
FULL STACK
DEVELOPER
BOOTCAMP
Desarrollamos
{ talento }

FUNDAMENTOS DE PROGRAMACIÓN

INTRODUCCIÓN A JAVASCRIPT

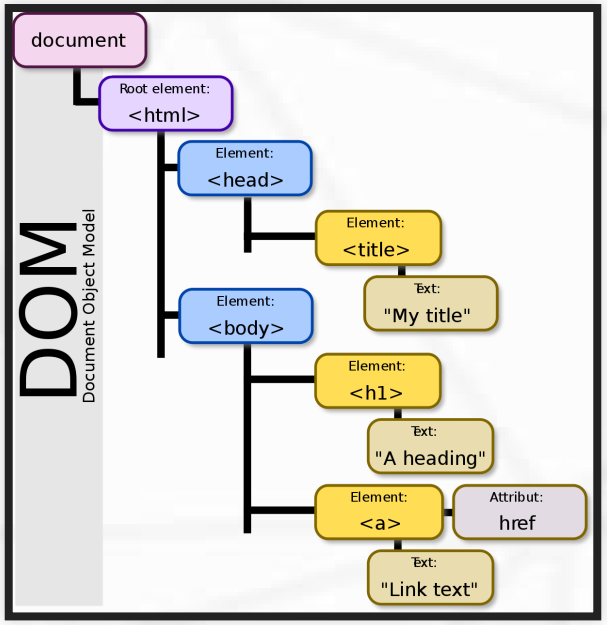
HISTORIA

- 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 Microsytem lo cambiaron a JavaScript
- Aparece por primera en Netscape Navigador 2.0
- Es usado por todos los navegadores y webview móbiles actuales.

HISTORIA

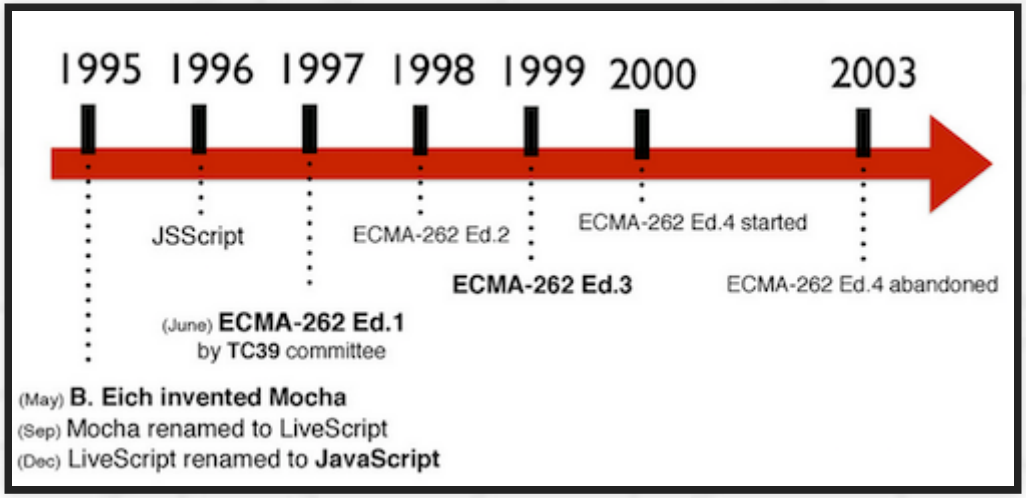
En 1997 se crea un comité (TC39) para estandarizar Javascript por la European Computer Manufacturer's Association, ECMA

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


DOM
Document Object Model
document
Root element:
<html>
Element:
<head>
Element:
<title>
Text:
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribut:
href
Text:
"Link text"

HISTORIA

A partir de ese momento se genea los primeros estandares.



HISTORIA

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

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

HISTORIA

En 2015 se establecio la última versión ECMASCRIPT 6

Soporte Actual de EMASCRIPT 6

ESTANDAR DE JAVASCRIPT

Viene definido por el ECMA Internacional

Standard ECMA-262

¿QUE ES JAVASCRIPT?

