UNIVERSIDAD DE GRANADA

DECSAI

Tecnologías Web

Grado en Ingeniería Informática

Tema 4 – Programación en el lado del cliente
El lenguaje JavaScript

© Javier Martínez Baena
jbaena@ugr.es
Departamento de Ciencias de la Computación e Inteligencia Artificial
http://decsai.ugr.es

UNIVERSIDAD DE GRANADA

DECSAI

Tecnologías Web

3º Grado en Ingeniería Informática

Programación en el lado del cliente - JavaScript

El lenguaje JavaScript: ECMAScript-262
Historia

JavaScript es creado en 1995 por Brendam Eich (que trabajaba para Netscape)

No tiene nada que ver con Java, salvo el nombre

Se comienza a usar para hacer algunos efectos visuales en páginas web y para validar formularios en el lado del cliente

En 1997, ECMA (European Computer Manufacturers Association) se hace cargo de su estandarización: ECMAScript (ECMA-262) [ISO/IEC-16262]

Ediciones de ECMA-262 (ISO/IEC 16262)
1.- 1997
2.- 1998
3.- 1999
4.- (abandonada)
5.- 2009
5.1.- 2011
6.- 2015
7.- en progreso

Compatibilidad con navegadores: http://kangax.github.io/compat-table

Zakas. "Professional JavaScript for Web developers". Wiley-Wrox. 2012

El lenguaje JavaScript: ECMAScript-262
Historia

Sobre las versiones ...

- ECMAScript-262. Desarrollado por Ecma International (estándard)
- JavaScript. Implementación de Mozilla Foundation del estándard

Año	JavaScript	ECMA
1996	1.0	
1997	1.2	1
1998	1.3	2
2000	1.5	3
2009	1.8.1	5
2011	1.8.5	5.1
2015	1.8.5+	6
?		7

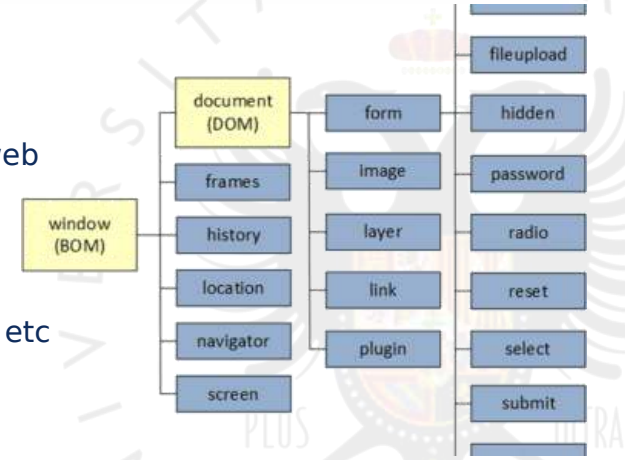
Motor	SpiderMonkey	V8	JavaScriptCore	Chakra
Navegador	Firefox	Chrome	Safari	IE

JavaScript está compuesto por ...

- ECMAScript-262. Define el núcleo del lenguaje de programación. No tiene nada que le permita interactuar con el entorno
- BOM (Browser Object Model). Es una API que permite interactuar con el navegador.
  - DOM (Document Object Model). Es una API que provee funcionalidad para manipular el contenido de una página web

Para qué se suele/puede usar:

- Validación de formularios
- Autocompletado de cajas de texto
- Modificación dinámica de páginas web
- Descarga de trabajo del servidor
- Interacción con servidores
- Mejora de la interfaz de usuario
- Interactividad de las páginas web
- Aplicaciones gráficas: animaciones, etc
- ...

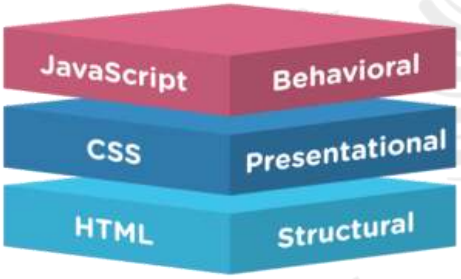


Zakas. "Professional JavaScript for Web developers". Wiley-Wrox. 2012
http://archive.cnx.org/contents/3ae9bdd1-2506-4c72-b807-46803e901812@7/java4330-html-and-css-fundamentals

Diseño progresivo (progressive enhacement)

Es una filosofía de programación

- Separación entre capas (estructura, presentación, comportamiento).
- Accesibilidad. Si una capa superior falla, las de abajo siguen ofreciendo el mismo contenido.
- Eficiencia. La separación en ficheros permite que funcione mejor la caché de los navegadores.
- Reusabilidad. Partes de una web reusables (separación de ficheros).



T. Wright "Learning JavaScript. A hands-on guide to the fundamentals of modern JavaScript". Addison-Wesley. 2013
http://blog.teamtreehouse.com/progressive-enhancement-past-present-future

Características principales del lenguaje

- Tipificación débil
- Variables globales
- Recolección de basura

- Clausuras

- Paradigmas
  - Funcional: construcción de programas como composición de funciones
    - First-class functions
      - Las funciones pueden pasarse como argumento
      - Pueden devolverse por parte de otras funciones
      - Pueden asignarse a objetos
      - Permite Lambda funciones (funciones anónimas)
  - Orientado a objetos
    - Existen objetos con
      - Propiedades
      - Métodos
    - No existen clases
    - Basado en instancias
    - Herencia por prototipado

Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts. After all, how can you build something good out of bad parts?

Javascript is built on some very good ideas and a few very bad ones.

The very good ideas include functions, loose typing, dynamic objects, and a expressive object literal notation.

The **bad ideas** include a programming model based on **global variables**.

