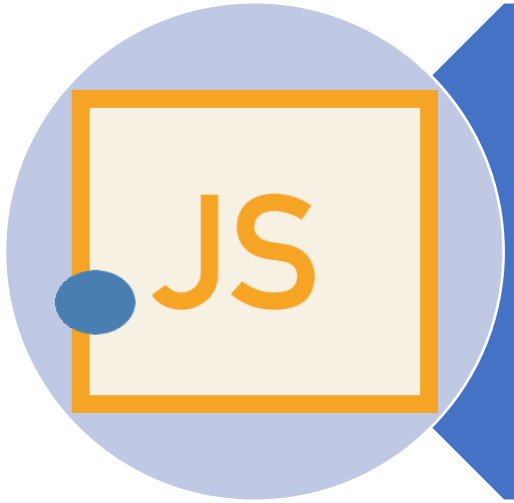
Desarrollo Web en Entorno Cliente

Jorge García Flores

Tema 3



Manejo de la sintaxis

sentencias

Las sentencias son cada una de las ordenes ejecutables de un programa. En JavaScript cada sentencia se debe delimitar separándolas en líneas o finalizándolas con ;

```
sentencia 1
sentencia 2
sentencia 3;
sentencia 4; sentencia 5;
```

En ocasiones es necesario o conveniente agrupar sentencias en bloques, incluyéndolas entre llaves { }

```
{
 sentencia 1;
 sentencia 2;
 …
}
```

Normalmente el código JS se comprime (*minify*) (y se ofusca) con el fin de reducir el peso del script y así conseguir acelerar la carga del sitio web, lo que es muy bueno para el SEO y Google.

Cuando se comprime JS el código suele aparecer una sola línea muy extensa, donde cada sentencia está separada con ;

https://herramientas-online.com/comprimir-descomprimir-javascript.html

De igual manera, también se suele comprimir los ficheros CSS para reducir su peso:

https://herramientas-online.com/comprimir-descomprimir-css.html

comentarios

Los comentarios son zonas en el motor de JS va a ignorar y no va a tener en cuenta a la hora de ejecutar el código. Existen dos posibles maneras de introducir comentarios en el código JS:

Los comentarios de una única línea comienzan con dos barras inclinadas `//` y finalizan al final de la línea.

Los comentarios de bloque comienzan con `/*` y finalizan con `*/`, pudiendo cubrir varias líneas.

```
//Este es un comentario de una línea completa

sentencia 1; //Este es un comentario de parte de una línea

sentencia 2;

/* Esto es un comentario de bloque que puede
extenderse a lo largo de varias líneas */

sentencia 3;
```

Los comentarios durante el desarrollo son muy útiles para explicar partes del código para que luego resulte más fácil, a nosotros mismos u a otras personas, entenderlo y así mantenerlo.

No obstante, tampoco conviene abusar de los mismos porque pueden llegar a resultar excesivos y producir el efecto contrario, como, por ejemplo:

```
/*
==================
FUNCION CARRITO
==================
*/
```

palabras reservadas

Como cualquier lenguaje, JS posee un conjunto de palabras que utiliza para realizar tareas específicas y que por lo tanto no se pueden utilizar para nombrar a variables, funciones, constantes...

Este conjunto de palabras se conoce como "palabras reservadas".

Palabras reservadas				
break	export	super	continue	do
case	extends	switch	if	else
catch	finally	this	typeof	new
class	for	throw	var	with
const	function	try	import	enum
debugger	default	in	void	yield
delete	let	instanceof	while	return

Las siguientes palabras se incluirán en un futuro como reservadas pues tienen significado en JS:

Palabras reservadas en un futuro			
package	public	implements	private
interface	protected	await	static

Las siguientes palabras tienen significado en JS por lo que, aunque se pueden usar, conviene no hacerlo:

Palabras que conviene evitar usar			
null	undefined	true	Math
false	hasOwnProperty	undefined	NaN
isNaN	Infinity	isFinite	Number
alert	isPrototypeOf	valueOf	String
prompt	conform	prototype	Object
length	name	toString	

https://mothereff.in/js-variables

literales

Una literal es un valor constante formado por una secuencia de caracteres o números. Son ejemplos de literales los booleanos, los números, las cadenas de caracteres...