- JavaScript es un lenguaje de programación de **alto nivel interpretado**. Sin embargo posee ciertas características que lo distinguen del resto de lenguajes
- 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 las etiqueta javascript en el propio html

```
<script type="text/javascript">
        console.log("Hola Mundo");
</script>
```

- Importando el script

```
<script type="text/javascript" src="/js/codigo.js"></script>
```

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 interprtean por el navegador

```
// ->Doble barra para comenta una línea.
/* */ ->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

TIPOS DE DATOS

TIPOS DE DATOS

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

Los tres tipos de datos primitivos son:

- Números
- Cadena de texto
- Valores Booleanos
- Null
- Undefined
- Symbol(ECMASCRIPT 6)
- Object

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.
- Este forma es el llamado double en los lenguajes Java o C++.

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.

CADENAS DE ESCAPE

Secuencia de escape	Carácter
\'	Comilla simple
\"	Comilla doble
\\	Barra inversa
\0	Carácter nulo
\a	Alerta
\b	Backspace
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical

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.

TIPO NULL

El valor null es un literal de Javasript que representa valor nulo o vacío

TIPO UNDEFINED

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

TIPO SYMBOL(E6)

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

OBJETOS

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 sintáxis 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.

OBJETOS

Los Objetos tienen dos tipos de propiedades, los cuales tienen ciertos atributos:

- Propiedades de datos
- Propiedades de acceso

OBJETOS

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

Atributos de una propiedad de datos

Atributo	Tipo	Descripción	Valor por defecto
[[Value]]	Cualquier tipo JavaScript	El valor obtenido mediante un acceso get a la propiedad.	undefined
[[Writable]]	Boolean	Si es `false`, el [[Value]] de la propiedad no puede ser cambiado.	false
[[Enumerable]]	Boolean	Si es `true`, la propiedad será enumerada en ciclos for...in. Consultar también Enumerabilidad y pertenencia de las propiedades	false
[[Configurable]]	Boolean	Si es `false`, la propiedad no puede ser eliminada y otros atributos que no sean [[Value]] y [[Writable]] no pueden ser cambiados.	false

OBJETOS

Asocia una clave con una o más funciones de acceso (get y set) para obtener o almacenar un valor y tiene los siguientes atributos:

Atributos de una propiedad de acceso

Atributo	Tipo	Descripción	Valor por defecto
[[Get]]	Función, objeto o undefined	La función es llamada con una lista de argumentos vacía y devuelve el valor de la propiedad cada vez que un acceso get al valor es realizado. Consultar también `get`.	undefined
[[Set]]	Function object or undefined	La función es llamada con un argumento que contiene la variable asignada y es ejecutada siempre que una propiedad especifica se intenta cambiar. Consultar también `set`.	undefined
[[Enumerable]]	Boolean	Si el valor es `true`, la propiedad será enumerada en bucles for...in.	false
[[Configurable]]	Boolean	Si el valor es `false`, la propiedad no podrá ser eliminada y tampoco ser modificada a una propiedad de datos.	false

OBJETOS

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

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

OBJETOS ESPECIALES DATE

Uno de los granes 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...
```

VARIABLES

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.

DECLARACION DE UNA VARIABLE

Dependiendo del contexto donde se vaya a ejecutar, var o let (ES6)

let permite declara variables limitando su alcance al bloque, declaración o expresión donde se esté usando

Var define una variable global o local en una función sin importar el ámbito del bloque

DECLARACION DE UNA VARIABLE

JS | Result

EDIT ON

```
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4; // El alcance es dentro del bloque if
  var b = 1; // El alcance es dentro de la función

  console.log(a);  // 4
  console.log(b);  // 1
}
```

DECLARACION DE UNA VARIABLE

Ademas let no nos permite definir atributos de la clase para ello se usa var

JS Result EDIT ON

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
console.log(this.y); // undefined
```

DECLARACIÓN CONSTANTES CONST (ES6)

Las constantes presentan un ámbito de bloque tal y como lo hacen las variables definidas usando la instrucción let, con la particularidad de que el valor de una constante no puede cambiarse a través de la reasignación, y no se puede redeclarar.

DECLARACIÓN CONSTANTES CONST (ES6)

JS | Result

EDIT ON

```
const MY_FAV = 7;

// lanzara un error: Unkeught TypeError: Asignación a variable
constante.
MY_FAV = 20;

// imprimira 7
console.log('my favorite number is: ' + MY_FAV);
```

INICIALIZACIÓN DE VARIABLES.

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

- Asignación direca de un valor concreto.
- Asignación indirecta a raves de un calculo en el que se implican a otras variables o constantes.
- Asignación a traves de la solicitud del valor al usuario del programa.

INICIALIZACIÓN DE VARIABLES.