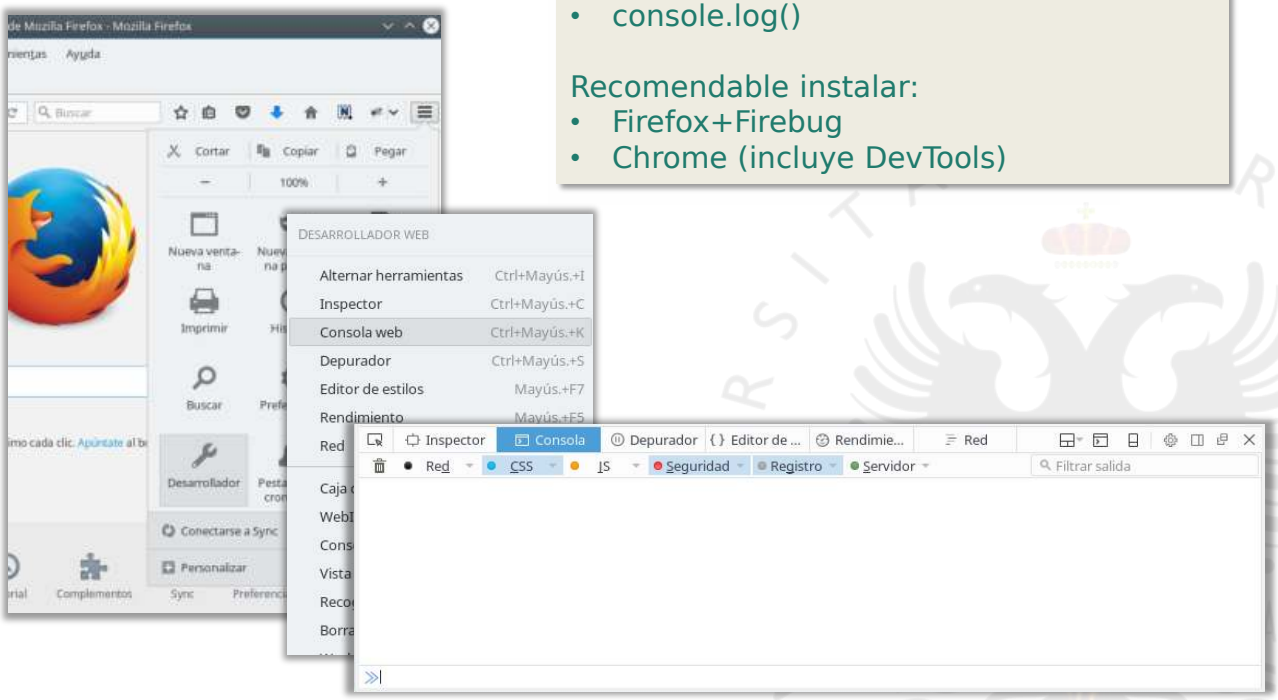
Douglas Crockford. **Javascript. The good parts**. *O'Reilly, 2008*

JavaScript
Ejecución de código JavaScript

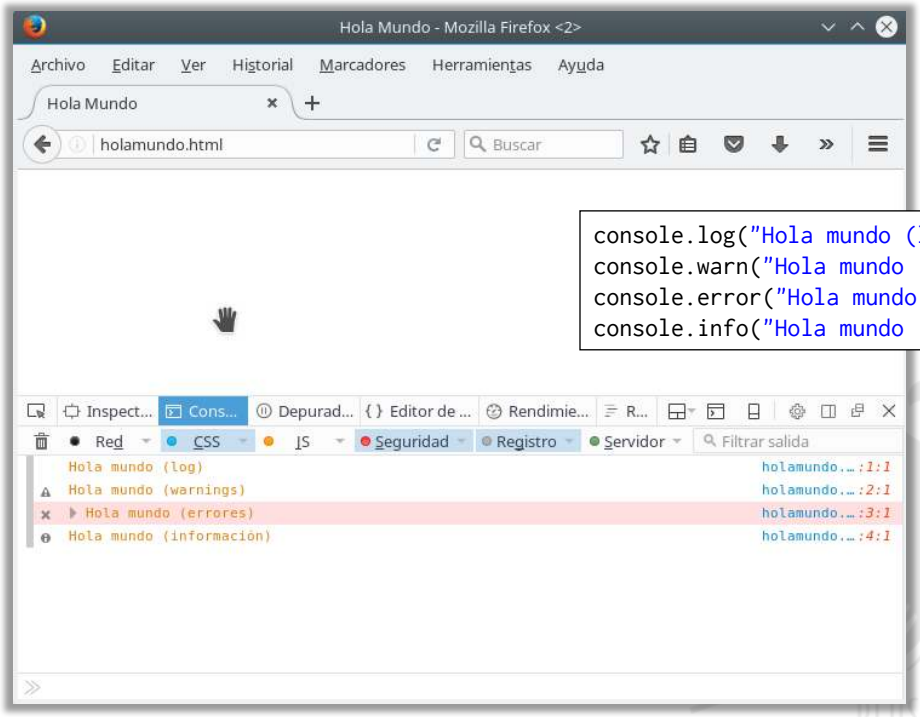
- Ejecución en el navegador
- Ejecución mediante intérprete (Rhino, Node.js, ...)

Para mostrar mensajes en consola:
- console.log()

Recomendable instalar:
- Firefox+Firebug
- Chrome (incluye DevTools)



JavaScript
Consola de JavaScript




```
console.log("Hola mundo (log)")
console.warn("Hola mundo (warnings)")
console.error("Hola mundo (errores)")
console.info("Hola mundo (información)")
```

https://developer.mozilla.org/es/docs/Web/API/Console

JS embebido en la página web

```
<script>
 ...
</script>
```

JS en fichero externo

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

doc.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hola Mundo</title>
  </head>
  <body>
    <script>
    console.log("Hola mundo")
    </script>
  </body>
</html>
```

doc.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hola Mundo</title>
  </head>
  <body>
    <script src="holamundo.js"></script>
  </body>
</html>
```

holamundo.js

```
console.log("Hola mundo")
```


UNIVERSIDAD DE GRANADA

Tecnologías Web

3º Grado en Ingeniería Informática

Programación en el lado del cliente - JavaScript

DECSAI

Cuestiones básicas sobre el lenguaje

- Es sensible a mayúsculas/minúsculas
- Se suelen ignorar los espacios en blanco y saltos de línea entre tokens
- El sangrado de código es libre (se recomienda usar buen estilo)

- Los comentarios son de la forma (igual que C++):

```
/* … */
//
```

- Los identificadores comienzan por una letra, subrayado (_) o dólar ($) seguidos de letras, dígitos, _, $
  Las letras pueden ser cualquier carácter Unicode (se recomienda alfabeto inglés)

- Las instrucciones se escriben una por línea y acabadas en ;
  - Cerrar con ; no es obligatorio
  - JS interpreta que un salto de línea equivale a un ; (en caso de omisión) si el carácter que viene a continuación no puede ser interpretado como continuación de la línea anterior.

```
var x = 2 + f
(s).toString()
```

  - Consejo: cerrar siempre con ;

