APUNTES DE

JAVASCRIPT

Nivel intermedio

Por JuanMa Garrido

Apuntes de Javascript I

Nivel Intermedio

JuanMa Garrido

Este libro está a la venta en http://leanpub.com/apuntes-javascript-intermedio

Esta versión se publicó en 2015-01-28

A mi madre...

"Si tu tonto no eres, lo que pasa es que te tiene que gustar" (mi madre)

Índice general

1. Prologo

Este libro contiene la **primera parte** de los materiales que he ido realizando para diferentes trainings JAVASCRIPT impartidos desde 2010.

Esta primera parte abarca desde lo más basico hasta un nivel intermedio que incluye:

- Las bases de javascript como lenguaje de programacion (variables, operadores, condicionales, bucles, tipos datos, etc...)
- Conceptos importantes asociados al lenguaje y como manejarlos (JSON, AJAX, eventos, dom, bom)
- Algunos conceptos un poco mas avanzados pero fundamentales para entener el javascript actual (closures, expresiones regulares, objetos)

Los conceptos cubiertos en este libro te permitiran trabajar y comprender el javascript del 90% de los proyectos

Espero que este libro te ayude a entender/implementar el codigo javascript que necesites.

Cualquier feedback será bien recibido :)

1.1 Referencias

Ademas de los enlaces reflejados, este material está basado en los siguientes libros:

- JavaScript: The Good Parts by Douglas Crockford
- Object-Oriented JavaScript by Stoyan Stefanov
- JavaScript Patterns by Stoyan Stefanov

1.2 Agradecimientos

Gracias a @gemmabeltra por la super-portada del libro (...y por aguantarme en general)

A @amischol por elevar mi nivel de javascript y el de todos los que hemos trabajado a su lado (cuando nosotros vamos, él ya tiene su chalet alli montado)

A @cvillu por animarme a escribir el libro y descubrirme LeanPub (South-Power!!!)

A @crossrecursion por nuestras charlas/reuniones hablando de Javascript (me excitas... intelectualmente)

Y a @softonic por darme la oportunidad de impartir los trainings JS y rodearme de tanta gente crack en JavaScript. No hay universidad que me pueda enseñar lo que he podido aprender alli.

2. Ideas Claras de Javascript

JavaScript es un lenguaje de programación interpretado por lo que no es necesario compilar los programas para ejecutarlos

Según una separacion en 3 capas de la pagina web, con el Javascript controlariamos la capa del comportamiento:

- Contenido → HTML
- Presentacion → CSS
- Comportamiento → Javascript

Javascript está basado en ECMAScript (o Ecma-262) que es una especificación de lenguaje de programacion (otro lenguaje "famoso" basado en este standard es *ActionScript*).

Las diferentes revisiones del Ecma-262 y su implementacion en los navegadores han ido marcando los desarrollos en Javascript

Con la llegada de AJAX (que no es más que el uso de un objeto javascript con el que podemos interactuar con el servidor sin tener que forzar una recarga de página) se abrió una nueva era en la historia del lenguaje

El uso tradicional de Javascript ha sido en el browser, pero ya se ha extendido su uso tambien en el lado del servidor (Node.js), en aplicaciones desktop y en aplicaciones mobile

Hay diferencias entre los navegadores debido al uso de diferentes motores de Javascript. Algunos de ellos son:

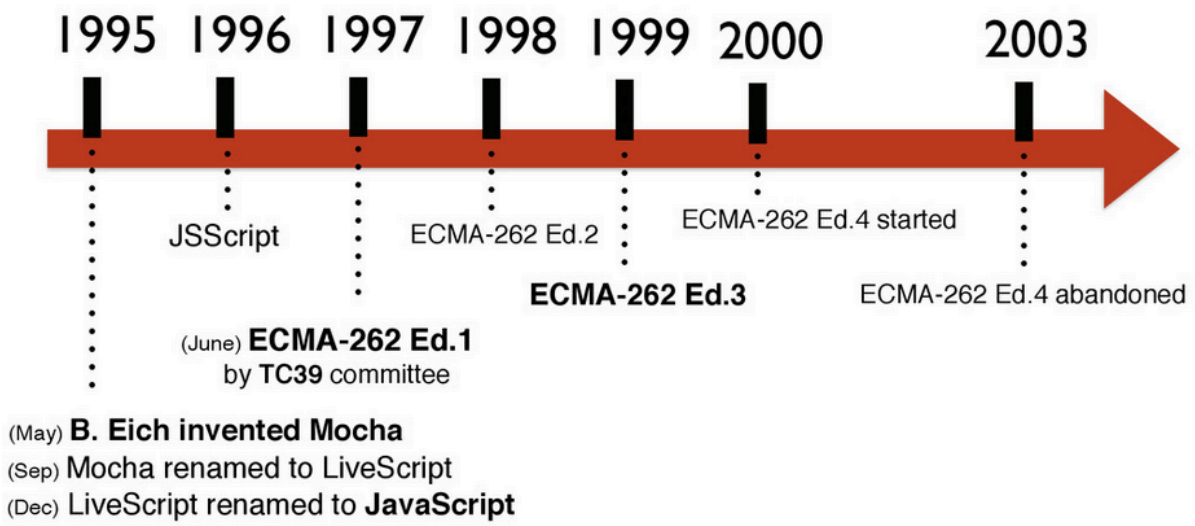
- Chrome → V8
- Firefox 4 → JagerMonkey
- Opera 10 → Carakan
- Safari → Nitro
- Internet Explorer 9 → Chakra

Aunque la diferencia grande siempre ha estado entre Internet Explorer y el resto (hasta IE9)

Estos interpretes (motores) de JS que hay en cada navegador, realizan optimizaciones de código cada uno a su manera de ahí el diferente rendimiento entre navegadores

Un Framework (o libreria) es una coleccion de utilidades comunmente utilizadas que pueden ser utilizadas para desarrollar aplicaciones ahorrando tiempo y esfuerzo. El framework mas conocido y utilizado es jQuery.

3. Breve historia de Javascript



JS History 1

Javascript fue creado en 10 dias en Mayo de 1995 por Brendan Eich, bajo el nombre de Mocha

La primera version del Javascript aparece en el navegador Netscape 2.0

En diciembre de 1995 SUN Microsystems y Netscape deciden darle el nombre **JavaScript** (antes se llamó *Mocha* y *LiveScript*) por una cuestion de puro marketing (*Java* era el lenguaje más popular por aquellos dias).

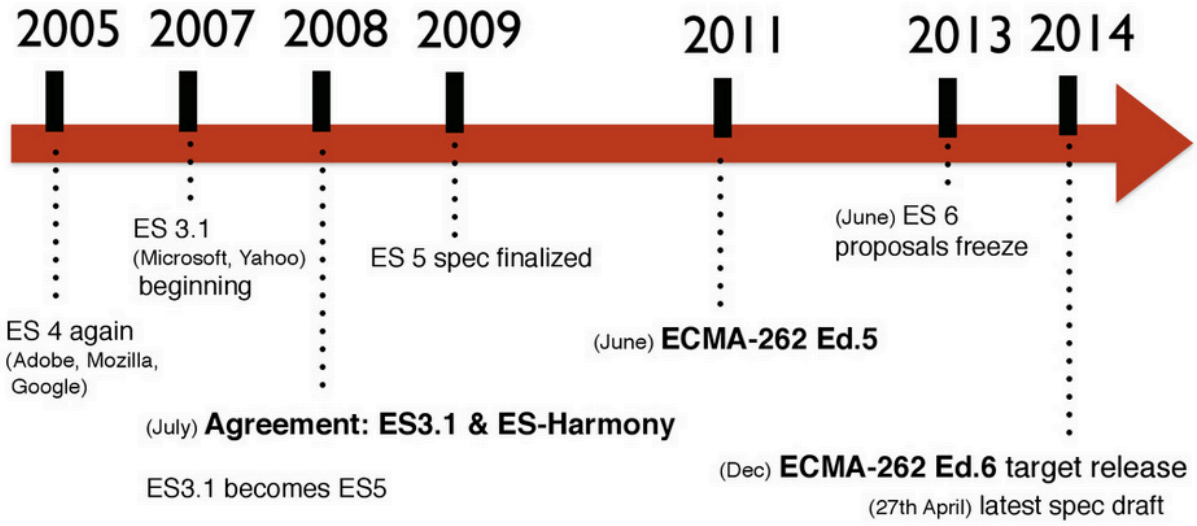
En 1996 Internet Explorer 3.0 incluye una version propia de lo que seria el standard ECMAScript que llama *JScript*