En JS los literales representan valores de tipo booleano, numérico o de cadena de caracteres. Estos son valores fijos, constantes y no variables.

Una cadena literal consta de cero o más caracteres encerrados entre comillas dobles (") o simples ('). Una cadena debe estar delimitada por comillas del mismo tipo (es decir, ambas comillas simples o, ambas comillas dobles).

Para representar a algunos caracteres se necesita una secuencia de escape, que se construyen con la barra hacia atrás y un carácter, por ejemplo: `\'` `\"` `\t`

```
20;
'Jorge';
'literal';
console.log ('mi literal');
```

```
'foo';
"bar";
'1234';
"una linea \n otra linea";
"Hola \" Jorge\"";
```

literales

Un literal de tipo booleano se escribe con *true* o *false*.

Un literal numérico se pueden escribir en decimal (en base 10), hexadecimal (base 16), octal (base 8) y binario (base 2).

Un literal numérico decimal es una secuencia de dígitos sin un *0* (cero) inicial.

Un *0* (cero) inicial en un literal numérico, o un *0o* inicial (o *0O*) indica que está en base octal.

Un *0x* inicial (o *0X)* indica un tipo numérico hexadecimal.

Un *0b* inicial (o *0B*) indica un literal numérico binario. Los números binarios solo pueden incluir los dígitos *0* y *1*.

```
0, 117, -34, 1489n           (decimal)
015, 001, -0o77, 0o77n       (octal)
0x13, 0x001, -0xFa, 0x1EFn   (hexadecimal)
0b11, 0b011, -0b11, 0b1101n (binário)
```

```
true;
21;
let valor=false;
let numero=07;
let registro=0b111;
```

literales

Un literal de coma flotante puede tener las siguientes partes:

- Un entero decimal que puede tener un signo (precedido por "*+*" o "*-*"),
- Un punto decimal ("*.*"),
- Una fracción (otro número decimal),
- Un exponente.
- La parte del exponente es una "*e*" o "*E*" seguida de un número entero, que puede tener signo (precedido por "*+*" o "*-*"). Un literal de coma flotante debe tener al menos un dígito y un punto decimal o "*e*" (o "*E*").

Específicamente la sintaxis es:

```
[(+|-)][dígitos].[dígitos][(E|e)[(+|-)]dígitos]
```

Existen otros tipos de literales como los objetos literales, expresiones regulares literales, plantillas literales… se irán estudiando a medida que avance el curso.

```
3.1415926;
-.123456789;
-3.1E+12;
.1e-23;
```

variables

Una variable es un contenedor para almacenar un valor, como un número o una cadena de texto. Realmente, son espacios de la memoria reservados para almacenar datos, el nombre de la variable hace referencia a ese lugar.

Las variables en JS se crean (*definen*) con la palabra reservada `Let` seguido del nombre que se elija para la variable:

```
Let numero;
```

Se puede dar un valor inicial a la variable (*inicializarla*) cuando se define utilizando el signo igual junto con un literal, este último será el valor asignado a la variable:

```
Let numero=6;
```

Para nombrar las variables se puede usar cualquier palabra que comience por una letra o _ (no pueden empezar por números) y que no constituya una palabra reservada, tampoco puede contener espacios.

Si se intenta ver el valor de una variable no inicializada se obtendrá *undefined*:

```
Let numero;
console.log(numero);
```

Esto indica que la variable no está definida, realmente no está inicializada.

En la siguiente página web se pueden ver nombres de variables que se pueden usar pero que no conviene hacerlo:

https://mathiasbynens.be/notes/javascript-identifiers

variables

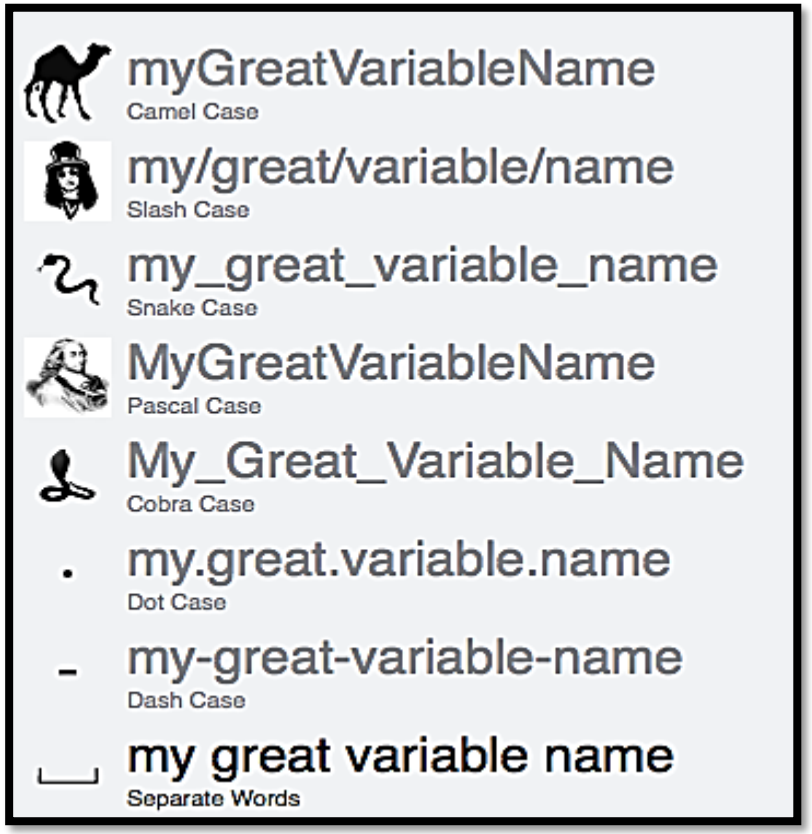
En JS se utiliza la notación Camel Case para nombrar las variables. En notación Camel Case el nombre de la variable está en minúsculas si solo es una palabra.

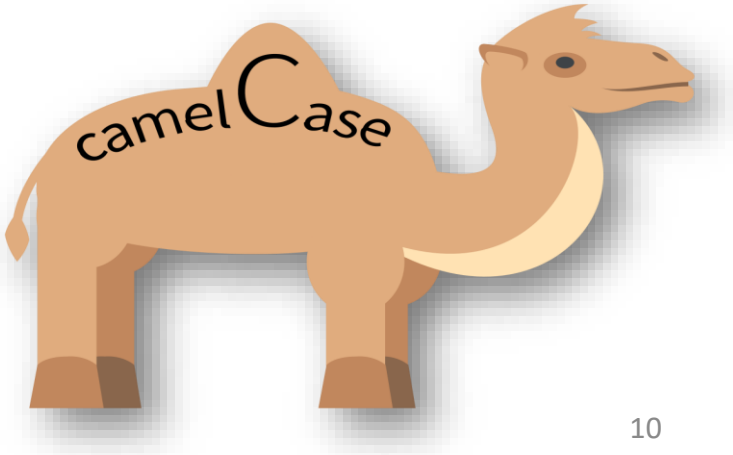
Si son varias palabras, como "*carrito compra*", como no se pueden usar espacios, la notación Camel Case combina las palabras directamente, sin usar ningún símbolo, estableciendo que la primera letra de cada palabra esté en mayúscula, a excepción de la primera palabra: "*carritoCompra*".

Otro ejemplo: "cliente tienda hotel" , en notación Camel Case: "*clienteTiendaHotel*".

Otras notaciones: "*alumno ciclo*":

Camel Case: alumnoCiclo **Pascal Case:** AlumnoCiclo
Snake Case: alumno_ciclo **Kebab Case:** alumno-ciclo





variables

let fue introducido en JS con ECMAScript2015 (ES2015 o ES6), antes solo se utilizaba *var* para definir variables.

La palabra reservada *let* declara una variable con ámbito local o de bloque (*block scope*), limitando su alcance (*scope*) al bloque, declaración, o expresión donde aparece la sentencia de declaración.

let se inicializa a un valor sólo cuando un analizador lo evalúa.

```
let apellido='Garcia';
console.log (apellido);
```

```
let apellido='Garcia';
console.log (apellido);
```

```
Garcia
```

variables

Cuando hay que definir o inicializar muchas variables se puede simplificar evitando poner muchos *let*, excepto el primero, y separando cada nueva variable por comas:

```
let variable1=1;
let variable2=2;
let variable3=3;
let variable4=4;
let variable5=5;
let variable6=6;
```

=

```
let variable1=1,
    variable2=2,
    variable3=3,
    variable4=4,
    variable5=5,
    variable6=6;
```

=

```
let variable1=1,variable2=2,…variable6=6;
```

El valor de la variable puede ir "variando" a lo largo del programa, mediante asignaciones de valores, se realizan poniendo el nombre de la variable, un igual, y el nuevo valor de la variable.

```
let variable=1;

variable=2;

variable;
```

Para conocer o acceder al valor que almacena una variable en un determinado momento se invoca el nombre de la variable.

scope

El ámbito de una variable (*scope*) es la zona del programa en la que se puede acceder y exista una variable.