Capítulo 2. Flanagan. JavaScript: The definitive guide (6ed). O'Reilly. 2011

Tipos de datos

- **Tipos primitivos (*son inmutables*)**
  - Números (valores: enteros y reales)
  - Booleanos (valores: `true`, `false`)
  - Cadenas
  - Tipo Null (único valor: `null`)
    - Valor especial que significa "sin valor"
  - Tipo Undefined (único valor: `undefined`)
    - Variable declarada pero sin asignar valor
    - Argumento que no se ha pasado a una función
    - Función que no devuelve nada

- **Objetos (*son mutables*)**
  - Todo lo que no sea un tipo primitivo
  - Un objeto es una colección de propiedades (nombre+valor)
  - Las funciones son objetos
  - Objetos intrínsecos:
    - Array, Date, Math, RegExp
    - Boolean, Number, String (diferente de tipos primitivos)
  - Son denominados "tipos referencia"

Variables

- Las variables no tienen tipo
- Se les puede asignar un valor de cualquier tipo en cualquier momento
- Por defecto se asumen globales, para que sean locales se deben declarar con la palabra reservada `var`

- **Recomendación: declarar SIEMPRE las variables con `var`**

```
a=1;

var b=2;

var c;

B=4;

b="texto";
```

B	4
a	1
b	"texto"
c	*undefined*

Sobre las variables y su ámbito

- Toda variable ha de ser declarada
- Las variables se inicializan a `undefined` en el momento de ser creadas

- Si no se usa `var` → variable global

- Si se usa `var` (recomendado), el ámbito es local a la **función** (o global si la declaración es externa a una función)
- No existe el ámbito local de bloque `{ }`

- Una variable puede ser declarada múltiples veces

- JavaScript eleva las declaraciones de variables al comienzo del ámbito (***declaration hoisting***):
  - Las variables se pueden declarar en cualquier parte
  - JavaScript lleva la declaración al comienzo del ámbito
  - Las inicializaciones no son "elevadas" al comienzo del ámbito

- Una variable puede declararse después de su uso (consecuencia del hoisting)

Recomendación: declarar todas las variables con `var` al comienzo de la función (o del script)

JavaScript. Elementos básicos
Ámbito de las variables

Toda variable ha de ser declarada
Las variables se inicializan a `undefined` en el momento de ser creadas

```
var a = 3;
b = 4;
var c;
var e = a * b * c * d;
```

```
var a;
var b = 3*a;
console.log(a);
console.log(b);
```

Si no se usa `var` → variable global

```
function f() {
  b=6;
}
f();
console.log(b);
```

```
a=2;
function f() {
  a=3;
}
f();
console.log(a);
```

Si se usa `var` → ámbito local a la función
No existe el ámbito local de bloque `{ }`

```
a=2;
function f() {
  var a=3;
}
f();
console.log(a);
```

```
function f() {
  var a=2;
  if (a>1) {
    var b=3;
  }
  console.log(b);
}
f();
```


JavaScript. Elementos básicos
Ámbito de las variables - hoisting

Una variable puede ser declarada múltiples veces

```
var a=3;
var a=6;
var a=8;
console.log(a);
```

Declaration hoisting

- Las variables se pueden declarar en cualquier parte
- JavaScript lleva la declaración al comienzo del ámbito
- Las inicializaciones no son “elevadas” al comienzo del ámbito

```
var b = a*3;
var a = 4;
console.log(b);
```

```
var a;
var b = a*3;
a = 4;
console.log(b);
```

```
var a=4;
function f() {
  console.log(a);
  var a=6;
  console.log(a);
}
f();
```

```
function f(n) {
  var i=1;
  if (n>0) {
    var j=0;
    for(var k=0; k<5; k++) {
      console.log(k+i);
    }
    console.log(k);
  }
  console.log(j);
}
f(2);
f(-2);
```

ECMAScript 6 y el ámbito

- Existe la declaración mediante `let`
  - Restringe declaraciones a ámbito de bloque
  - No permite duplicar declaraciones dentro de un mismo bloque
  - No se hace hoisting
    - No se puede usar hasta haber sido declarada
- Existe la declaración de constantes (`const`)
  - Con ámbito de bloque

```
var a = 1;
{
  let a = 2;
  console.log( a );
}
console.log( a );
```

```
function f() {
  console.log(a);
  let a = 1;
}
f();
```

```
for (let i = 0; i<5; i++) {
  console.log(i);
}
console.log(i);
```

```
const a=2;
var b=a*3;
a=3;
```

https://davidwalsh.name/for-and-against-let
http://www.genbetadev.com/javascript/ecmascript-6-nuevas-variables-en-javascript-let-y-const

Datos de tipo numérico

- Pueden ser enteros o reales (no se distingue) y se representan con IEEE-754 de 64 bits:
  - Mayor (pos/neg): ± 1.7976931348623157 x 10^{308}
  - Menor (pos/neg): ± 5 x 10^{-324}
  - Enteros: -9.007.199.254.740.992 ... 9.007.199.254.740.992
- Literales enteros en hexadecimal: precedidos por `0x`
- Valores especiales:
  - `NaN`
  - `Infinity`

Operadores aritméticos

- `++, -- (post)`
- `- unario`
- `++, -- (pre)`
- `*, /, %`
- `+, -`

El objeto Math

- `Math.abs(x)`
- `Math.sqrt(x)`

- `Math.cos(x)` `Math.sin(x)` `Math.tan(x)`
- `Math.acos(x)` `Math.asin(x)` `Math.atan(x)` `Math.atan2(y,x)`

  Ángulos en radianes

- `Math.ceil(x)` `Math.floor(x)` `Math.round(x)`

- `Math.exp(x)` `Math.log(x)` `Math.pow(x,y)`

  log en base e

- `Math.random()`

  Devuelve aleatorio en [0,1[

- `Math.max(a,b,…)` `Math.min(a,b,…)`

  Número arbitrario de argumentos

- `Math.PI` `Math.E` `Math.SQRT2` `MATH.SQRT1_2`
- `Math.LN2` `Math.LN10` `Math.LOG2E` `Math.LOG10E`

Situaciones típicas de error (**¡en JS no producen error!**):

- Situación de overflow ... devuelve el valor `Infinity` o `-Infinity`

```
var a = 1.7e308;
var b = 2*a; // Overflow: Infinity
```

- Situación de underflow ... devuelve el valor `0` o `-0`

```
var a = 5e-324;
var b = a/2; // Underflow: 0
```

- División por cero ... devuelve el valor `Infinity` o `-Infinity`

```
var a = 3/0; // Infinity
```

- `0/0` ... devuelve el valor `NaN`

```
var a = 0/0; // NaN
```

- Otras operaciones no definidas ... devuelven el valor `NaN`

```
var a = Math.sqrt(-2);  // NaN
```