```
// Asignación directa.
let variable1=39;

// Asignación indirecta a través de un cálculo.
let variable2=variable1 + 20;

// Asignación a traves de la solicitud de un usuario.
let variable3=prompt('Introduce un valor');
```

OPERADORES

JavaScript tiene los siguientes tipos de operadores

OPERADORES ARITMÉTICOS

Permite realizar cálculos elementales entre variables numéricas.

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

OPERADORES ARITMÉTICOS

Otros operadores Aritméticos Especiales

- - Negación unaria

```
-"3" -> -3 //Intenta convertir un número al operand
//y devuelve su forma negativa.
```

- + Unario positivo

```
+"3" -> +3 //Intenta convertir un número al operand
//y devuelve su forma positiva.
```

- ** Exponente

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

OPERADORES LÓGICOS

Combinan diferentes expresines 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 ue se encuentra a la izquierda del operador

- += Suma y asigna
- -= Resta y asigna
- *= Multiplica y asigna
- /= Divide y asigna
- %= Módulo y asigna
- **= Asignación de exponencación
- <<=A signación de desplazamiento a la izquierda /li>
- >>= Asigación de desplazamiento a la derecha/li>
- >>>= Asignación sin signo de desplazamiento a la derecha
- &= Asignación AND
- ^= Asignación XOR
- |= Asignación OR

OPERADORES DE COMPARACION

Permite comparar todo tipo de variables y devuelven un valor booleano

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

OPERADORES CONDICIONALES

Permiten indicar al navegador que ejecute una acción en concreto después de evaluar una expresión.

? :

```
age > 18 ? alert("Eres mayor de edad") : alert ("Eres menor de
```

OPERADORES CADENAS DE CARÁCTERES

Operador de concatenación es el único operador que permite sobrecarga +

```
console.log("mi "+"cadena");
```

OPERADOR COMA

(,) simplemente evalúa ambos oprando 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

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

Tipo de operador	operadores individuales
miembro	`. []`
llamar / crear instancia	`() new`
negación / incremento	`! ~ - + ++ -- typeof void delete`
multiplicación / división	`* / %`
adición / sustracción	`+ -`
desplazamiento binario	`<< >> >>>`
relación	`< <= > >= in instanceof`
igualdad	`== != === !==`
AND binario	`&`
XOR binario	`^`
OR binario	`|`
AND lógico	`&&`
OR lógico	`||`
condicional	`?:`
asignación	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
coma	`,`

SENTENCIAS DE CONTROL

SENTENCIAS DE CONTROL

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

Las sentencias condiciones JavaScript son:

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
}
```

SENTENCIAS IF

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

```
if (a > 2 || a < -2) {
    result = 'a is not between -2 and 2';
} else if (a === 0 && b === 0) {
    result = 'both a and b are zeros';
} else if (a === b) {
    result = 'a and b are equal';
} else {
    result = 'I give up';
}
```

```
var result = (a === 1) ? "a is one" : "a is not one";
```

SENTENCIAS CONDICIONALES

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

Es necesario emplear la instrucción reak para no seguir con la ejecución del siguiente caso (y salir del switch). La cláusual 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;;
}
```

SENTENCIAS REPETITIVAS

Existen 5 tipus de bucles en JavaScript:

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

SENTENCIA WHILE

Se ejecutarán las instrucciones de dentro del bucle while hasta que la condición valga a falso. En dicho momento finalizará el bucle

El Bucle While tiene la siguiente estructura:

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

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

SENTENCIA 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)
```

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

SENTENCIA 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

SENTENCIA FOR

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

SENTENCIA FOR-IN

Se utiliza para recorrer los elementos de un objeto

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

// Visit non-inherited enumerable keys
Object.keys(obj).forEach(function(key) {
    console.log(key);
});
```

SENTENCIA 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.