JS define dos ámbitos para las variables: global y local.

Una variable se puede definir solamente con inicializarla, esto es, asignando un nombre a la variable (sin anteponer ninguna palabra reservada) y dándole seguidamente un valor. Hacerlo de esta manera hará que tenga ámbito global (se crea en el objeto *window*) y estará definida en cualquier punto del programa (incluso dentro de cualquier función u otros ficheros):

```
nombre='Jorge';
```

Esto supone un gasto de memoria innecesario, además, existe la posibilidad de poder colisionar con otras variables definidas por el navegador, por otros desarrolladores del equipo o por nosotros mismos, Por estos motivos, no se recomienda su uso.

JS es un lenguaje *Case Sensitive*, sensible a mayúsculas y minúsculas, esto quiere decir que NO es igual:
"*Jorge*" que "*jorge*"

Incluso en la sintaxis del lenguaje:

console.log();

No es lo mismo que

Console.log();

En este último caso, el motor dará un error porque no entiende que se le quiere decir.

var

Como se ha comentado, también se puede utilizar *var* para definir variables, de la misma forma que se hace con *let*, aplicándose para el nombre de la variable las mismas reglas, sin embargo, hay diferencias entre usar *let* y *var*.

El alcance o ámbito de *var* depende de donde se declare:

- Si se declara dentro de una función la variable tendrá alcance local, solo accesible dentro de la función.

- Si se declara fuera de una función (se estudiarán en próximos temas) la variable tendrá alcance global (se crea en el objeto *window*) y estará definida en cualquier punto del programa (incluso dentro de cualquier función u otros ficheros).

```
{
    var nombre='Jorge';
    let apellido='Garcia';
}
console.log (nombre);
console.log (apellido);
```

```
Jorge
Uncaught ReferenceError: apellido is not defined
```

var

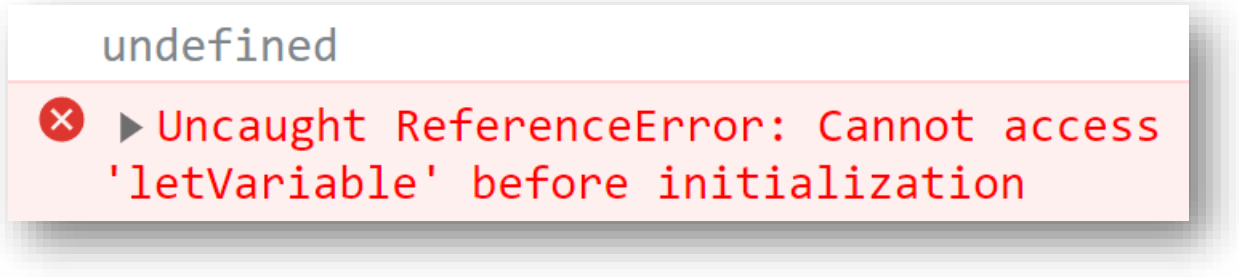
Cuando se declara una variable se procesa por el motor antes de ejecutar cualquier código, así, al declarar una variable en cualquier parte del código es equivalente a declararla al inicio del mismo.

Este comportamiento es llamado *Hoisting* (elevación) ya que la declaración de una variable parece haber sido movida colocada al principio de una función o al comienzo del código. El *hoisting* afecta de manera diferente si la variable ha sido declarada con *var* o con *let*.

El *hoisting* normalmente sólo afecta a la declaración de la variable, pero no su inicialización, y su valor será asignado en el momento que se alcance la sentencia de asignación, salvo que sea declarada con *var*, en ese caso la variable se declara y se inicializa como *undefined*.

```
console.log(variable);
console.log(letVariable);

var variable = 5;
let  letVariable = 1;
```



var

Otra característica propia de las variables declaradas con *var* es que se pueden redeclarar, y si la variable redeclarada no se le asigna un nuevo valor, la variable mantendrá su valor actual.