- `Infinity/Infinity` ... devuelve el valor `NaN`

Valores booleanos

- `true`
- `false`

Operadores lógicos

- `!` Negación
- `&&` Conjunción lógica
- `||` Disyunción lógica

Equivalencia con otros tipos

Equivalentes a false (*falsy values*):

- `undefined`
- `null`
- `0`
- `-0`
- `NaN`
- `“”`

El resto de valores equivalen a true (*truthy values*)

Operadores lógicos &&, ||

- Ambos se evalúan en corto.
- Los operandos no tienen porqué ser booleanos, lo que se comprueba es si evalúan a truthy o falsy.
- Devuelven uno de sus operandos (no un valor true o false)

A && B

- Si A evalúa a falsy:
  - Devuelve A
  - (No evalúa B)
- Si no:
  - Devuelve B

A || B

- Si A evalúa a truthy:
  - Devuelve A
  - (No evalúa B)
- Si no:
  - Devuelve B

```
var a = {x:1, y:2 };
var b = "hola";

var c = a && b;
var d = a || b;

console.log(c);
console.log(d);
```

```
hola
Object { x: 1, y: 2 }
```

```
false && “hola”;  // false
false || “hola”;  // “hola”

“hola” && false;  // false
“hola” || false;  // “hola”

“hola” && “adios”;  // “adios”
“hola” || “adios”;  // “hola”
```

Cadenas de caracteres

- Son inmutables (no se pueden modificar)
- Almacenan caracteres Unicode (UTF-16). Cada carácter ocupa 2 bytes
- Se indexan desde la posición cero
- No existe el tipo "carácter" → Cadena de longitud uno
- Los literales se escriben entre comillas simples o dobles

Ejemplos de cadenas:

- "π"
  Tiene longitud 1 (aunque ocupa 2 bytes), UTF-16=0x03C0
- "e" (símbolo del número e, UTF-16=0x1D452)
  Tiene longitud 2, ocupa 4 bytes según codificación UTF-16)
- "texto de prueba"
- 'texto de prueba'
- "con \n secuencias \t de escape"

Operaciones sobre cadenas (algunas ... hay más)

`+`	Concatenación de cadenas
`cad.length`	Longitud de la cadena
`cad.charAt(pos)`	Carácter (cadena) de la posición pos
`cad1.concat(cad2,cad3,…)`	Devuelve la concatenación de dos o más cadenas
`cad.endsWith(cad2)`	Devuelve si `cad` acaba en `cad2`
`cad.includes(cad2,pos)`	Devuelve si `cad` incluye `cad2` (buscando desde posición `pos` –opcional-)
`cad.indexOf(cad2,pos)`	Posición de cad2 dentro de cad (buscando desde posición `pos` –opcional-). -1 si no está
`cad.match(reg)`	Devuelve en un array las ocurrencias de `cad` que cumplen la expresión regular `reg`
`cad.replace(old,new)`	Busca la cadena `old` y la sustituye por `new` (1ª ocur.) `old` puede ser expr.reg → todas las ocurrencias
`cad.search(cad2)`	Busca `cad2` en `cad` (-1 si no está) `cad2` puede ser expresión regular
`cad.substring(ini,fin)`	Subcadena desde `ini` hasta `fin-1`
`cad.substr(ini,lon)`	Subcadena desde `ini` y de longitud `lon`
`cad.split(sep)`	Subdivide la cadena en subcadenas
`cad.trim()`	Elimina espacios de principio y fin
`cad[pos]`	Acceso al elemento de pos (ECMAScript 5)
…	

Ningún método modifica el objeto que lo invoca

Objetos en JavaScript

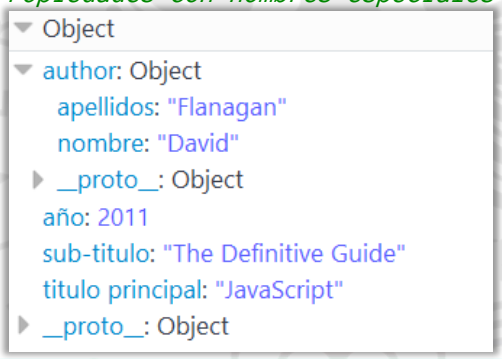
- Son colecciones desordenadas de propiedades
- Cada propiedad tiene un nombre (string) y un valor
- Si las propiedades son funciones se denominan métodos

```
var vacio = { };              // Objeto literal vacío

var punto = { x:0, y:0 };     // Objeto literal con 2 propiedades
punto.x = 3;                  // Acceso a propiedades
punto.y = 7;

var libro = {                 // Objeto literal compuesto por otros objetos
        "titulo principal": "JavaScript",    // Propiedades con nombres especiales
        'sub-titulo': "The Definitive Guide",
        "año": 2011,
        author: {
             nombre: "David",
             apellidos: "Flanagan"
        }
};

var obj = new Object(); // Objeto vacío
```



Acceso a las propiedades de un objeto

```
var a = libro["año"];                          // a=2011

libro["sub-titulo"]="The very long guide";     // Modificar propiedad
libro.author.nombre="Deivid";
libro["author"]["nombre"]="Deivid";

libro.editorial="O'Reilly";                    // Añadir propiedad

delete libro.editorial;                        // Borrar propiedad
```

Creación de objetos con el operador new

```
var a = new Object(); // Objeto vacío { }
var b = new Number(); // Número 0
var c = new String(); // Cadena vacía

var a = new Object({x:3,y:4});
var b = new Number(467);
var c = new String("Hola");
```

JavaScript. Elementos básicos
Wrapper objects

Los valores primitivos de tipo cadena, número y bool no son objetos y, por tanto, no tienen métodos ni propiedades

```
var cad="hola pepe";
console.log(typeof(cad));

var t = cad.length;
console.log(t);

var p = cad.indexOf("pe");
console.log(p);
```

```
string
9
5
```

- Un dato string NO es un objeto String
- JS, cuando lo necesita, crea un objeto (*wrapper object*) a partir del tipo primitivo para poder usar propiedades y métodos

```
var cad1="hola pepe";
console.log(typeof(cad1));

var cad2=new String("hola pepe");
console.log(typeof(cad2));

var cad3=cad2.valueOf(cad2);
console.log(typeof(cad3));
```

```
string
object
string
```