```
let iterable = [10, 20, 30];

for (let value of iterable) {
  value += 1;
  console.log(value);
}
```

FUNCIONES

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.

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

FUNCIÓN EVAL()

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

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

```
let input=prompt("Intoruzca una operación numérica");
let 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

```
isFinite(Infinity)
isFinite(-Infinity)
isFinite(12)
isFinite(1e308)
isFinite(1e309)
```

FUNCIÓN ISNAN()

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

```
isNaN(NaN)
isNaN(123)
isNaN(1.23)
isNaN(parseInt('abc123'))
```

FUNCIÓN NUMBER()

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

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

let n =
Number(x1) + "<br>" +
Number(x2) + "<br>" +
Number(x3) + "<br>" +
Number(x4) + "<br>" +
Number(x5);
```

FUNCIÓN STRING()

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

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

let res =
String(x1) + "<br>" +
String(x2) + "<br>" +
String(x3) + "<br>" +
String(x4) + "<br>" +
String(x5);
```

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.

```
let input=prompt("Introduce un valor: ");
let 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

```
let uri = "my test.asp?name=ståle&car=saab";
let res = encodeURI(uri);
```

FUNCIÓN DECODEURI()

Decodificación de la URI

```
let uri = "my test.asp?name=ståle&car=saab";
let enc = encodeURI(uri);
let dec = decodeURI(enc);
let res = "Encoded URI: " + enc + "<br>" + "Decoded URI: " + d
```

FUNCIONES DEL USUARIO

- Es posible crear funciones personalizadas diferentes a la funciones predefinidas en el lenguaje.
- Con estas funciones se pueden realizar las tareas que queramos.
- Una tarea se realiza mediante un grupo de instrucciones relacionadas a las cuales debemos dar un nombre
- Cuando queramos ejecutar este grupo de instrucciones en cualquier otra parte de la aplicación, simplemente debemos utilizar el nombre que le hayamos dado a la función

FUNCIONES DEL USUARIO

La definición de una función consta de 5 partes

- La palabra clave function
- El nombre de la función
- Los argumentos utilizados
- El grupo de instrucciones
- La palabra clave return

FUNCIONES DE USUARIO

El nombre de la función debe cumplir los siguientes requisitos.

- El nombre de la función se situa al inicio de la definición y antes del paréntesis que contiene los posibles argumentos
- Deben usarse sólo letras, números o el carácter de subrayado.
- Debe ser unico en el código JavaScript de la página web
- No pueden empezar por número.
- No puede ser una de las palabras reservadas del lenguaje.
- Tampoco puede ser una palabra clave del lenguaje.

FUNCIONES DE USUARIO

Ejemplo de funciones

```
function aplicarIva(producto){
        CONST IVA = 1.18;
        let productoConIva = producto * IVA;
        alert("Aplicado el IVA es -> " + productoConIva);
}
```

FUNCIONES DE USUARIO

En EcmaScript 6 las funciones genera un Scope Local las variables definidas en una función sólo son accesibles dentro de esa función

```
function nombre(){
        let nombre="Pepe";
        alert(nombre);
}
alert(nombre);
```

FUNCIONES DE USUARIO

ECMAScript 6 Introduce también el concepto de **Closure()**

Un closure es una función que es libre de varaibles, esto quiere de cir que las variables de la función padre funcionan, pero el closure no tiene varaibles propias.

```
function padre() {
        var a = 1;
        function closure() {
        console.log(a);
        }
        closure();
}
padre();
```

FUNCIONES DE USUARIO

ECMAScript 6 otro ejemplo de closure

```
function crearFuncion() {
  var a = 1;
  function closure() {
    console.log(a);
  }
  return closure;
}
var miFuncion = crearFuncion();
miFuncion();
```

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros por defecto.

Nos permite definir parámetros por defecto con la única condición que aquellos que se definan con parámetros opcionales son obligatorios.

```
//ES5
function(valor){
    valor = valor || "foo";
    console.log(valor);
}
//ES6
function(valor="foo"){
    console.log(valor);
}
```

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros por defecto.

JavaScript cuando se crea una función crea un objeto **Arguments** donde se recogen los parámetros de la función.

```
 function operacion(a,b){
        console.log(arguments);
}
operacion(5,6);
```

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros por defecto.

Ahora los argumentos con parámetros opcionales no actuan, por ejemplo.