En el siguiente ejemplo, se puede observar que mientras que las variables declaradas con *var* permiten volver a declararlas, las variables declaradas con *let* darán error.

```
var variable = 5;
var variable = 50;
let  letVariable = 1;
let  letVariable = 20;
```

```
Uncaught SyntaxError: Identifier 'letVariable' has already been declared
```

En el ejemplo de la derecha, debido al *hoisting*, la instrucción *var variable;* es "movida" al inicio del código fuente, así, la instrucción *variable = 5;* asigna el valor 5 a la variable "ya declarada".

```
{
  variable = 5;
  var variable;
}
console.log(variable);
```

```
5
```

Estas características de las variables declaradas con *var* pueden resultar confusas con comportamientos “extraños” o “no esperados”.

var proviene de épocas remotas del lenguaje JS y uso en la actualidad no se recomienda en absoluto, sobrevive en las últimas especificaciones únicamente para hacer todavía funcional el código antiguo escrito en JS, esta retrocompatibilidad evita que dejen de funcionar webs antiguas en navegadores modernos.

Como buenas prácticas, a fin de evitar problemas, se deben declarar siempre las variables, evitando el ámbito global, usando siempre *let* y *const*, esto permite a un grupo de desarrolladores no colisionar en el mismo ámbito al utilizar el mismo nombre para una variable con usos distintos.

constantes

En ocasiones, dentro del código se necesita almacenar valores que no van a cambiar, permanecen inmutables durante toda la ejecución del mismo. En este caso, es preferible utilizar la palabra *const* en vez de *let* (entre otras cosas porque consumen menos cantidad de memoria).

const define lo que se llama una constante y permite almacenar un valor en memoria y recuperarlo mediante el nombre de la constante, de la misma forma que se hace con las variables. Para el nombre de la constante se aplican las mismas reglas para el nombre de las variables, y su ámbito es de bloque, al igual que las variables definidas con *let*.

A diferencia de las variables, las constantes cuando se definen tienen que ser inicializadas, pues su valor permanece desde ese momento inalterable, y si se intenta cambiarlo se obtendrá un error.

```
const iva = 16;
console.log (iva);

//Esto no esta permitido
iva = 21;
```

tipos de datos

Un tipo de datos es la propiedad de un dato que determina qué rango de valores puede tomar, qué operaciones se le pueden aplicar, cómo es representado y cómo se almacena internamente en la memoria del ordenador.

JS es un lenguaje débilmente tipado y tipado dinámicamente. No hace falta declarar los tipos de las variables, y la conversión de tipo se hace dinámicamente por lo que las variables son, en cada momento, del tipo del valor que contienen.

```
let variable;
console.log (typeof variable);

variable='Jorge';
console.log (typeof variable);

variable=1;
console.log (typeof variable);
```

```
>undefined

>string

>number
```

La palabra reservada *typeof* seguido de un nombre de variable o constate indica el tipo de dato que almacena.

tipos de datos

En JS existían tradicionalmente los siguientes tipos de datos:

- *NUMBER*
- *STRING*
- *ARRAY*
- *BOOLEAN*
- *OBJECT*
- *UNDEFINED*
- *NULL*

Estrictamente hablando, *ARRAY* y *OBJECT* no son tipos de datos primitivos.

Cuando se mezclan diferentes tipos de datos (mediante operadores o funciones) se realiza una conversión automática.

Ejemplo: en caso de conflicto si hay números y cadenas (*strings*) con el operador *+*, tienen preferencia las cadenas.

Se llama tipo primitivo o tipo elemental a los tipos de datos originales de un lenguaje de programación, esto es, aquellos que nos proporciona el lenguaje y con los que podemos construir tipos de datos abstractos y estructuras de datos.

```
let numero=20;
let cadena='Jorge';
let resultado=cadena + numero;
console.log (typeof resultado);
```

```
let numero=20;
let cadena='10';
let resultado=cadena + numero;
console.log (resultado);
```

operadores aritméticos

Un operador es un elemento de programa que se aplica a uno o varios operandos en una expresión o instrucción, El operador genera un resultado.