JavaScript. Elementos básicos
Arrays

Arrays en JavaScript

Son colecciones ordenadas de valores de cualquier tipo
Cada valor puede diferir en tipo
Son indexados desde el cero
Son dinámicos
Pueden ser dispersos (*sparse*)

```
var a1=[1,2,5,3];
var a2=["pepe","juan","maría"];
var a3=[4,"juan",true,NaN];
var a4=[1,,,,,,2];
var a5=[[1,2,3],{x:3,y:6}]
```

```
Array [ 1, 2, 5, 3 ]
Array [ "pepe", "juan", "maría" ]
Array [ 4, "juan", true, NaN ]
Array [ 1, <5 ranuras vacías>, 2 ]
Array [ Array[3], Object ]
```

```
var b1 = new Array(); // Crea un array vacío
var b2 = new Array(10); // Crea un array de tamaño 10 (sin elementos)
var b3 = new Array(4,2,5,"hola");
```

JavaScript. Elementos básicos
Arrays

```
var a = [6,7,8];
var x = a[2];          // x=8                Acceso a elementos
a[0] = 4;              // a=[4,7,8]
var tam = a.length;    // tam=3              Consulta de tamaño
a[3] = 9;              // a=[4,7,8,9]        Añadir elemento
a[7] = 3;              // a=[4,7,8,9,,,,3]   Añadir elemento
```

Los arrays son objetos:

- Los índices son en realidad propiedades ... al escribir a[0] se crea una propiedad llamada "0"
- Pueden tener más propiedades

```
var a = [1,2];
var x = a["1"];           // x = 2
a["nombre"] = "pepe";     // Añade una propiedad (no índice)
a["2"] = 3;               // a = [1,2,3]
var tam = a.length;       // tam = 3 (nombre es propiedad, no índice)
a[-1] = 99;               // Propiedad
a[1.43] = 77;             // propiedad
var y = a[43];            // y=undefined
```


JavaScript. Elementos básicos
Arrays

Operaciones sobre arrays (algunas ... hay más)

• `a.reverse()`	Invierte el array
• `a.sort()`	Ordena los elementos
• `a.concat(b)`	Añade los elementos de b
• `a.slice(ini,fin)`	Devuelve un subarray
• `a.push(elem)`	Añade elementos al final
• `a.pop()`	Elimina elementos del final
• `a.unshift(elem)`	Añade elementos al principio
• `a.shift()`	Borra elementos del principio
• ...	

```
var a = [1,2,3,4,5]; a.push(6);   // a=[1,2,3,4,5,6]
a.pop();                          // a=[1,2,3,4,5]
delete a[2];                      // a=[1,2,,4,5]
delete a[4];                      // a=[1,2,,4,]
a.length = 2;                     // a=[1,2]
```

Operadores de asignación

Asignación:

`=`

Asignaciones que incluyen una operación

`+=` `-=` `*=` `/=` `%=`	(aritméticas)
`<<=` `>>=` `>>>=`	(desplazamientos de bits)
`&=` `|=` `^=`	(de bits)

Estos operadores devuelven el mismo valor que se asigna (permite asignaciones en cadena)

El tipo objeto también es llamado tipo referencia (reference type):
La variable almacena una referencia al objeto
(Esto no ocurre con los tipos primitivos: booleanos y números)

```
var a=4;
var b=a;
b=5;
console.log(a);   // 4
console.log(b);   // 5
```

```
var o1 = { x:1 };
var o2 = o1;
o2.x = 2;
console.log(o1);   // Object { x:2 }
console.log(o2);   // Object { x:2 }
```


UNIVERSIDAD DE GRANADA

Tecnologías Web

3º Grado en Ingeniería Informática

Programación en el lado del cliente - JavaScript

DECSAI

JavaScript. Estructuras de control
Condicionales

```
if (a>1)
  console.log("a es mayor que 1");
```

```
if (a>1)
  console.log("a es mayor que 1");
else
  console.log("a no es mayor que 1");
```

```
if (a>1) {
  console.log("a es mayor que 1");
  console.log("y aquí hay más de una instrucción");
} else
  console.log("a no es mayor que 1");
```

```
if (a<1)
  console.log("a es menor que 1");
else if (a==2)
  console.log("a es igual a 2");
else
  console.log("ninguna de las otras");
```


JavaScript. Estructuras de control
Operadores relacionales

Operadores relacionales de comparación

Comprueban el orden relativo de los operandos (numérico o alfabético)

`<` `>` `<=` `>=`

Sólo pueden comparar números y cadenas. Si un operando es de otro tipo: se convierte de forma implícita.

1. Si algún operando es un objeto → se convierte a un valor primitivo
   Para ello JS usa los métodos toString() y/o valueOf()
2. Si ambos valores primitivos son cadenas, se compara el orden alfabético según Unicode de 16 bit
3. Si al menos uno de ellos no es una cadena ambos se convierten a números y se comparan. Si alguno de ellos vale `NaN` el resultado de la comparación es false.

```
console.log( 11 < 3 );      // false
console.log( "11" < "3" );  // true  (comparación de cadenas)
console.log( "11" < 3 );    // false ("11" se convierte a 11)
console.log( "once" < 3 );  // false ("once" se convierte a número: NaN)
```

Secciones 3.8.3 y 4.9.2. Flanagan. JavaScript: The definitive guide (6ed). O'Reilly. 2011

Operadores relacionales de comparación de igualdad

Comprueban la igualdad o desigualdad de los operandos

===	!==	Comparación estricta de (des)igualdad (o de identidad)
==	!=	Comparación de (des)igualdad

=== Dos operandos se consideran estrictamente iguales si:

- Valores primitivos: si son del mismo tipo y tienen el mismo valor
- Objetos: si son el mismo objeto
- NaN es diferente de todo (incluso de NaN)

== Dos operandos se consideran iguales si:

- Si tienen igual tipo se comprueba con ===
- Si tienen diferente tipo:
  - numero == string : convertir el string a número
  - true equivale a 1
  - false equivale a 0
  - null==undefined
  - Si un operando es un objeto y el otro un número o cadena, se intenta convertir el objeto a valor primitivo (toString o valueOf)

Secciones 3.8.3 y 4.9.2. Flanagan. JavaScript: The definitive guide (6ed). O'Reilly. 2011


```
// Coincidencia de valor (con conversiones)
console.log( 3 == 3 );       // true: comparación de números
console.log( "3" == "3" );   // true: comparación de string
console.log( "3" == 3 );     // true: "3" se convierte a 3
console.log( true == 1 );    // true: true se convierte a número 1

// Coincidencia de tipo y valor (sin conversiones)
console.log( 3 === 3 );      // true: comparación de números
console.log( "3" === "3" );  // true: comparación de string
console.log( "3" === 3 );    // false: no son el mismo objeto
console.log( true === 1 );   // false: no son el mismo objeto


var a1 = ["antonio","juan"];
var a2 = ["antonio","juan"];
var a3 = a1;
console.log( a1==a2 );    // false: usa ===
console.log( a1===a2 );   // false: no son el mismo objeto
console.log( a1==a3 );    // true: usa === (son el mismo objeto)
console.log( a1===a3 );   // true: son el mismo objeto (referencia)
```