En 1997 se propuso este lenguaje como standard y la *European Computer Manufacturers Association* (ECMA) lo adopta como tal. De ahí que tambien se le llame **ECMAScript**

En Junio de 1997 se publica la primera edición del ECMA-262

En 1998 y a raiz de las diferencias surgidas entre navegadores, la *W3C (World Wide Web Consortium)* diseñó el standard **DOM** que es un interfaz (API) para acceder y modificar el contenido estructurado del documento.

En 1999, se sientan las bases del Javascript moderno con el lanzamiento de la tercera edición del ECMA-262, tambien llamado **EcmaScript 3**



JS History 2

En 2005, se acuña el termino AJAX y revoluciona el mundo del desarrollo web con la llegada de sitios web asíncronos (Gmail, Google Maps...)

En 2005 sale la primera versión de jQuery. Las diferencias entre navegadores han marcado los desarrollos en Javascript hasta el dia de hoy, y han hecho habitual el uso de frameworks (como jQuery) que nos ayuden a lidiar con estas diferencias.

En 2009 se completa y libera la quinta edición del ECMA-262, más conocida como **ECMAScript 5**. La edicion 5.1 se libera en 2011

En diciembre de 2014 se aprueba la 6ª edicion del ECMA-262 o **ECMAScript 6**. Se espera el lanzamiento oficial para Junio de 2015

4. ECMAScript

4.1 EcmaScript 5

ECMAScript5.1 fue lanzado en 2011 y podemos decir que es el actual standard de Javascript (2014).

Si miramos las estadisticas de uso de navegadores junto con la compatibilidad de estos con ES5 podemos concluir que: *basandonos en ES5 nuestro codigo funcionará bien en la mayoria de los navegadores utilizados actualmente (2014).*

Si queremos, podemos dar soporte de algunas features de ES5 en navegadores antiguos que no la soporten, utilizando el correspondiente shim

Esta version **amplia** los anteriores standards con algunas mejoras:

`strict mode`

Muy recomendado utilizarlo desde ya

```
function() {
  "use strict";
}
```

`Object` new methods

```
// Creates an object with parent as prototype and properties from donor
Object.create(parent, donor);

// Meta properties of an object
var descriptor = {
  value: "test",
  writable: true,    // Can the value be changed?
  enumerable: true,  // Will it appear in for-in and Object.keys(object)?
  configurable: true, // Can the property be removed?
  set: function(value) { test = value}, // Getter
  get: function() { return test }       // Setter
}

// Methods for manipulation the descriptors
Object.defineProperty(object, property, descriptor)
Object.defineProperties(object, descriptors)
```

```
  Object.getOwnPropertyDescriptor(object, property)
  Object.getPrototypeOf(object)

  // Returns an array of enumerable properties
  Object.keys(object)
  // Returns an array of all properties
  Object.getOwnPropertyNames(object)

  // Prevents anyone from adding properties to the object, cannot be undone.
  Object.preventExtensions(object)
  Object.isExtensible(object)

  // Prevents anyone from changing, properties or descriptors of the object.
  // The values can still be changed
  Object.seal(object)
  Objcect.isSealed(object)

  // Prevents any changes to the object.
  Object.freeze(object)
  Object.isFrozen(object)
```

Function.prototype.bind()

http://www.smashingmagazine.com/2014/01/23/understanding-javascript-function-prototype-bind/

```
var tapir = {
  method: function(name){
    this.name = name;
  }
};
setTimeout( tapir.method.bind(tapir, "Malayan"), 100 );
```

