



Academy_

FULL STACK DEVELOPER BOOTCAMP

Desarrollamos
{ talento }

JAVASCRIPT

Historia

- Fue creado por **Brendan Eich** en Netscape en el año 1995 con la finalidad de hacer páginas web
- Inicialmente se llamaba Mocha y lo renombraron a LiveScript
- Finalmente cuando Netscape fue adquirida por Sun Microsystem lo cambiaron a JavaScript
- Aparece por primera en Netscape Navigador 2.0
- Es usado por todos los navegadores y webview móviles actuales.

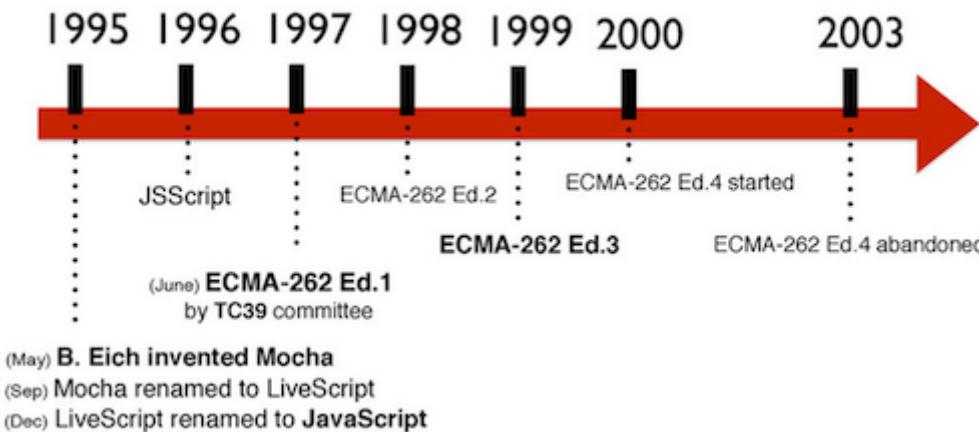
Historia

En 1997 se crea un comité (TC39) para estandarizar Javascript por la European Computer Manufacturer's Association, ECMA

Se diseña el estandar DOM (Document Object Model) para evitar las incompatibilidades entre navegadores.

Primeros estándares

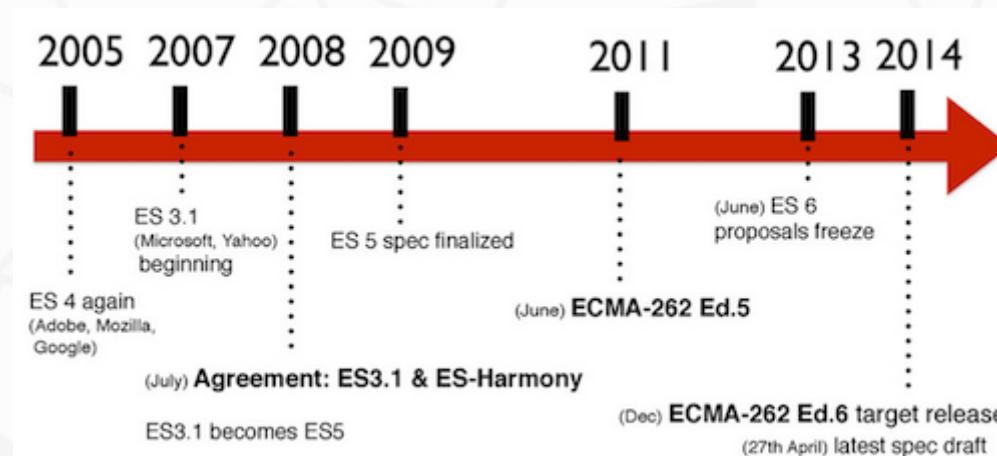
A partir de ese momento se generan los primeros estándares.



Primeros estándares

En 1999 aparece la versión 3 del estándar ECMAScript, que es la que más estable se ha mantenido.

Pero hasta 2011 no se aprobó y estandarizó la versión 5 (ES5) que es la que utilizamos hoy en día.



ECMASCRIPT 6

En 2015 se estableció la última versión ECMASCIPT 6

- Soporte Actual de EMASCIPT 6

Estándar de JavaScript

Viene definido por el ECMA Internacional

[Standard ECMA-262](#)

¿Qué es JavaScript?

- JavaScript es un lenguaje de programación de **alto nivel interpretado**.
- Es un lenguaje orientado a objetos
- Es un lenguaje orientado a eventos
- Es Débilmente tipado
- Dinámico
- Está basado en prototipos

Característica de JavaScript

- Es un lenguaje liviano
- Multiplataforma, ya que puede utilizar en cualquier Sistema Operativo
- Es interpretado, no se compila para poder ejecutarse.
- Podemos utilizarlo tanto en el Front com en el Back
- Excelente para trabajar en un mismo stack

Que necesitamos

- Un navegador web: Chrome, Firefox, etc.
- un editor de texto o IDE: WebStorm, Atom, Brackets, Sublime, etc.
- Herramientas de depuración del navegador.