```
function operacion(a=1, b=2){
    console.log(arguments);
}
operacion();
```

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros REST(Parámetros sin nombre)

- El prámetros "rest" es indicado con tres puntos (...) seguido del nombre que le asignaremos a dicho parámetro
- Ese parámetro se convierte en un array que contiene el resto de los parámetros pasados a la función.
- De ahí se estable el nombre de parámetros **"REST"**

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros REST(Parámetros sin nombre)

```
function agregar_cliente(arr_clientes, ...clientes ){
    console.log(arguments);
    for( let i = 0; i< clientes.length; i++){
        arr_clientes.push(clientes[i]);
    }
    return arr_clientes;
}
let arr_clientes = ["Pepe"];
let arr_clientes2= agregar_cliente(arr_clientes, "Antonio", "M
console.log(arr_clientes2);
```

FUNCIONES DE USUARIO

ECMAScript 6 Parámetros REST(Parámetros sin nombre)

Restricciones a tener en cuenta con el parámetro REST

- Sólo puede existir un parámetro rest en la función.
- El parametro rest debe ir siempre como último parámetro

FUNCIONES DE USUARIO

EMCAScript 6 Operador SPREAD

La diferencia con el parámetro Rest, es que mientras Rest permite especificar argumentos independientes que serán combinados en un array.

El operador SPREAD permite especificar un array que será separado y cada item enviado será un argumento independiente a la función.

FUNCIONES DE USUARIO

Operador SPREAT

```
let numeros=[1,5,10,20,100,234];
//EcmaScript 5
let max= Math.max.apply(Math, numeros);
console.log(max);
//EcmaScript 6
let max1 = Math.max( ...numeros );
console.log(max1);
```

FUNCIONES DE USUARIO

Diferencias entre REST y SPREAD

REST

Junta los elementos en un arreglo.

SPREAD

Separa los elementos como si fuerna enviados de forma separada.

FUNCIONES DE USUARIO

En versiones anteriores a ECMASCRIPT 5 las funciones tenían un doble proposito en función de la forma invocarlas

- Utilizando la palabra reservada new
- Sin utilizar la palabra reservada new

FUNCIONES DE USUARIO

Cuando realizamos la llamada con **NEW** el valor THIS dentro de la función es un nuevo objeto y es retornado

Cuando realizamos la llamada de la función, siempre espramos el retorno de algún valor que puede ser

- Objeto
- Undefined
- Null

FUNCIONES DE USUARIO

```
function Pelicula(titulo){
        this.titulo = titulo;
}
var pelicula = new Pelicula("El señor de los anillos"); //This
console.log(pelicula);
var pelicula1 = Pelicula("El señor de los anillos"); //This ap
console.log(pelicula1);
```

FUNCIONES DE USUARIO

Por ejemplo

```
function Pelicula(titulo){
  if (this instanceof Pelicula){
        this.titulo = titulo;
  } else{
        throw new Error('Debe utilizarse con new');
  }
  this.titulo = titulo;
}
let pelicula = Pelicula("El señor de los anillos");
let pelicula1 = new Pelicula("El señor de los anillos");
let pelicula2 = Pelicula.call( pelicula1 , "StarWars");
```

FUNCIONES DE USUARIO

Con ECMAScript6 aparece una meta propiedad, denominada **NEW.TARGET**

Es una propiedad de un no-objeto que provee información adicional relacionada con su procedencia (como el new)

Cuando el constructor de la función es llamda, new.target se llena con el operador new.

si la función Call() es ejecutada, new.target no estará definida ya que no se ejecutó el constructor.

FUNCIONES DE USUARIO

Ejemplo

```
function Pelicula(titulo){
    if (typeof new.target !=="undefined"){
            this.titulo = titulo;
    }else{
            throw new Error('Debe ser util
    }
    this.titulo = titulo ;
}
let pelicula = new Pelicula("El señor de l
let pelicula1 = Pelicula.call( pelicula, "
```

FUNCIONES DE USUARIO

Este tipo de funciones tienen un sintaxis variada que depende de lo que quieras realizar.

Pero nomalmente tiene la misma estructura:

- Función con argumentos
- Seguido de una función arrow =>
- Y por último el cuerpo de la función.

FUNCIONES DE USUARIO

Para que se pueden utilizar.

- Menos código que es más simple de interpretar.
- No hay un nuevo "this" dentro de las funciones.
- No hay constructores, no tiene new.
- No puedes cambiar el valor del "this", aunque usemos call(), apply() o bind()
- No dispone del objeto arguments

FUNCIONES DE USUARIO

En definitiva, son funciones definidas con una nueva sintaxis que usa una flecha => pero se comportan de una forma muy diferente a las funciones normales en ECMAScript 5

FUNCIONES DE USUARIO

Estas funciones tiene 6 características:

- 1. NO hay creación de: **this, super, arguments y new.target**
- 2. NO puede ser llamado con **new**
- 3. NO tienen prototipos **prototype**
- 4. NO puede cambiar **this**
- 5. NO existe el objeto **arguments**
- 6. NO pueden tener nombres de parámetros **duplicados**

FUNCIONES DE USUARIO

Ejemplo de funciones flecha.