String.prototype.trim()

```
>>> var orig = '   foo  ';
>>> console.log(orig.trim());
'foo'
```

Array new methods

```
// Do all elements satisfy predicate?
Array.prototype.every(predicate)

// Return a new array with the elements that satisfy predicate?
Array.prototype.filter(predicate)

// Call action(element) for each element.
Array.prototype.forEach(action)

// What is the index of the first element that equals value?
Array.prototype.indexOf(value, fromIndex)

// What is the index of the last element that equal value?
Array.prototype.lastIndexOf(value, fromIndex)

// Create a new array by applying unaryFunc to each element
Array.prototype.map(unaryFunc)

// Reduces the elements of the array, by applying binaryFunc
// between the elements
// [a0, a1].reduce(+ , seed)  == seed + a0 + a1
Array.prototype.reduce(binaryFunc, seed)

// Is at least one element satisfied by the predicate?
Array.prototype.some(predicate)
```

Native JSON support

With `JSON.parse()` and `JSON.stringify()`

4.2 EcmaScript 3

ECMAScript 3 fue lanzado en 2001 y todos los navegadores (antiguos y modernos) siguen este standard.

Añade (respecto del standard anterior): `do-while`, expresiones regulares, nuevos metodos de `string` (`concat`, `match`, `replace`, `slice` , `split` con expresiones regulares, etc.), manejo de excepciones y más.

4.3 EcmaScript 6

ECMAScript 6 será el proximo standard de Javascript pero aun no está lo suficientemente implantado en los navegadores mas utilizados.

→ Learn ES6 | A detailed overview of ECMAScript 6 features
→ ECMAScript 6 Learning! (Eric Douglas Github)
→ ES6 Rocks | A collaborative website about the ECMAScript sixth edition, a.k.a. ES6.
→ Use ECMAScript 6 Today (tutsplus.com)

Aunque podemos dar soporte de estas features de ES6 en navegadores que no las soporten utilizando el correspondiente shim

5. Variables

Las Variables se utilizan para almacenar datos

Las variables solo tienen visibilidad (ambito) dentro de las funciones donde se declaran

Antes de poder utilizar una variable hay que:

- Declarar la variable (con la sentencia `var`)
- Inicializar la variable (en el momento de la declaración o despues)

```
var a = 1;
var b;
b = 2;
```

Las variables son Case Sensitive

```
var case_matters = 'lower';
var CASE_MATTERS = 'upper';
case_matters
CASE_MATTERS
```

¿Cómo chequear la existencia de una variable?

Forma Mala

```
>>> var result = '';
>>> if (somevar){result = 'yes';}
somevar is not defined
>>> result;
""
```

Genera un warning y que 'somevar' devuelva `false` no quiere decir que no esté definida

Forma Correcta

```
>>> var somevar;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""
```

```
>>> somevar = 123;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
"yes"
```

Si la variable está definida y tiene algún valor, su tipo de datos siempre será distinto de `undefined`

6. Primitivas y Tipos de Datos

Cualquier valor que se utilize en JS es de un cierto tipo. En Javascript existen los siguientes tipos de datos primitivos:

- **Number**: Puede contener numeros enteros (integer), decimales (float), hexadecimales, octales, exponentes y los números especiales `NaN` y `Infinity`
- **String**: Cualquier numero de caracteres entre comillas
- **Boolean**: puede ser `true` or `false`
- **Undefined**: Es un tipo de datos con un solo valor: `undefined`
  Lo devuelve JS cuando no existe una variable o no está inicializada.
- **Null**: Otro tipo de datos con un solo valor: `null`
  Lo podemos asignar nosotros para inicializar a vacio.