Como utilizar JavaScript en la parte Front

- Utilizando la etiqueta script en el propio html

```
<script>
    console.log("Hola Mundo");
</script>
```

- Importando el script

```
<script src="main.js"/>
```

Sintaxis

- No se tienen en cuenta los espacios en blanco y las nuevas líneas
- Se distinguen las mayúsculas y minúsculas
- No se definen el tipo de las variables.
- No es necesario terminar cada sentencia con el carácter de punto y coma
- Se puede incluir comentarios

Comentarios

Los comentarios no se interpretan por el navegador

```
// ->Doble barra para comentar una línea.  
/*  
var a = 42;  
*/ ->Comentar varias líneas
```

Tabulaciones y saltos de línea:

- JavaScript ignora los espacios, las tabulaciones y los saltos de línea con algunas excepciones.
- Emplear la tabulación y los saltos de línea mejora la presentación y la legibilidad del código.

Palabras reservadas

- Algunas palabras no se pueden utilizar para definir nombres de variables, funciones o etiquetas.
- Es aconsejable no utilizar tampoco las palabras reservadas para futuras versiones de JavaScript

Palabras reservadas JavaScript/ES6

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

Tipos de datos

Los tipos de datos especifican qué tipo de valor se guardará en una determinada variable.

Los tipos datos se pueden categorizar en dos grupos:

- Simples: Son los tipos básicos o primarios
 - number 1
 - string 'holá'
 - boolean true
 - undefined undefined
 - function function() {}
- Complejos: son estructuras de datos que contienen otros tipos de datos
 - object
 - object / null
 - object / Array

Los nuevos tipos de datos que ofrece Ecmas 6 son:

- Symbol
- object / Map
- object / weakMap
- object / Set
- object / weakSet

Tipos de datos básicos

Números

- En JavaScript existe sólo un tipo de dato numérico.
- Todos los números se representan a través del formato de punto flotante de 64 bits. (max 10^{308})
- Esta forma es el llamado double en los lenguajes Java o C++.

```
89; // Numero
```

Valores Booleanos

- También conocido como valor lógico
- Sólo admite dos valores: true o false
- Es muy útil a la hora de evaluar expresiones lógicas o verificar condiciones.

```
true;  
false;
```

Tipo Undefined

Una variable a la cual no se le haya asignado valor entonces el valor es undefined

```
undefined;
```

Cadenas de texto

- El tipo de datos para representar cadenas de texto se llama string.
- Se puede representar letras, dígitos, signos de puntuación o cualquier otro carácter de Unicode.
- La cadena de caracteres se debe definir entre comillas dobles o comillas simples.

```
'1 Esta es una cadena con comillas simples.';  
'2 Esta cadena tiene comillas dobles. @';  
'Podemos utilizar emojis: 🍕';
```

Cadenas de escape

Podemos "escapar" por ejemplo una comilla doble con la contrabarra "\\" para que se interprete como un caracter literal en vez del fin de la cadena de texto.

```
"esto es un string"  
"esto es un comilla \" dentro de un string"  
'esto es un comilla " dentro de un string'  
" esto " da error " // Error"  
"La contrabarra se escribe con doble contrabarra \\"
```

Tipo Symbol(EM6)

El **Symbol** es un nuevo tipo en JavaScript introducido en la versión ECMAScript Edition 6. Un Symbol es un valor primitivo único e inmutable y puede ser usado como la clave de una propiedad de un Object

Es un principio de acercamiento a la **metaprogramación** por parte de javascript

Tipos de datos complejos: Objetos

Object

Un objeto es un valor en memoria al cual es posible referirse mediante un identificador

En JavaScript los objetos pueden ser vistos como una colección de propiedades. Con la sintaxis literal de objetos, un limitado grupo de propiedades son inicializadas; luego pueden ser agregadas o eliminadas otras propiedades. Los valores de las propiedades pueden ser de cualquier tipo, incluyendo otros objetos lo cual permite construir estructuras de datos complejas. Las propiedades se identifican usando claves. Una clave es un valor String o Symbol.

Sintaxis objeto

Asocia una clave con un valor y tiene los siguientes atributos:

```
{  
  clave: "valor",  
  clave2: 1234,  
}
```

Clave: valor

En definitiva un objeto es una lista de pares de llaves y valores

Pero también tenemos funciones que son objetos regulares con una capacidad adicional de poder ser llamadas o invocadas.

```
var o = {  
    str: 'asdas',  
    num: 87398,  
    f: function() {},  
    bool: true,  
    arr: [],  
    obj: {},  
    und: undefined,  
};  
o.str; // 'asdas'  
o['str']; // 'asdas'  
var s = 'str';  
o[s]; // o['str'] -> 'asdas'  
o.s; // undefined  
s = 'arr';  
o[s]; // o.arr -> []  
s = 'und';  
o[s]; // o.und -> undefined
```

Array

Un array es un conjunto ordenado de valores relacionados, estos valores se denominan elementos y cada elemento dispone de un índice que indica su posición numérica en el array.