```
let funcion1= function(valor){
        return valor;
}
let funcion2 = valor => valor;
console.log(funcion1("hola"));
console.log(funcion2("hola"));
```

FUNCIONES DE USUARIO

Ejemplo de funciones flecha.

```
let multiplicar1 = function (valor1, valor2){
        return valor1* valor2;
}
let multiplicar2 = (valor1, valor2) => valor1*valor2;
console.log(multiplicar1(10*10));
console.log(multiplicar2(10*10));
```

FUNCIONES DE USUARIO

Ejemplo de funciones flecha devolución de un string

```
let saludarPersona1 = function(nombre){
                let salida="Hola, " + nombre;
                return salida;
}
let saludarPersona2 = nombre =>{
                let salida = `Hola ${nombre}`;
                return salida;
}
console.log(saludarPersona1("Javi1"));
console.log(saludarPersona2("Javi2"));
```

FUNCIONES DE USUARIO

Ejemplo como devolver un objeto literal

```
let obtenerPelicula1=function(id){
        return {
                id:id,
                titulo: "El señor de los anillos"
                }
}
let obtenerPelicula2 = id => ({id:id, titulo:"StarWars"});
console.log(obtenerPelicula1(1));
console.log(obtenerPelicula2(1));
```

FUNCIONES DE USUARIO

Funciones anónimas.

Las funciones anónimas son funciones que se ejecutan en el momento de crearse

Estas funciones se utilizan para:

- Cuando queremos pasar como parámetro una función sencilla.
- Cuando intentamos evitar a toda costa el uso de variables globales.
- Además son funciones que pueden ser autoejecutas colocándolas entre ()

FUNCIONES DE USUARIO

```
//ECMAScript 5
let saludo1=function(nombre){
                return "Hola " + nombre;
}("Javier");
//ECMAScript 6
let saludo2=( nombre => `Hola ${nombre}`)("Xavi");
console.log(saludo1);
console.log(saludo2);
```

FUNCIONES DE USUARIO

Utilización de las funciones flecha con funciones predefinidas.

```
let arrayNumeros = [2,3,89,54,66];
//EcmaScript 5
let ordenando1 = arrayNumeros.sort(function(a,b){
        return a-b;
});
//EcmaScript 6
let ordenando2 = arrayNumeros.sort((a,b)=>a-b);
console.log(ordenando1);
console.log(ordenando2);
```

ESTRUCTURA DE DATOS

En javaScript podemos disponer de diferentes tipos de estructuras de datos:

- Arrays
- Maps
- Set

ARRAYS

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 numerica en el array.

ARRAYS

Para poder declarar el **Array** al igual que las variables es necesario declararlo antes de poder usarlo.

La declaración consta de 6 partes.

- La palabra clave let o const
- El nombre del array
- El operador de asignación
- La palabra clave para la creación de objetos **new**
- El constructor **Array**
- El parentesis final.

ARRAYS

Para la declaración de los Arrays podemos realizar de la siguiente forma:

```
let [nombre del array] = new Array();
//O iniciando con un número de elementos.
let [nombre del array] = new Array(número de elementos);
```

ARRAYS

Para inicializar el **Array** se realizará de la siguiente forma.

```
nombredelarray[índice]= valorElemento:
```

Es posible declarar e inicializar simultánemamente mediante la escritura de elementos dentro del constructor

```
let colores = new Array('rojo','azul','verdes');
```

ARRAYS

Podemos recorrer un array utilizando bucles, por ejemplo:

```
let numeros = new Array();
for(let i=0; i<10; i++){
        numeros[i] ="Numero:"+ i;
}
// O podemos iterar medainte forEach
numeros.forEach(function(valor){
        console.log(valor);
});
```

ARRAYS

Propiedades de los Arrays.

Los Arrays tienen dos propiedades:

- Array.Length
- Array.prototype

ARRAYS

La propiedad **Array.Length** nos devuelve el tamaño el array