Cualquier valor que no pertenezca a uno de estos 5 tipo de primitivas es un objeto

Asi que los tipos de datos en javascript pueden ser:

- Primitivas (Los 5 tipos)
- No primitivas (Objetos)

Aunque existe el operador `typeof` que devuelve el tipo de dato, es mejor utilizar `Object.prototype.toString`

```
>>> typeof([1,2,3])
"object"
>>> Object.prototype.toString.call([1,2,3])
"[object Array]"
>>> typeof(function(){})
"function"
>>> Object.prototype.toString.call(function(){})
"[object Function]"
>>> typeof(new Date())
"object"
>>> Object.prototype.toString.call(new Date())
"[object Date]"
>>> typeof(27)
"number"
>>> Object.prototype.toString.call(27)
"[object Number]"
```

Existe el valor especial `NaN` (Not a Number) que obtenemos cuando intentamos hacer una operación que asume numeros pero la operación falla.

```
>>> var a = 10 * f;
>>> a
NaN
```

7. Operadores (Aritméticos, Lógicos y de Comparación)

Los Operadores toman uno o dos valores (o variables), realizan una operación, y devuelven un valor

El operador simple de asignación es =

```
1  var a = 1;
```

7.1 Operadores Aritméticos

Los operadores aritméticos básicos son:

+ Suma

```
1  >>> 1 + 2;
2  3
```

- Resta

```
1  >>> 99.99 - 11;
2  88.99
```

* Multiplicación

```
1  >>> 2 * 3;
2  6
```

/ División

```
1  >>> 6 / 4;
2  1.5
```

% Modulo

El resto de la división

```
>>> 6 % 3;
0
>>> 5 % 3;
2
```

Podemos utilizar el operador modulo, por ejemplo, para comprobar si un numero es par (`% = 0`) o impar (`% = 1`) .

```
>>> 4 % 2;
0
>>> 5 % 2;
1
```

++ Incremento en 1

Post-incremento devuelve el valor original (return) y despues incrementa el valor en 1.

```
>>> var a = 123; var b = a++;
>>> b;
123
>>> a;
124
```

Pre-incremento incrementa el valor en 1 y despues devuelve (return) el nuevo valor (ya incrementado).

```
>>> var a = 123; var b = ++a;
>>> b;
124
>>> a;
124
```

-- Decremento en 1

Post-decremento devuelve el valor original (return) y despues resta el valor en 1.

```
>>> var a = 123; var b = a--;
>>> b;
123
>>> a;
122
```

Pre-incremento resta el valor en 1 y despues devuelve (return) el nuevo valor (ya restado).

```
>>> var a = 123; var b = --a;
>>> b;
122
>>> a;
122
```

Tambien hay operadores compuestos

```
>>> var a = 5;
>>> a += 3;
8
```

7.2 Operadores Lógicos

Los Operadores Lógicos son:

- `!` ⟶ logical NOT (negation)
- `&&` ⟶ logical AND
- `||` ⟶ logical OR

```
>>> var b = !true;
>>> b;
false
```

La doble negación nos devuelve el valor original

```
>>> var b = !!true;
>>> b;
true
```

Las posibles operaciones y sus resultados son:

Operation	Result
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

7.3 Operadores de Comparación

Los Operadores de Comparación son:

== Igualdad

Devuelve `true` cuando los dos operandos son iguales. Los operandos son convertidos al mismo tipo de datos antes de la comparacion

```
>>> 1 == 1;
true
>>> 1 == 2;
false
>>> 1 == '1';
true
```

=== Igualdad y Tipo

Devuelve `true` cuando los dos operandos son iguales Y cuando son del mismo tipo de datos. Suele ser mejor y más seguro, utilizar esta comparación de igualdad (no hay transformaciones de tipo no controladas)

```
>>> 1 === '1';
false
>>> 1 === 1;
true
```

!= No Igualdad

Devuelve `true` cuando los dos operandos NO son iguales (despues de una conversion de tipo)

```
>>> 1 != 1;
false
>>> 1 != '1';
false
>>> 1 != 2;
true
```

!== No Igualdad Sin conversion de tipo

Devuelve `true` cuando los dos operandos NO son iguales o cuando son de tipos diferentes

```
>>> 1 !== 1;
false
>>> 1 !== '1';
true
```