Declaración de Arrays

Para la declaración de los Arrays podemos realizar cualquier de las siguientes formas:

```
var nombreDelArray1 = new Array();
// igual que:
var nombreDelArray2 = [];
//O
var nombreDelArray3 = new Array('item1', 2, true, 'hola');
// igual que:
var nombreDelArray4 = ['item1', 2, true, 'hola']; // FORMA PREFERIDA
```

Accediendo a sus items

```
var miArray = [ '1', 'hola', 'mundo', { a: 1 }, [2, 3]];  
  
miArray[0]; // '1'  
  
miArray[3]; // {a:1}  
  
miArray[99]; // undefined  
  
miArray[4]; // [2,3]  
  
miArray[4][0]; // 2  
  
miArray[4] = true;  
  
miArray; // ['1','hola','mundo', {a:1}, true ];
```

Null

El valor null es un literal de Javascript que representa objeto nulo o vacío.

Su tipo es object:

```
typeof null; // 'object'
```

Date

Uno de los grandes problemas que comparten todos los lenguajes de programación son el tratamiento de las fechas.

Javascript dispone de un objeto especial llamado Date.

```
var date1 = new Date('December 17, 1995 03:24:00');
// Sun Dec 17 1995 03:24:00 GMT...
var date2 = new Date('1995-12-17T03:24:00');
// Sun Dec 17 1995 03:24:00 GMT...
// Su tipo es object
typeof date2; // 'object'
```

Variables

Introducción

Se pueden definir como zonas de la memoria de un ordenador que se identifican con un nombre y en las cuales se almacenan ciertos datos

El desarrollo de un script conlleva:

- Declaración de variables.
- Inicialización de variables.

Declaración de una variable

"var" define una variable global o local

```
var miVariable;
```

Inicialización de variables.

Se puede asignar un valor a una variable de tres formas básicas:

- Asignación directa de un valor concreto.
- Asignación indirecta a través de un cálculo en el que se implican a otras variables o constantes.
- Asignación a través de la solicitud del valor al usuario del programa.

Inicialización de variables.

```
// Asignación directa.
```

```
var variable1 = 39;
```

```
// Asignación indirecta a través de un cálculo.
```

```
var variable2 = variable1 + 20; // 59
```

```
// Asignación y declaración en diferentes líneas
```

```
var variable3;
```

```
variable3 = 100;
```

```
// Asignación a través de la solicitud de un usuario.
```

```
var variable4 = prompt('Introduce un valor');
```

Operadores

Tipos

JavaScript tiene los siguientes tipos de operadores

- Aritméticos.
- Lógicos.
- De asignación.
- De comparación.
- Condicionales/Ternarios.
- Cadenas de caracteres.
- Operador coma
- Operadores unarios
- Operadores de bit a bit.

Operadores Aritméticos

Permite realizar cálculos elementales entre variables numéricas.

- + Suma
- - Resta
- * Multiplicación
- / División
- % Módulo
- ** exponente
- ++ Incremento
- -- Decremento

Operadores Aritméticos

Otros operadores Aritméticos Especiales

- - Negación unaria

```
console.log(-"3");
// -> -3, intenta convertir un numero al operando y devuelve su forma
negativa
```

- + Unario positivo

```
console.log(+''3'');  
// -> 3, intenta convertir un numero al operando y devuelve su forma  
positiva
```

- **** Exponente**

```
2 ** 3 //Devuelve 8  
10 ** -1 //Devuelve 0.1
```

Operadores Lógicos

Combinan diferentes expresiones lógicas con el fin de evaluar si el resultado de dicha combinación es verdadero o falso.

- && Y
- || O
- ! No

Operadores de Asignación

Permiten obtener métodos abreviados para evitar escribir dos veces la variable que se encuentra a la izquierda del operador

- = asigna var a = 1
- += Suma y asigna a += 2 // a = a + 2
- -= Resta y asigna a -= 2 // a = a - 2
- *= Multiplica y asigna a *= 2 // a = a * 2
- /= Divide y asigna a /= 2 // a = a / 2
- %= Módulo y asigna a %= 2 // a = a % 2
- **= Asignación de exponencación a **= 2 // a = a ** 2

Binarios:

- <<= Asignación de desplazamiento a la izquierda
- >>= Asignación de desplazamiento a la derecha
- >>>= Asignación sin signo de desplazamiento a la derecha
- &= Asignación AND
- ^= Asignación XOR
- |= Asignación OR