JavaScript. Estructuras de control
Condicionales

```
switch (a) {
  case 1: console.log("vale 1");
          break;
  case 2: console.log("vale 2");
          break;
  case 3: console.log("vale 3");
          break;
  default: console.log("vale otra cosa")
}
```

- Mismo comportamiento que en C++
- Las etiquetas pueden ser expresiones
- La comparación se hace con === (sin conversiones)

JavaScript. Estructuras de control
Bucles

```
var x=0;
while (x<4) {
  console.log(x);
  x++;
}
```

```
var x=0;
do {
  console.log(x);
  x++;
} while (x<4);
```

```
for (var x=0; x<4; x++)
  console.log(x);
```

Bucle for/in

Permite iterar sobre las propiedades de un objeto
El estándar no determina ningún orden preestablecido

```
for (var p in libro) {
  console.log(libro[p]);
}
```

```
JavaScript
The Definitive Guide
2011
Object { nombre: "David", apellidos: "Flanagan" }
```

```
var vec=[12,14,16,18];
```

```
for (p in vec)
  console.log(p);
```

```
0
1
2
3
```

```
for (p in vec)
  console.log(vec[p]);
```

```
12
14
16
18
```

También funciona con tipos primitivos (string)

```
var cad="hola";

for (p in cad)
  console.log(p);

for (p in cad)
  console.log(cad[p]);
```

```
0
1
2
3
h
o
l
a
```

```
var vec=[12,14,16,18];
vec["nombre"] = "el vector";
vec[7]=20;
vec["apellido"] = "de números";

for (p in vec)
  console.log(vec[p]);
```

```
12
14
16
18
20
el vector
de números
```


UNIVERSIDAD DE GRANADA

Tecnologías Web

3º Grado en Ingeniería Informática

Programación en el lado del cliente - JavaScript

DECSAI

Funciones en JavaScript

- Pueden ser recursivas
- Por defecto devuelven `undefined`
- Pueden anidarse

```
function fact(n) {
  var f=1;
  for (var i=2; i<=n; i++)
    f *= i;
  return f;
}
```

```
function fact(n) {
  if (n<2)
    return 1;
  else
    return n*fact(n-1);
}
```

```
function cuarta(n) {
  function cuadrado(n) {
    return n*n;
  }
  return cuadrado(cuadrado(n));
}
```

```
function f(v) {
  for (e in v)
    console.log(v[e]);
}
var x = f([1,4,3,2]);
console.log(x);
```

```
1
4
3
2
undefined
```


Funciones como objetos

Las funciones son objetos:
- Pueden asignarse a variables
- Se pueden definir en expresiones (*function expressions*)

```
var cuadrado = function f(n) { return n*n; }  // Function expression (asignada)
console.log(cuadrado(3));
vector.sort(function f(a,b) { return a-b; }); // Function expression (argumento)
```

Las funciones son objetos:
- Pueden tener propiedades
- Estas propiedades son globales a todas las llamadas a la función (similar a static en C++)

```
function siguiente() {
  siguiente.valor++;
  return siguiente.valor;
}

siguiente.valor=0;
console.log(siguiente());
console.log(siguiente());
console.log(siguiente());
```

Funciones en JavaScript

El nombre es opcional (funciones anónimas):

- Cuando se usa en una expresión
- Cuando se invoca en el momento de ser definida

```
var cuadrado = function(n) { return n*n; }   // Function expression anónima
console.log(cuadrado(3));
```

IIFE: Inmediatly-invoked function expression

```
var cuadrado = (function(n) { return n*n;}) (3);
console.log(cuadrado);

(function(x) { console.log("Hola "+x); })("Javi");
```


Funciones en JavaScript

Método: función asignada a una propiedad de un objeto

```
var animal = [ { nombre: "Beethoven", tipo: "Perro" },
               { nombre: "Doraemon", tipo: "Gato" },
               { nombre: "Scooby", tipo: "Perro" },
               { nombre: "Garfield", tipo: "Gato" } ];

for (a in animal)
  if (animal[a].tipo=="Perro")
    animal[a].llamar = function() { console.log("Guau"); };
  else if (animal[a].tipo=="Gato")
    animal[a].llamar = function() { console.log("Miau"); };
  else
    animal[a].llamar = function() { console.log("..."); };

for (a in animal)
  animal[a].llamar();
```

```
Guau
Miau
Guau
Miau
```

El paso de parámetros

Siempre es copia por valor

- En el caso de tipos primitivos (números, booleanos, cadenas) el resultado es el esperado
- En el caso de objetos: es un paso por copia ... de la referencia al objeto ... por lo que a efectos prácticos se comporta como un paso por referencia

```
function f(a) {
  a=3;
}

var x=1;
console.log(x);
f(x);
console.log(x);
```

```
function f(a) {
  a="adiós";
}

var x="hola";
console.log(x);
f(x);
console.log(x);
```

```
function f(x) {
  x.a=3;
}

var obj={a:1};
console.log(obj);
f(obj);
console.log(obj);
```


Sobre los parámetros de las funciones

- Puede haber menos argumentos que parámetros, es decir, los parámetros son opcionales.
- Si no se usa un argumento, el parámetro vale `undefined`

```
function f(a,b) {
  console.log(a,b);
}
f(1,2); // 1 2
f(3);   // 3 undefined
f();    // undefined undefined
```

```
function f(a) {
  if (a===undefined)
    a=1;
  console.log(a);
}
f(3);   // 3
f();    // 1
```

Si el argumento que se espera es un objeto es frecuente comprobar así:

```
function f(a) {
  a = a || {objeto};
  console.log(a);
}
```

Sobre los parámetros de las funciones

- Puede haber más argumentos que parámetros
- Independientemente de que se hayan usado argumentos o no, siempre podemos usar el parámetro implícito `arguments`, que es un objeto similar a un array con todos los argumentos usados (NO es un array).

```
function f(a,b) {
  if (arguments.length>0)
    console.log("a: ", a);
  console.log("Total: ", arguments.length);
  for (p in arguments)
    console.log(arguments[p]);
}

f(2,3,"hola",{a:2});
```

```
a:  2
Total:  4
2
3
hola
Object { a: 2 }
```