En JS hay operadores unarios, binarios y ternario.

Reciben como entrada tipos de datos numéricos.

+	Suma	*num1 + num2*
-	Resta	*num1 - num2*
-	Menos unario	*num1= -num1*
*	Multiplicación	*num1 * num2*
/	División	*num1 / num2*
%	Módulo (resto de la división)	*num1 % num2*
++	Post incremento	*num1++*
++	Pre incremento	*++num1*
--	Post decremento	*num1--*
--	Pre decremento	*--num1*
+=	Suma y asignación	*num1 += num2*
-=	resta y asignación	*num1 -= num2*
*=	multiplicación y asignación	*num1 *= num2*
/=	división y asignación	*num1 /= num2*
%=	modulo y asignación	*num1 %= num2*

operadores de comparación

Reciben como entrada tipos de datos.

==	Igual a	*dato1 == dato2*
===	Exactamente igual a (valor y tipo)	*dato1 === dato2*
!=	Distinto de	*dato1 != dato2*
!==	Exactamente distinto a (valor y tipo)	*dato1 !== dato2*
>	Mayor que	*dato1 > dato2*
>=	Mayor o igual a	*dato1 >= dato2*
<	Menor que	*dato1 < dato2*
<=	Menor o igual a	*dato1 <= dato2*

operadores lógicos

Reciben como entrada tipos de datos.

&&	Y lógico	*dato1 && dato2*
||	O lógico	*dato1 || dato2*
!	Negación lógica	*!dato1*

otros operadores

=	Asignación	*variable = 2*
+	Concatenación (en strings)	*cadena1 + cadena2*
this	Sirve para hacer referencia al objeto actual	*this.propiedad=2*
new	Crea una instancia de un objeto	*new String()*
typeof	Devuelve el tipo de una variable	*typeof variable*
&	Y binario	*variablebyte & 1*
|	O binario	*variablebyte | 2*
~	Negación binaria	*~variablebyte*
^	XOR	*variablebyte ^ 3*
<<	Desplazamiento de bits a la izquierda	*variablebyte << 1*
>>	Desplazamiento de bits a la derecha	*Variablebyte >> 2*
&=	Y binario y asignación	*num1 &= num2*
|=	O binario y asignación	*num1 |= num2*
^=	XOR binario y asignación	*num1 ^= num2*

conversión de tipos de datos

El último estándar ECMAScript define ocho tipos de datos, siete de estos tipos de datos son primitivos:

boolean. *true* y *false*.

null. Una palabra clave especial que denota un valor nulo. (Debido a que JS distingue entre mayúsculas y minúsculas, null no es lo mismo que Null, NULL o cualquier otra variante).

undefined. Una propiedad o variable cuyo valor no está definido.

number. Un número entero o un número con coma flotante. Por ejemplo: 42 o 3.14159.

bigInt. Un número entero con precisión arbitraria. Por ejemplo: 900719925474099 2n.

string. Una secuencia de caracteres que representan un valor de texto. Por ejemplo: "Hola"

symbol (nuevo en ECMAScript 2015) Un tipo de dato cuyas instancias son únicas e inmutables

object Incluye objetos y arrays. Este tipo de dato no es primitivo.

conversión de tipos de datos

En JS realmente todo son objectos menos los tipos de datos primitivos.

No obstante, el tipo de dato *null* es a menudo considerado de tipo objeto debido a que los motores de los navegadores y en los intérpretes de JS se considera de tipo *object*.

```
let variable=null;
console.log (typeof variable);
>object
```

El tipo de datos *symbol* es un tipo de dato primitivo que se utiliza para identificar propiedades de manera única e inmutable. Crear identificadores de manera única

En algunos lenguajes de programación son también llamados *atoms* (atómicos).

En JS, *symbol* es uno de los valores primitivos y el objeto de *symbol* es un envoltorio alrededor de un *symbol primitivo*, parecido al tipo de dato *null*

```
let simbolo1=Symbol();
let simbolo2=Symbol();

console.log (typeof simbolo1);
>symbol
console.log (simbolo1==simbolo2);
>false
```

conversión de tipos de datos

JS es un lenguaje débilmente tipado por lo que no se tiene que especificar el tipo de dato de una variable cuando se declara.