Operadores de Comparación

Permite comparar todo tipo de variables y devuelven un valor booleano

- < Menor que
- <= Menor igual que
- == Igual
- > Mayor que
- >= Mayor o igual que
- != Diferente
- === Estrictamente igual
- !== Estrictamente diferente

Operadores condicionales (o Ternarios)

Permiten indicar al navegador que ejecute una acción en concreto después de evaluar una expresión. `x ? 'x es cierto' : 'x es falso'`

```
var age = 22;  
var repuestaPortero = age > 18 ? 'puedes entrar' : 'no puedes entrar';
```

Operadores Cadenas de caracteres

Operador de concatenación es el único operador que permite sobrecarga +

```
var output = 'mi ' + 'cadena';
console.log(output); // "mi cadena"
```

```
var s1 = 'hola';
var s2 = 'mundo';
var hello = s1 + s2;
hello; // 'holamundo'
```

```
var x = 'hola' + 2 + 1; // 'hola21'
var y = 2 + 1 + 'hola'; // '3hola'
```

Operador coma

(,) evalúa ambos operandos y retorna el valor del último

```
function myFunc() {  
  var x = 0;  
  return (x += 1), x; // the same as return ++x;  
}
```

Operadores unarios

Los operadores unarios son aquellos que sólo necesita un operando

- typeof
- delete
- void
- in
- instanceof

Ejemplo typeof

```
typeof 1; // "number"
typeof true; // "boolean"
typeof []; // "object"
typeof {}; // "object"
typeof 'asd'; // "string"
typeof 1; // "number"
typeof true; // "boolean"
```

Operadores binarios o Bitwise

Son operadores que tratan su operando como una secuencia de 32 bits

- a & b And binario
- a | b Or binario
- a ^ b Xor Binario
- ~ a Bitwise not (Invierte los bits de operando)

Orden de prevalencia de los operandos

Operador	Descripción
.	Acceso a campos, indización de matrices, llamadas a funciones y agrupamiento de expresiones
++ -- ~ ! delete new typeof void	Operadores unarios, tipos de datos devueltos, creación de objetos, valores no definidos
* / %	Multiplicación, división, división módulo
+ - +	Suma, resta, concatenación de cadenas
<< >> >>>	Desplazamiento bit a bit
< <= > >= instanceof	Menor que, menor o igual que, mayor que, mayor o igual que, instanceof
== != === !==	Igualdad, desigualdad, igualdad estricta y desigualdad estricta
&	AND bit a bit
^	XOR bit a bit
OR bit a bit	
&&	AND lógico
OR lógico	
?:	Condicional
= OP=	Asignación, asignación con operación (como += y &=)
,	Evaluación múltiple

Control del flujo

Introducción

Con las sentencias condicionales se puede gestionar la toma de decisiones y el posterior resultado por parte del navegador. Dichas sentencias evalúan condiciones y ejecutan ciertas instrucciones en base al resultado de la condición

Las sentencias condiciones JavaScript son:

- if
- switch
- while
- for

Sentencias condicionales

Mediante if/else se permite plantear una condición que según si se cumpla o no se realizan diferentes acciones.

La estructura es:

```
if (condicion) {  
    //código si se cumple la condición  
} else {  
    //código si No se cumple la condición  
}
```

if

```
if (a > 2) {  
    result = 'a is greater than 2';  
}
```

if else

```
if (a > 2) {  
    result = 'a is greater than 2';  
} else {  
    result = 'a is NOT greater than 2';  
}
```

Es similar a operador ternario:

```
var result = (a > 2) ? 'a is greater than 2' : 'a is NOT greater than 2';
```

if else if

```
var a = 120;
if (a < 0) {
  result = 'a es menor que cero';
} else if (a < 10) {
  result = 'a entre 0 y 9 inclusive';
} else if (a < 100) {
  result = 'a entre 10 y 99';
} else {
  result = 'a es mayor que 100';
}
```

switch

Setencia switch tiene un único bloque de instrucciones y los distitnos casos se comportan como etiquetas

Es necesario emplear la instrucción break para no seguir con la ejecución del siguiente caso (y salir del switch). El caso "default" es opcional y la estructrua es:

```
switch(/*expresión*/) {  
    case valor1: instrucciones a ejecutar; break;  
    case valor2: instrucciones a ejecutar; break;  
    case valor3: instrucciones a ejecutar; break;  
    default: instrucciones a ejecutar;  
}
```

```
var a = 'hola';
switch (a) {
  case 'adios':
    log('a es adios');
    break;
  case 'valor2':
    log('a es valor2');
    break;
  case 'hola':
    log('a es hola');
    break;
  default:
    log('a es no es ni adios ni valor2 ni hola');
}
```

Bucles

Existen 5 tipos de sentencias repetitivas en JavaScript:

- Bucle While
- Bucle do/while
- Bucle for
- Bucle for in
- Bucle for of

While

Se ejecutarán las instrucciones de dentro del bucle while hasta que la condición evaluada sea igual a false. En dicho momento finalizará el bucle

El bucle "while" tiene la siguiente estructura:

```
while(/*expresión*/) {  
//código JavaScript si se cumple la expresión  
}
```

Ejemplo while

```
var i = 0;  
while (i < 10) {  
    i++;  
}
```

do While

Es una variación del bucle while la evaluación en este caso se realiza al final del ciclo y no al principio

```
do {  
//código JavaScript si se cumple la expresión  
} while(/*expresión*/)
```

Ejemplo do while

```
var i = 0;  
do {  
    i++;  
} while (i < 10);
```

For

Un bucle for se utiliza cuando se sabe a priori cuantas veces se tiene que repetir unas determinadas instrucciones.

Tiene la siguiente estructura:

```
for (;;) /*inicializador*/ /*comprobación*/ /*contador*/ {  
    //código a ejecutar  
}
```

- Inicializador: suele ser una expresión que da valor a una variable con tal de llevar la cuenta de repeticiones
- Comprobación: es una expresión que se tiene que evaluar para seguir ejecutando el bucle o detenerlo.
- Contador: Es el cambio que tiene la variable cada vez que se ejecuta el bucle

Ejemplo bucle for

```
for (var j = 0; j < 10; j++) {  
    console.log(j);  
}
```

for vs while

```
for (var j = 0; j < 10; j++) {  
    console.log(j);  
}  
// Exactamente igual que:  
var i = 0; // inicializador  
while (i < 10) {  
    console.log(i); // cuerpo de ejecución  
    i++; // contador  
}
```

Bucles for anidados

```
var res = '\n';
for (var i = 0; i < 10; i++) {
  for (var j = 0; j < 10; j++) {
    res += '* ';
  }
  res += '\n';
}
console.log(res);
```

Bucles for anidados

```
var matriz = [[ '00', '01', '02'], ['10', '11', '12'], ['20', '21',  
'22']];  
  
for (var i = 0; i < 3; i++) {  
    for (var j = 0; j < 3; j++) {  
        console.log(matriz[i][j]);  
    }  
}
```

For in

Se utiliza para recorrer los elementos de un objeto, itera devolviendo las claves de valor.

```
var obj = { first: 'John', last: 'Doe' };

for (var propiedad in obj) {
  console.log(propiedad); // 'first/last'
  console.log(obj[propiedad]); // 'John/Doe'
}
```

For of

La sentencia `for...of` crea un bucle que itera a través de los elementos de objetos iterables (incluyendo Array, Map, Set, el objeto arguments, etc.), ejecutando las sentencias de cada iteración con el valor del elemento que corresponda.

```
var iterable = [10, 20, 30];
for (var value of iterable) {
  console.log(value); // 10, 20, 30
}
```

Funciones

Introducción

En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones, que conforman el llamado cuerpo de la función. Se pueden pasar valores a una función, y la función puede devolver un valor.

Declaración de funciones

La definición de una función (también llamada declaración de función o sentencia de función) consiste de la palabra clave (reservada) `function`, seguida por:

- El nombre de la función (opcional).
- Una lista de argumentos para la función, encerrados entre paréntesis y separados por comas (,).
- Las sentencias JavaScript que definen la función, encerradas por llaves, {}.

El siguiente código define una función simple llamada square:

```
function square(number) {  
    return number * number;  
}
```

Expresiones de función

Si bien la declaración de la función anterior es sintácticamente una sentencia, las funciones pueden también ser creadas por una expresión de función. Tal función puede ser anónima; no debe tener un nombre. Por ejemplo, la función square podría haber sido definida como:

```
var square = function(number) {  
    return number * number;  
};  
var x = square(4); //x obtiene el valor 16
```

Llamando funciones

Definir una función no la ejecuta. Definir una función simplemente la nombra y especifica que hacer cuando la función es llamada. Llamar la función es lo que realmente realiza las acciones especificadas con los parámetros indicados. Por ejemplo, si define la función square, podría llamarla como sigue:

```
square(5);
```

La sentencia anterior llama a la función con el argumento 5. La función ejecuta sus sentencias y retorna el valor 25.

Usando el objeto arguments

Los argumentos de una función son mantenidos en un objeto similar a un array. Dentro de una función, los argumentos pasados a la misma pueden ser direccionados de la siguiente forma:

```
function myConcat(separator) {  
    var result = '', // initialize list  
    i;  
    // iterate through arguments  
    for (i = 1; i < arguments.length; i++) {  
        result += arguments[i] + separator;  
    }  
    return result;  
}  
// returns "red, orange, blue, "  
myConcat(', ', 'red', 'orange', 'blue');
```

Funciones predefinidas.

JavaScript cuenta con una serie de funciones integradas en el lenguaje.

