividir divide un numero en dos si es par, si no devuelve el número
alert (dividir(5));
alert (par(5));  // error!
```

Ver ej 02 09

Entrada de datos para para ejercicios y/o pruebas

```
<script>
    let dato = prompt("Introduce un dato");
    alert (dato);
</script>
```