> Mayor que

Devuelve `true` si el operando de la izquierda es mayor que el de la derecha

```
>>> 1 > 1;
false
>>> 33 > 22;
true
```

>= Mayor o Igual que

Devuelve `true` si el operando de la izquierda es mayor o igual que el de la derecha

```
>>> 1 >= 1;
true
```

< Menor que

Devuelve `true` si el operando de la izquierda es menor que el de la derecha

```
>>> 1 < 1;
false
>>> 1 < 2;
true
```

<= Menor o Igual que

Devuelve `true` si el operando de la izquierda es menor o igual que el de la derecha

```
>>> 1 <= 1;
true
>>> 1 <= 2;
true
```

8. Conversiones

Si utilizamos un numero entre comillas (string) en una operación aritmética Javascript lo convierte en numero

```
>>> var s = "100"; typeof s;
"string"
>>> s = s * 1;
100
>>> typeof s;
"number"
```

¡OJO! `undefined` y `null` devuelven cosas diferentes al convertirlas a numero

```
>>> 1*undefined
NaN
>>> 1*null
0
```

Si utilizamos `true` or `false` entre comillas Javascript lo convierte en string

```
>>> var b = "true"; typeof b;
"string"
```

La doble negación `!!` es una forma sencilla de convertir cualquier valor en su Booleano correspondiente.

```
>>> !!0
false
>>> !!1
true
>>> !!""
false
>>> !!"hola"
true
>>> !!undefined
false
>>> !!null
false
```

Aplicandolo podemos comprobar como cualquier valor covertido a Booleano es `true` excepto:

- `""`
- `null`
- `undefined`
- `0`
- `NaN`
- `false`

9. Condiciones (Bifurcaciones Condicionales)

Una *condición* es una estructura que realiza una tarea u otra dependiendo del resultado de evaluar una condición. Aquí vamos a ver:

- La estructura **IF... ELSE**
- La estructura **SWITCH**

Un Bloque (de código) es el conjunto de expresiones que quedan encerradas entre llaves. Estos bloques se pueden anidar.

```
{
    var a = 1;
    var b = 3;
    var c, d;
    {
        c = a + b;
        {
        d = a - b;
    }
}
```

9.1 Condicional `if - else`

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

Las partes de una condición if

- La sentencia `if`
- Una condición entre paréntesis. Esta condición siempre devolverá un booleano. Esta condición puede contener:
  - Una operación lógica: `!`, `&&` o `||`
  - Una comparación como `===`, `!=`, `>` y demás
  - Cualquier valor o variable que pueda ser convertido a Booleano
  - Una combinación de estas
- El bloque de código a ejecutar si se cumple la condición

En el `if` tambien puede haber una parte else opcional seguido de un bloque de código que se ejecutará si la condición se evalua a `false`

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

Entre el `if` y el `else`, pueden haber ilimitado numero de condiciones `else if`

```
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';
}
```

Existe también lo que se llama el operador ternario `?` que nos permite abreviar algunas sentencias if simples

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

9.2 Condicional `switch`

```
var a = '1';
var result = '';
switch (a) {
  case 1:
    result = 'Number 1';
    break;
  case '1':
    result = 'String 1';
    break;
  default:
    result = 'I don\'t know';
    break;
}
result;
```

Las partes de un switch

- La sentencia `switch`

- Una expresión entre paréntesis.
  Esta expresión normalmente será una variable, pero puede ser cualquier expresión que devuelva un valor
- Cierto numero de bloques `case` entre corchetes
- Cada sentencia `case` va seguida de una expresión.
  El resultado de esta expresión se compara con la expresión que hay después del switch.
  Si la comparación de igualdad devuelve `true`, se ejecuta el bloque que hay tras este `case`
- Puede (y debe) haber una sentencia `break` al final de cada bloque `case`.
  Estos `break` provocan la salida del `switch` (de esta manera nos aseguramos de ejecutar un solo bloque `case`)
- Tambien puede (y debe) haber una sentencia `default` que es seguida de un bloque de código que se ejecuta si ninguno de los `case` es evaluado a `true`

10. Bucles (Loops)

Un bucle es una estructura que nos permite repetir un bloque de código muchas veces.