Además, es de tipado dinámico, por lo que los tipos de datos se convierten automáticamente según sea necesario durante la ejecución del script.

```
let variable=7;
variable='hola';
```

Como ya se ha visto, JS convierte los valores numéricos en cadenas en expresiones que involucran valores numéricos y de cadenas de caracteres con el operador *+*.

Pero JS no convierte valores numéricos en cadenas con el resto de operadores.

Ejemplo:

```
console.log ('37' - 7);
>30
console.log ('37' + 7);
>377
```

conversión de tipos de datos

En el caso de que haya un valor de tipo *string* y se quiera representar como *number* (número) existen dos métodos para la conversión:

parseInt()
parseFloat()

parseInt solo devuelve números enteros, por lo que se pierden los decimales.

Además, una práctica recomendada para *parseInt* es incluir siempre el parámetro *radix*.

El parámetro *radix* se utiliza para especificar la base del número.

Su sintaxis es la siguiente:

```
parseInt(string, [radix])
```

parseFloat devuelve números en coma flotante.

Su sintaxis es la siguiente:

```
parseFloat(string)
```

Ejemplos:

```
console.log(parseInt('2.5'));
>2
console.log(parseFloat('2.5'));
>2.5
```

conversión de tipos de datos

Si se intenta convertir una cadena de caracteres en número con *parseInt* o *parseFloat* y no es posible se obtendrá *NaN* (*Not a Number*).

```
console.log(parseInt('Hola'));
>NaN
```

No obstante, el tipo devuelto por *parseInt* o *parseFloat* es un *number*.

Ejemplo:

```
const numero=parseFloat('Hola');
console.log(typeof numero);
```

Saldrá por la consola *number*.

conversión de tipos de datos

En el caso de que haya un valor de tipo *number* o *bigInt* y se quiera convertir a *string* se puede usar el método *toString()*.

Su sintaxis es la siguiente:

```
numero.toString([radix])
```

El parámetro *radix* se utiliza para especificar la base del número.

Ejemplo:

```
const numero=2;
console.log (numero.toString());
>2
```

toString() devuelve un *string*, con el número convertido en cadena de caracteres.

template strings

Desde ES2015 (ES6) también están disponibles las plantillas literales o *Template Strings*.

Las plantillas literales están encerradas por la comilla invertida (`) (acento grave) en lugar de comillas simples o dobles.

Los *template strings* proporcionan una forma sintáctica más enriquecida de construir cadenas.

Permiten crear de forma sencilla:

- Cadenas con variables dentro (interpolación).
- Generar cadenas multilínea.
- Ejecutar expresiones, funciones y etiquetados.

Ejemplos:

```
// Creación de cadenas literales básicas
`en JS '\n' es un retorno de línea.`
```

```
// Cadenas multilínea
`En JS, los template strings pueden
ocupar varias líneas, a diferencia de
las cadenas entre comillas dobles o
 simples que no pueden.`
```

```
// Interpolación de cadenas
const nombre='Jorge', apellido='García';
console.log(`Hola ${nombre} ${apellido}`);
>Hola Jorge García
```

alert, prompt y confirm

Se puede tener interactividad básica con el usuario, a través del navegador web, mediante tres métodos del objeto *window*: *alert*, *prompt* y *confirm*.

En la actualidad, estos métodos se usan poco o nada, debido a que existen métodos visualmente más atractivos utilizando librerías.

alert muestra un diálogo de alerta con mensaje opcional y un botón Aceptar.

Su sintaxis es la siguiente:

```
alert([mensaje])
```

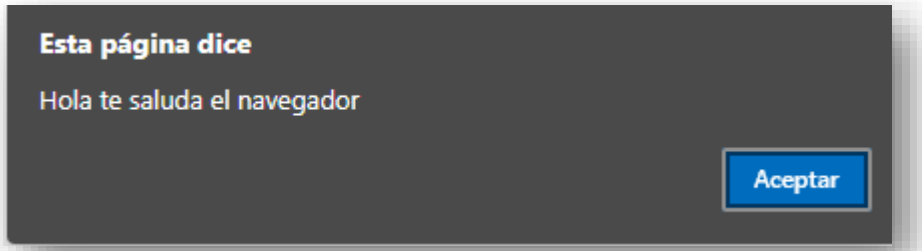
La ventana con el mensaje se llama "ventana o ventana de dialogo modal". La palabra "modal" significa que la "página web queda bloqueada" y el usuario no puede interactuar con ella hasta que se haya hecho clic en el botón Aceptar.