Dichas funciones se pueden ejecutar sin conocer todas las instrucciones que ejecuta.

Simplemente se debe conocer el nombre de la función y el resultado que se obtiene al utilizarla.

Funciones predefinidas.

- escape()
- eval()
- isFinite()
- isNaN()
- Number()
- String()
- parseInt()
- parseFloat()
- encodeUri()
- decodeUri()
- encodeURIComponent()
- decodeURIComponent()

Función escape()

La función `escape()` tiene como argumento una cadena de texto y devuelve dicha cadena utilizando la codificación hexadecimal en el conjunto de caracteres latinos ISO. Por ejemplo, la codificación ISO del carácter "?" es %3F.

```
var input = prompt('Introduzca una cadena');
var inputCodificado = escape(input);
alert('Cadena codificada: ' + inputCodificado);
```

Función eval()

La función eval() convierte una cadena que pasamos como argumento en código JavaScript ejecutable

Por ejemplo, el usuario ingresa una operación numérica y a continuación se mostrará el resultado de dicha operación tras usar la función eval()

```
var input = prompt('Introduzca una operación numérica');
var resultado = eval(input);
alert('El resultado es la operación es: ' + resultado);
```

Función isFinite()

Comprueba si el valor pasado por parámetro no es infinita o NaN

```
console.log(isFinite(Infinity)) // false
console.log(isFinite(-Infinity)) // false
console.log(isFinite(12)) // true
console.log(isFinite(1e308)) // true
console.log(isFinite(1e309)) /* false, supera el limite de 64 bits,
por lo que Js lo considera infinito */
```

Función isNaN()

La función isNaN() comprueba si el valor que pasamos como argumento es de tipo numérico.

```
console.log(isNaN(NaN)) // true
console.log(isNaN(123)) // false
console.log(isNaN(1.23)) // false
console.log(isNaN(parseInt('abc123')))) // true
```

Función Number()



La función Number() convierte los valores de diferentes objetos en sus números

```
var x1 = true;
var x2 = false;
var x3 = new Date();
var x4 = '999';
var x5 = '999 888';

var n =
  Number(x1) +
  ', ' +
  Number(x2) +
  ', ' +
  Number(x3) +
  ', ' +
  Number(x4) +
  ', ' +
  Number(x5);

console.log(n); // 1, 0, 1528879296237, 999, NaN
```

Función String()



La función String() convierte los valores de diferentes objetos en string

```
var x1 = Boolean(0);
var x2 = Boolean(1);
var x3 = new Date();
var x4 = '12345';
var x5 = 12345;

var res =
  String(x1) +
  ', ' +
  String(x2) +
  ', ' +
  String(x3) +
  ', ' +
  String(x4) +
  ', ' +
  String(x5);

console.log(res); // false, true, Wed Jun 13 2018 10:43:09 GMT+0200
(GMT+02:00), 12345, 12345
```

Función parseInt()

La función parseInt() convierte la cadena que pasamos como argumento en un valor numérico de tipo entero. Como se puede comprobar, si tenemos un numérico con parte decimal, con el parseInt() se trunca y se queda únicamente con la parte entera.

```
var input = prompt('Introduce un valor: ');
var inputParseado = parseInt(input);
alert('parseInt(' + input + '): ' + inputParseado);
```

Función parseFloat()

La función parseFloat () convierte la cadena que pasamos como argumento en un valor numérico de tipo flotante.

```
var input=prompt("Introduce un valor");
var inputParseado=parseFloat(input);
alert("parseFloat ("+input+") : " + inputParseado);
```

Función encodeUri()

La función encodeUri() codifica la URI

```
var uri = 'my test.asp?name=ståle&car=saab';
var res = encodeURI(uri);
console.log(res); // my%20test.asp?name=st%C3%A5le&car=saab
```

Función decodeURI()

Decodificación de la URI

```
var uri = 'my test.asp?name=ståle&car=saab';
var enc = encodeURI(uri);
var dec = decodeURI(enc);
var res = 'Encoded URI: ' + enc + '' + 'Decoded URI: ' + dec;
console.log(res); /*
Encoded URI: my%20test.asp?name=st%C3%A5le&car=saabDecoded
URI: my test.asp?name=ståle&car=saab */
```

Prototypes

Array.prototype

La propiedad **Array.prototype** contiene propiedades y métodos al objeto Array

length

La propiedad **.length** nos devuelve el tamaño el array

```
var numeros = [1, 24, 456, 4];
console.log(numeros.length); // 4
for (var i = 0; i < numeros.length; i++) {
  console.log(numeros[i]);
}
```

Métodos

A continuación se detalla los métodos más usados de los Arrays

Método	Descripción
push()	Añade nuevos elementos al array y devuelve la nueva longitud del array
concat()	Selecciona un array y lo concatena con otros elementos en un nuevo array
join()	Concatena los elementos de un array en una sola cadena separada por un carácter opcional
reverse()	Invierte el orden de los elementos de un array
unshift()	Añade elementos al inicio de una array y devuelve el número de elementos del nuevo array modificado
shift()	Elimina el primer elemento de un array
pop()	Elimina el último elemento de un array
slice()	Devuelve un nuevo array con un subconjunto de los elementos del array que ha usado el método
sort()	Ordena alfabéticamente los elementos de un array. Podemos definir una nueva función para ordenarlos con otro criterio.
splice()	Elimina, sustituye o añade elementos del array dependiendo de los argumentos del método.