```
for (let i=0; i < numeros.length;i++){
        console.log(numeros[i]);
}
```

ARRAYS

La propiedad **Array.prototype** nos permite añadir y modificar propiedades y métodos al objeto Array

ARRAYS

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.

ARRAYS

El método **PUSH**

Añade nuevos elementos al array devuelve la nueva longitud del array

```
let resultado = colores.push('gris');
console.log(resultado);
```

ARRAYS

El método **CONCAT**

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

```
let equipos_a = new Array('Valencia','Barça','Real Madrid');
let equipos_b = new Array('Hercules','Zaragoza','Valladolid');
let equipos_copa = equipos_a.concat(equipos_b);
console.log(equipos_copa);
```

ARRAYS

El método **JOIN**

Concatena los elementos de un array en una sóla cadena separada pro un caracter opcional

```
let nombres = new Array("Manuel","Antonio","Pepe");
console.log(nombres.join('-'));
```

ARRAYS

El método **REVERSE**

Invierte el orden de los elementos de un array

```
console.log(numeros.reverse());
```

ARRAYS

El método **UNSHIFT**

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

```
let valor_nombres = nombres.unshift("Jesús");
console.log(valor_nombres);
```

ARRAYS

El método **SHIFT**

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

```
let valor_resto = nombres.shift();
console.log(valor_resto);
```

ARRYAS

El método POP

Elimina el último lemento de un array

```
console.log(nombres.pop());
```

ARRAYS

El método **SLICE**

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

```
let numeros = new Array(1,2,3,4,5,6,7,8,9,10);
let primerosTresNumeros = numeros.slice(0,3);
let ultimosCuatroNumeros = numeros.slice (-4);
console.log(primerosTresNumeros);
console.log(ultimosCuatroNumeros);
```

ARRAYS

El método **SORT**

Ordena alfabéticamente los elementos de un array.

```
let nombres = new Array("Manuel", "Antonio", "Sergio","Ricardo
nombres.sort();
console.log(nombres);
```

ARRAYS

El método **SPLICE**

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

```
let peliculas = new Array("Superman","Superman 3");
peliculas.splice(1,0,"Superman 2");
console.log(peliculas);
```

ARRAYS

El método **SPLICE**

Dispone de los siguientes elementos

```
arrayObj.splice(start, deleteCount , [item1[,item2[,...[itemN]
```

ARRAYS

Elementos de SPLICE

- arrayObj -> Objeto Array
- start -> Obligatorio, Ubicación basada en cero de la matriz.
- deleteCount -> Obligatorio, número de elementos que se va a quitar.
- itemN -> Opciones, Elementos que se van a insertar en la matriz en lugar de los elementos eliminados.

MAP Y SET

MAP

El objeto Map almacena pares clave/valor. Cualquier valor (tanto objetos como valores primitivos) pueden ser usados como clave o valor.

Documentación

```
var myMap = new Map();
miMap.set('clave', 'valor');
var objeto = {objeto:'dePruebas'}
miMap.set(objeto , () => 'esto es una función');
miMap.get(objeto)  // devuelve la función asociada:  () => 'es
```

MAP

Métodos heredados para todas las instacias Map.

```
Map.prototype.clear() // borra todo el contenido
Map.prototype.delete(item) // borra un item
Map.prototype.entries()
Map.prototype.forEach()
Map.prototype.get()
Map.prototype.has()
Map.prototype.keys()
Map.prototype.set()
```

SET

El objeto Set te permite almacenar valores únicos de cualquier tipo, incluso valores primitivos u objetos de referencia.

Diferencia clave con las arrays: Los Sets almacenan solo una copia de cada objeto

Documentación

SET

```
var miSet = new Set();
var a = 2;
miSet.add(a);
miSet.add(a);
// esta segunda copia no se guarda
//ya que el objeto a ya estaba contenido en el set
```

Documentación

SET

```
Set.prototype.add()
Set.prototype.clear()
Set.prototype.delete()
Set.prototype.entries()
Set.prototype.forEach()
Set.prototype.has()
Set.prototype.values()
```

Documentación

SET

```
const mySet = new Set();
mySet.add(1);
mySet.add(5);
mySet.add('some text');
mySet.has(1); // true
mySet.has(3); // false, 3 no ha sido añadido al Set
mySet.has(5);                 // true
mySet.delete(5); // Elimina 5 del Set
mySet.has(5);    // false, 5 fue eliminado
```

Documentación