Function hoisting

- Las declaraciones completas de las funciones son elevadas al comienzo del script
- Pueden llamarse antes de haber sido declaradas

```
console.log(f1(2));

function f1(n) {
  return n*n;
}
```

- Las expresiones de tipo función asignadas a una variable no son elevadas. En ese caso solo se eleva la variable (variable hoisting: se eleva la declaración, no la asignación)

```
var f2 = function(n) { return n*n; };
console.log(f2(2));

console.log(f2(2));
var f2 = function(n) { return n*n; };
```

https://jherax.wordpress.com/2014/07/04/javascript-hoisting/

JavaScript. Funciones
Contexto, ámbito y el objeto this

Ámbito
Relacionado con el acceso a las variables

Contexto
Relacionado con el objeto `this` y con la forma de invocar la función

El objeto `this`
- `this`: objeto que hace referencia al contexto en el que se está ejecutando el código
- Objeto global `this`: contiene todas las funciones, variables, objetos, etc, globales

```
var x=2;
console.log(x);
console.log(this.x);
```

```
function cuad(n) { return n*n; }
console.log(cuad(2));
console.log(this.cuad(2));
```

Patrones de llamada a una función
Definen el contexto (`this`) del código
- *Function*: como función global
- *Method*: como método de un objeto
- *Constructor*: como constructor de un objeto
- *Indirect*: hace explícito el contexto

JavaScript. Funciones
El patrón de llamada *Function*

Patrón de llamada *function*
Llamada habitual de funciones
Dentro de la función, el objeto `this` es **siempre** el objeto global `this`

```
var x=1;
function f() {
  var x=2;
  console.log(x);
  console.log(this.x);
}
f();
console.log(x);
```

Ámbito != Contexto

```
var y=1;
function f() {
  var y=2;
  console.log(y);
  console.log(this.y);
  function g() {
    var y=3;
    console.log(y);
    console.log(this.y);
  }
  g();
}
f();
console.log(y);
```

Particularmente confuso cuando se usa en métodos de objetos (POO)

Patrón de llamada *method*

Llamada de métodos de objetos
Dentro del método, `this` es el objeto que se ha usado para hacer la llamada

```
var obj = {
  x : 0,
  escribeSiguiente : function() {
    console.log(this.x);
    this.x++;
  }
};

obj.reset = function() {
  this.x=0;
}

obj.escribeSiguiente();    // 0
obj.escribeSiguiente();    // 1
obj.escribeSiguiente();    // 2
obj.reset();
obj.escribeSiguiente();    // 0
obj.escribeSiguiente();    // 1
obj.escribeSiguiente();    // 2
```


Patrón de llamada *method*

Llamada de métodos de objetos
Dentro del método, `this` es el objeto que se ha usado para hacer la llamada

```
var a=1;
var obj = {
      a: 2,
      f: function() {
            console.log(a);
            console.log(this.a);
         }
}
obj.f();
```

```
var a=1;
var obj = {
      a: 2,
      f: function() {
            console.log(a);
            console.log(this.a);
            function g() {
                console.log(a);
                console.log(this.a);
            }
            g();
      }
}
obj.f();
```

Ámbito != Contexto

Patrón de llamada *method*

Llamada de métodos de objetos
Dentro del método, `this` es el objeto que se ha usado para hacer la llamada

```
var a=1;
var obj = {
      a: 2,
      f: function() {
            console.log(a);
            console.log(this.a);
            function g() {
                console.log(a);
                console.log(this.a);
            }
            g();
      }
}
obj.f();
```

¿Desde g() se puede acceder a obj.a?

```
f: function() {
      var self = this;
      function g() {
          console.log(self.a);
          console.log(this.a);
      }
      g();
}
```

... más adelante se ve lo que es una "clausura"

Patrón de llamada *constructor*

Llamada de constructores de objetos con palabra reservada `new`
Dentro del constructor, `this` es el objeto que se está creando

JavaScript es un lenguaje sin clases pero con herencia por prototipado

```
var Matematico = function(x,y,s) {
  this.arg1 = x;
  this.arg2 = y;
  this.signo = s;
  this.resultado = function() {
    if (this.signo=='+')
      return this.arg1+this.arg2;
    else
      return this.arg1-this.arg2;
  }
}

obj1 = new Matematico(6,4,'+');
obj2 = new Matematico(6,4,'-');
console.log(obj1.resultado());
console.log(obj2.resultado());
```

Patrón de llamada *indirect*

Llamada a la función o método de forma indirecta a través de los métodos `call`, `apply`, `bind`

En la llamada se deja explícito quién es `this`

```
var obj1 = { nombre: 'Juan' }
var obj2 = { nombre: 'Pepe' }
var verDatos = function() { console.log(this.nombre); }

verDatos(); // undefined
verDatos.apply(obj1);
verDatos.apply(obj2);
verDatos.call(obj1);
verDatos.call(obj2);

var saludar = function(antes,despues) {
  console.log(antes + this.nombre + despues);
}
saludar.call(obj1,"Hola, ", ", ¿cómo estás?");    // Lista de argumentos
saludar.apply(obj2,["Hola, ", ", ¿cómo estás?"]); // Array con argumentos
```

Método `bind`: se verá más adelante

http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/
http://javascriptissexy.com/javascript-apply-call-and-bind-methods-are-essential-for-javascript-professionals/

Clausuras (closures)

Objeto especial formado por:
- Función
- Entorno en el que se ha creado la función

```
function inicia() {
  var nombre="Pepe";           // Variable local de la función
  function muestraNombre() {   // Función interna
    alert(nombre);
  }
  muestraNombre();
}
inicia();
```

```
function creaFunc() {
  var nombre="Juan";
  function muestraNombre() {
    alert(nombre);
  }
  return muestraNombre;
}
var miFunc=creaFunc();
miFunc();
```

- Las variables locales se destruyen al finalizar la función.
- Las funciones internas crean clausuras: mantienen todas las variables que están a su alcance en el momento de su definición

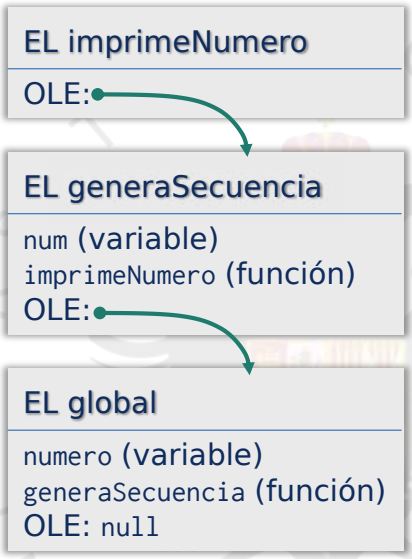
https://developer.mozilla.org/es/docs/Web/JavaScript/Closures