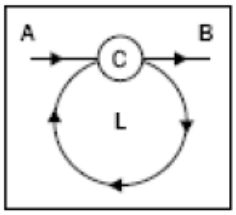
El número de repeticiones dependerá del resultado de evaluar una condición, antes (o después) de cada iteración

En Javascript hay 4 tipos de bucles:

- `while` loops
- `do-while` loops
- `for` loops
- `for-in` loops

10.1 El bucle `while`

```
1  var i = 0;
2  while (i < 10) {
3    i++;
4  }
```



Bucle While JS

La sentencia `while` va seguida de una condición entre paréntesis y un bloque de código entre corchetes. Mientras la condición se evalue a true, el código se ejecutará una y otra vez.

El número de repeticiones dependerá del resultado de evaluar una condición, antes (o después) de cada iteración

10.2 El bucle `do-while`

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

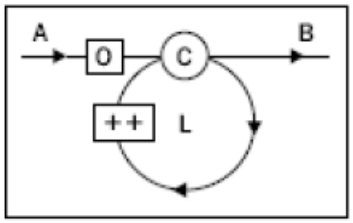
El bucle `do-while` es una pequeña variación del bucle `while`

La sentencia `do` va seguida de un bloque de código y una condición (con `while`) después del bloque.

Esto implica que el bloque de código se va a ejecutar siempre, al menos una vez, antes de evaluar la condición.

10.3 El bucle `for`

```
var punishment = '';
for (var i = 0; i < 100; i++) {
  punishment += 'I will never do this again, ';
}
```



Bucle For JS

La estrucura del bucle for tiene 3 partes claramente diferenciadas (separadas por

- **Inicialización** *(var i=0)*: Código que es ejecutado antes de entrar en el bucle [O]
- **Evaluación** *(i<100)*: Mientras evalue a true seguimos con el bucle [C]
- **Incremento** *(i++)*: Código que es ejecutado después de cada iteración [++]

Se pueden anidar bucles `for`

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

10.4 El bucle `for-in`

```
var a = ['a', 'b', 'c', 'x', 'y', 'z'];
var result = '\n';
for (var i in a) {
  result += 'index: ' + i + ', value: ' + a[i] + '\n';
}
```

El bucle `for-in` es utilizado para recorrer los elementos de un array (o de un objeto)

Aunque basandonos en ES5 tambien podriamos utilizar `Object.keys` y `forEach` y hacer

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

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

11. Funciones

```
1  function sum(a, b) {
2    var c = a + b;
3    return c;
4  }
```

Las **funciones** nos permiten agrupar varias líneas de código bajo un nombre. De esta forma podemos reutilizar este código, invocando el nombre de la función.

Las partes de una función:

- La *sentencia* `function`
- El *nombre* de la función (*sum*)
- *Parámetros* (argumentos) que espera la función (*a* y *b*) Una función puede aceptar cero o más argumentos separados por comas
- Un bloque de código, también llamado el *cuerpo de la funcion*
- La sentencia `return`
  Una función siempre devuelve un valor.
  Si no devuelve explícitamente un valor, implícitamente devuelve el valor `undefined`

Una función *solo puede devolver un valor.*
Si se necesita devolver mas de un valor, se puede devolver un array o un objeto con esos valores

Para llamar a una función tan solo tenemos que usar su nombre seguido de algunos parámetros (o ninguno) entre paréntesis

```
1  >>> var result = sum(1, 2);
2  >>> result;
3  3
```

11.1 Parametros

Una función puede no requerir parámetros, pero si los requiere y no se les pasa a la función, Javascript les asignará el valor `undefined`

Si la función recibe mas parámetros de los esperados, simplemente los ignorará

Dentro de cada función tenemos disponible el objeto (pseudo-array) `arguments` que contiene los argumentos pasados a la función

```
function sumOnSteroids() {
  var i, res = 0;
  var number_of_params = arguments.length;
  for (i = 0; i < number_of_params; i++) {
    res += arguments[i];
  }
  return res;
}
```