Corchetes en la sintaxis [ ]

En la sintaxis, los corchetes indican que el parámetro es opcional, no requerido, y por lo tanto funcionará, aunque no se pongan en la instrucción.

Ejemplo:

```
alert ('Hola te saluda el navegador');
```



alert, prompt y confirm

prompt muestra una ventana de diálogo a la espera de que el usuario le introduzca algún dato.

prompt acepta dos argumentos:

title El texto o mensaje a mostrar al usuario.

default Un segundo parámetro opcional, el valor por defecto como respuesta al mensaje.

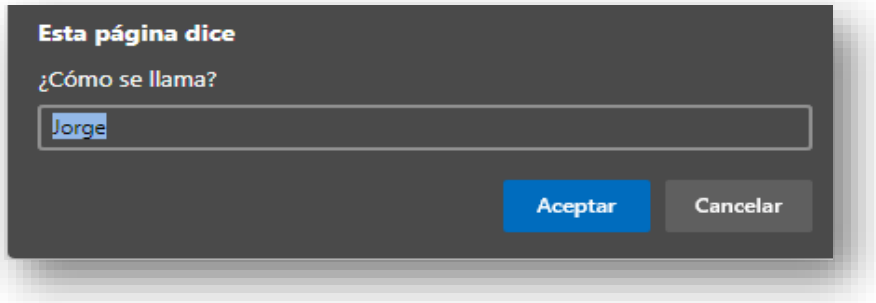
Su sintaxis es la siguiente:

```
prompt([title], [default])
```

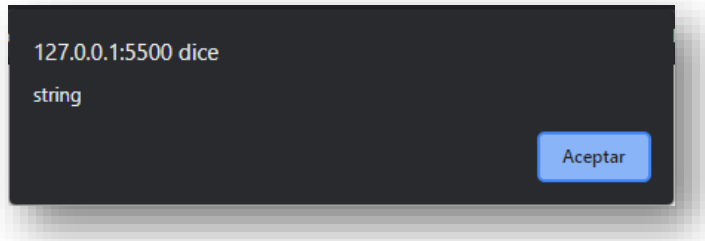
prompt devuelve el valor que ha introducido por el usuario en tipo *string*. Si no escribe nada y no valor defecto será un *string* vacío. Si el usuario pulsa el botón cancelar el valor devuelto es *null*.

Ejemplos:

```
let nombre=prompt('¿Cómo se llama?','Jorge')
```




```
let respuesta=prompt('¿Su número favorito? ',1);
alert (typeof respuesta);
```



alert, prompt y confirm

confirm muestra una ventana modal con una pregunta y dos botones: *Aceptar* y *Cancelar*.

Devuelve un booleano: *true* si se pulsa *Aceptar* y *false* en caso contrario.

confirm acepta un argumento:

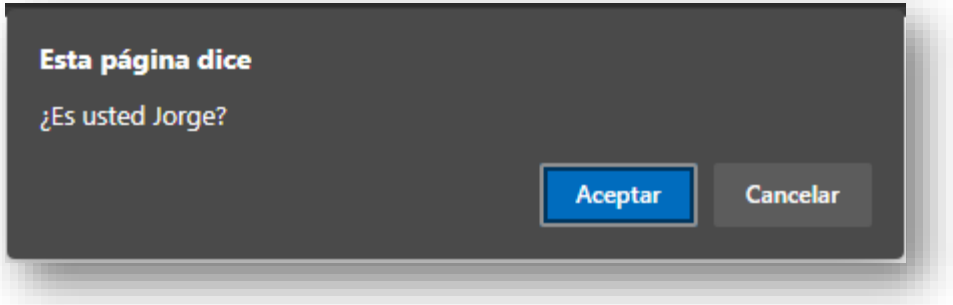
mensaje el texto o mensaje a mostrar al usuario con la pregunta.

Su sintaxis es la siguiente:

```
confirm([mensaje])
```

Ejemplo:

```
let respuesta=confirm('¿Es usted Jorge?');
```



Bibliografía y recursos online

- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_types
- https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/Variables
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_types
- https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/Variables
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/var