Clausuras (closures)

El Entorno Léxico (EL) es un objeto que contiene
- Variables locales del contexto en el que nos encontramos
- Referencia al entorno léxico inmediatamente anterior/superior (outer lexical environment)

```
function generaSecuencia() {
  var num = 1;
  function imprimeNumero() {
    console.log(num);
    num++;
  }
  return imprimeNumero;
}

var numero = generaSecuencia();
numero();   // 1
numero();   // 2
numero();   // 3
numero();   // 4
```



https://javascript.info/closure

Clausuras (closures)

Otro ejemplo: los parámetros también son variables locales

```
function creaSumador(x) {
  return function(y) {
    return x + y;
  };
}

var suma5 = creaSumador(5);
var suma10 = creaSumador(10);

console.log("Suma: " + suma5(2));  // 7
console.log("Suma: " + suma10(2)); // 12
```

- `suma5` es una clausura
- `suma10` es otra clausura
- Ambas contienen la misma función pero diferentes entornos

Clausuras (closures)

Error habitual en bucles ...

```
// Función que crea un array con 3 funciones
function crearFunciones() {
  var funcs=[];
  for (var i=0; i<3; i++) {
    var x = i;
    funcs[i] = function() {
      console.log("Valor de i: " + i);
      console.log("Valor de x: " + x);
    };
  }
  return funcs;
}

// Creamos el array
f=crearFunciones();

// Ejecutamos las funciones
for (var j=0; j<3; j++)
  f[j]();
```

```
Valor de i: 3
Valor de x: 2
Valor de i: 3
Valor de x: 2
Valor de i: 3
Valor de x: 2
```

http://stackoverflow.com/questions/750486/javascript-closure-inside-loops-simple-practical-example

Clausuras (closures)

Error habitual en bucles ... solución habitual: uso de IIFE (Inmediately-Invoked Function Expression)

```
function crearFunciones() {
  var funcs=[];
  for (var i=0; i<3; i++) {
    (function() {
      var x = i;
      funcs[i] = function() {
        console.log("Valor de i: " + i);
        console.log("Valor de x: " + x);
      };
    })();
  }
  return funcs;
}
```

```
Valor de i: 3
Valor de x: 0
Valor de i: 3
Valor de x: 1
Valor de i: 3
Valor de x: 2
```

Variable x:

- Es local a la función anónima
- Es diferente para cada ejecución de la misma

Clausuras (closures)

Error habitual en bucles ... solución con ECMAScript 6: uso de let

```
// Función que crea un array con 3 funciones
function crearFunciones() {
  var funcs=[];
  for (let i=0; i<3; i++) {
    let x = i;
    funcs[i] = function() {
      console.log("Valor de i: " + i);
      console.log("Valor de x: " + x);
    };
  }
  return funcs;
}

// Creamos el array
f=crearFunciones();

// Ejecutamos las funciones
for (var j=0; j<3; j++)
  f[j]();
```

```
Valor de i: 0
Valor de x: 0
Valor de i: 1
Valor de x: 1
Valor de i: 2
Valor de x: 2
```

http://stackoverflow.com/questions/750486/javascript-closure-inside-loops-simple-practical-example

Prec.	Descripción	Asoc	Operador
1		L2R R2L	. [] new
2	Llamada a función	L2R	()
3			++ --
4	Unarios	R2L	! ~ + - typeof void delete
5		L2R	* / %
6	Aritméticos	L2R	+ -
7		L2R	<< >> >>>
8		L2R	< <= > >= in instanceof
9		L2R	== != === !==
10		L2R	&
11		L2R	^
12		L2R	|
13		L2R	&&
14		L2R	||
15		R2L	? :
16		R2L	yield
17		R2L	= += -= *= /= %= <<= >>= >>>= &= ^= !=
18		L2R	,

Novedades de ES5

- Modo estricto (para fuentes de error habituales en programas)
  - Lanza excepciones en algunas situaciones donde antes no lo hacía
    - Asignar valores a objetos no modificables
    - Borrar propiedades que no son borrables
    - ...
  - Evita la creación de variables globales por error
  - Impide definir varias veces la misma propiedad
  - Impide nombres de parámetros duplicados en funciones
  - Prohibe la notación octal (números precedidos de 0)
  - Impide asignar propiedades a valores primitivos
  - Prohibe el uso de with
  - Impide usar algunas palabras como identificadores en previsión de futuros usos (let, implements, private, ...)
- Nuevos métodos sobre objetos
  - Impedir añadir nuevas propiedades a un objeto
  - Incorpora descriptores de propiedades:
    - value, get, set, writable, configurable, enumerable
  - ...
- Nuevos métodos sobre arrays
- Soporte nativo de JSON
- ...

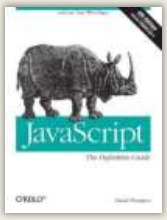
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Novedades de ES6

- Definición de constantes
- Definición de variables en ámbito de bloque (let)
- Funciones flecha (=>) para definir funciones anónimas de forma simplificada y resolver el problema del contexto de this
- Valores por defecto en parámetros
- Parámetros rest (número indeterminado de argumentos con ...)
- import/export similares a Python o Java
- class, extends
- Iteradores
- Funciones generadoras (permiten detenerse y volver más adelante por donde se quedaron)
- Estructura de datos: Set, Map
- Nuevos métodos sobre objetos, arrays, cadenas, etc

Soporte de ES6 en navegadores: http://kangax.github.io/compat-table/es6/

http://es6-features.org



David Flanagan
Javascript: The definitive guide (6th ed)
Addison Wesley. 2011

http://effectivejs.com/

Tim Wright
Learning JavaScript
Addison Wesley. 2013

http://learningjsbook.com

Ved Antani
Mastering JavaScript
Packt. 2016

Nicholas C. Zakas
Understanding ECMAScript 6
No Starch Press. 2016

http://javascriptissexy.com/
Artículos sobre cuestiones puntuales

http://javascript.crockford.com/
Varios artículos interesantes

http://exploringjs.com/
Libros de JS de Axel Rauschmayer

https://developer.mozilla.org/en-US/docs/Web/JavaScript