.push()

Añade nuevos elementos al array devuelve la nueva longitud del array

```
var numeros = [1, 2, 3];
var newLength = numeros.push(100);
console.log(newLength);
console.log(numeros); // [1,2,3,100]
numeros.push(200, 400);
console.log(numeros); // [1,2,3,100,200,400]
```

.concat()

Selecciona un array lo concatena con otros elementos en un nuevo array

```
var equipos_a = ['Valencia', 'Barça', 'Real Madrid'];
var equipos_b = ['Hercules', 'Zaragoza', 'Valladolid'];
var equipos_copa = equipos_a.concat(equipos_b);
console.log(equipos_copa); // ["Valencia", "Barça", "Real Madrid",
"Hercules", "Zaragoza", "Valladolid"]
console.log(equipos_a); // ["Valencia", "Barça", "Real Madrid"]
```

.join()

Concatena los elementos de un array en una sola cadena separada por un carácter opcional

```
var nombres = ['Manuel', 'Antonio', 'Pepe'];
console.log(nombres.join('-')) // "Manuel-Antonio-Pepe"
```

.reverse()

Invierte el orden de los elementos de un array

```
var nombres = ['Manuel', 'Antonio', 'Pepe'];
nombres.reverse();
console.log(nombres); // ["Pepe", "Antonio", "Manuel"]
```

.unshift()

Añade nuevos elementos al inicio de un array y devuelve el número de elementos del nuevo array modificado

```
var nombres = ['Manuel', 'Antonio', 'Pepe'];
var valor_nombres = nombres.unshift('Jesús');
console.log(valor_nombres); // 4
console.log(nombres); // ["Jesús", "Manuel", "Antonio", "Pepe"]
```

.shift()

Elimina el primer elemento de un array y devuelve el elemento extraido

```
var nombres = ['Manuel', 'Antonio', 'Pepe'];
var valor_resto = nombres.shift();
console.log(valor_resto); // Manuel
console.log(nombres); // [ 'Antonio', 'Pepe' ]
```

.pop()

Elimina el último elemento de un array y lo devuelve

```
var nombres = ['Manuel', 'Antonio', 'Pepe'];
console.log(nombres.pop()); // Pepe
console.log(nombres); // [ 'Manuel', 'Antonio' ]
```

.slice()

Devuelve un nuevo ARRAY con un subconjunto de los elementos del array que ha usado el método.

```
var numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var primerosTresNumeros = numeros.slice(0, 3); // devuelve 3 números
empezando desde 0 (incluido)
var ultimosCuatroNumeros = numeros.slice(-4); // Devuelve los últimos 4
números.
console.log(primerosTresNumeros); // [ 1, 2, 3 ]
console.log(ultimosCuatroNumeros); // [ 7, 8, 9, 10 ]
console.log(numeros); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

.splice()

Elimina, sustituye o añade elementos del array dependiendo de los argumentos

```
var peliculas = ['Superman', 'Superman 3'];
peliculas.splice(1, 0, 'Superman 2'); // Añade un elemento en la posicion
1
console.log(peliculas); // ["Superman", "Superman 2", "Superman 3"]
peliculas.splice(2, 1, 'Superman 4'); // Sustituye un elemento por la
cadena "Superman 4"
console.log(peliculas); // [ 'Superman', 'Superman 2', 'Superman 4' ]
peliculas.splice(0, 2); // Elimina 2 indices a partir del 0.
console.log(peliculas); // [ 'Superman 4' ]
```

.forEach()

```
var numeros = [6, 4, 2];
var iterador = function(valor) {
    console.log(valor);
};

// podemos iterar mediante forEach
numeros.forEach(iterador); // Output: 6, 4, 2

// es lo mismo que
iterador(numeros[0]); // iterador(6);
iterador(numeros[1]); // iterador(4);
iterador(numeros[2]); // iterador(2);
```

```
var numeros = [6, 4, 2];

numeros.forEach(function(item, index, array) {
    console.log(item, index, array);
});

/* Output:
   6 0 [ 6, 4, 2 ]
   4 1 [ 6, 4, 2 ]
   2 2 [ 6, 4, 2 ] */
```

.sort()

Ordena alfabéticamente los elementos de un array si no se paso como argumento una función de comparación

```
var nombres = ['Manuel', 'Antonio', 'Sergio', 'Ricardo', 'Francisco'];
nombres.sort();
console.log(nombres); // [ 'Antonio', 'Francisco', 'Manuel', 'Ricardo',
'Sergio' ]
```

// sin función de comparación se ordena alfabéticamente aunque sean números

```
var arr = [40, 1, 5, 200];
arr.sort();
console.log(arr); // [ 1, 200, 40, 5 ]
```

`arr.sort([compareFunction])` Si se provee compareFunction, los elementos del array son ordenados de acuerdo al valor que retorna dicha función de comparación. Siendo a y b dos elementos comparados, entonces:

- Si compareFunction(a, b) es menor que 0, se sitúa a en un indice menor que b. Es decir, a viene primero.
- Si compareFunction(a, b) retorna 0, se deja a y b sin cambios entre ellos, pero ordenados con respecto a todos los elementos diferentes. Nota: el estandar ECMAScript no garantiza este comportamiento, por esto no todos los navegadores (p.ej. Mozilla en versiones que datan hasta el 2003) respetan esto.
- Si compareFunction(a, b) es mayor que 0, se sitúa b en un indice menor que a.
- compareFunction(a, b) siempre debe retornar el mismo valor dado un par específico de elementos a y b como sus argumentos. Si se retornan resultados inconsistentes entonces el orden de ordenamiento es indefinido.

```
var arr = [40, 1, 5, 200];
arr.sort(function(a, b) {
    return a - b;
});
console.log(arr);
```

```
var items = [
  { name: 'a', value: 11 },
  { name: 'b', value: 22 },
  { name: 'a', value: 33 },
];

items.sort(function(a, b) {
  if (a.name > b.name) {
    return 1;
  }
  if (a.name < b.name) {
    return -1;
  }
  // a must be equal to b
  return 0;
});
console.log(items); /*
[ { name: 'a', value: 11 },
  { name: 'a', value: 33 },
  { name: 'b', value: 22 } ]
*/
```

.sort()

Sin función de comparación se ordena alfabéticamente

```
var arr = [12, 2, 8, 4, 0];
arr.sort();
console.log(arr); // [ 0, 12, 2, 4, 8 ]
```

Con función podemos personalizar la forma en la que se ordenan.

```
var arr = [12, 2, 8, 4, 0];
arr.sort(function(a, b) {
  return a - b; // Se ordena de menor a mayor
});
console.log(arr); // [ 0, 2, 4, 8, 12 ]
```

```
var arr2 = [12, 2, 8, 4, 0];
arr2.sort(function(a, b) {
  return b - a; // Se ordena de mayor a menor
});
console.log(arr2); // [ 12, 8, 4, 2, 0 ]
```

.sort()

Podemos ordenar los elementos de un array de objetos por mas de un factor:

```
var arrObjetos = [
  { fila: 2, columna: 2, z: 1 },
  { fila: 1, columna: 2, z: 0 },
  { fila: 3, columna: 2, z: 100 },
  { fila: 3, columna: 2, z: 1 },
  { fila: 3, columna: 2, z: 3 },
  { fila: 2, columna: 0, z: 0 },
  { fila: 2, columna: 10, z: 1 },
];
arrObjetos.sort(function(a, b) {
  return a.fila - b.fila || b.columna - a.columna || a.z - b.z;
});
console.log(arrObjetos); /*
[ { fila: 1, columna: 2, z: 0 },
  { fila: 2, columna: 10, z: 1 },
  { fila: 2, columna: 2, z: 1 },
  { fila: 2, columna: 0, z: 0 },
  { fila: 3, columna: 2, z: 1 },
  { fila: 3, columna: 2, z: 3 },
  { fila: 3, columna: 2, z: 100 } ] */
```

.map()



Crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
  return x * 2;
});
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]

// pasa la funcion como argumento cada uno de los elementos
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]

// Es la forma corta de hacer esto:
roots = numbers.map(function(number) {
  return Math.sqrt(number);
});
console.log(roots);
```

.filter()

Crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction',
'present'];

const result = words.filter(function(word) {
  return word.length > 6;
});

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

.reduce()

Aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor.

```
// sintaxis
var resultado = arr.reduce(funcion[, valorInicial]);
// primero se pone valor 1000(segundo argumento) y despues reduce el
array
var total = [1, 2, 3, 4, 5].reduce(function(
  valorAnterior,
  valorActual,
  indice,
  vector
) {
  return valorAnterior + valorActual;
},
1000);
console.log(total); // 1015
```

.find()

Retorna el primer valor que cumpla con las condiciones

```
var array1 = [5, 12, 8, 130, 44];  
  
var found = array1.find(function(element) {  
    return element > 10;  
});  
  
console.log(found);  
// expected output: 12
```

String.prototype

Number.prototype

Object.prototype