11.2 Funciones pre-definidas

Hay una serie de funciones que están directamente definidas dentro del motor de Javascript. Estas funciones pre-definidas son:

- `parseInt()`
- `parseFloat()`
- `isNaN()`
- `isFinite()`
- `encodeURI()`
- `decodeURI()`
- `encodeURIComponent()`
- `decodeURIComponent()`
- `eval()`

`parseInt()`

`parseInt()` toma un valor e intenta transformarlo en número entero. Si falla devuelve NaN. `parseInt()` admite un segundo parámetro opcional que indica la base del numero que se le está pasando (decimal, hexadecimal, binario, etc...)

```
>>> parseInt('123')
123
>>> parseInt('abc123')
NaN
>>> parseInt('1abc23')
1
>>> parseInt('123abc')
123
```

Se recomienda especificar siempre la base (10 normalmente) para evitar problemas de interpretaciones

```
>>> parseInt(" 0xF", 16);
15
>>> parseInt(" F", 16);
15
>>> parseInt("17", 8);
15
>>> parseInt(021, 8);
15
>>> parseInt("015", 10);
15
>>> parseInt(15.99, 10);
15
>>> parseInt("FXX123", 16);
15
>>> parseInt("1111", 2);
15
>>> parseInt("15*3", 10);
15
>>> parseInt("15e2", 10);
15
>>> parseInt("15px", 10);
15
>>> parseInt("12", 13);
15
```

parseFloat()

`parseFloat()` toma un valor e intenta transformarlo en número de coma flotante (con decimales).

```
>>> parseFloat('123')
123
>>> parseFloat('1.23')
1.23
>>> parseFloat('1.23abc.00')
1.23
>>> parseFloat('a.bc1.23')
NaN
```

isNan()

`isNan()` comprueba si el valor que se le pasa es un numero válido (devuelve true en caso de que no lo sea)

```
>>> isNaN(NaN)
true
>>> isNaN(123)
false
>>> isNaN(1.23)
false
>>> isNaN(parseInt('abc123'))
true
```

isFinite()

`isFinite()` comprueba si el valor que se le pasa no es ni Infinity ni NaN

```
>>> isFinite(Infinity)
false
>>> isFinite(-Infinity)
false
>>> isFinite(12)
true
>>> isFinite(1e308)
true
>>> isFinite(1e309)
false
```

encodeURI()

`encodeURI()` Nos permite 'escapar' (codificar) una URL reemplazando algunos caracteres por su correspondiente secuencia de escape UTF-8. *encodeURI()* nos devuelve una URL usable (solo codifica algunos caracteres)

```
>>> var url = 'http://www.packtpub.com/scr ipt.php?q=this and that';
>>> encodeURI(url);
http://www.packtpub.com/scr%20ipt.php?q=this%20and%20that
```

decodeURI()

`decodeURI()` Nos permite 'decodificar' un string codificado por `encodeURI()`

encodeURIComponent() y decodeURIComponent()

`encodeURIComponent()` y `decodeURIComponent()` Lo mismo que encodeURI()' pero esta función codifica (decodifica) TODOS los caracteres transformables

```
>>> encodeURIComponent(url);
"http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis%20and%20that"
```

eval()

`eval()` toma una cadena de texto y la ejecuta como código Javascript

`eval()` no debe utilizarse básicamente por 2 motivos:

- Rendimiento: Es mucho más lento evaluar código "en vivo" que tenerlo directamente en el script
- Seguridad: Teniendo en cuenta que ejecuta todo lo que se le pase puede ser un agujero de seguridad.

```
>>> eval('var ii = 2;')
>>> ii
2
```

alert()

`alert()` Nos muestra una ventana con un string
`alert()` no es parte del core JS pero está disponible en todos los navegadores

¡OJO! `alert()` para el código hasta que se acepte el mensaje

11.3 Ámbito (Scope) de las Funciones

En javascript las variables se definen en el ámbito de una función (y no en el ámbito de un bloque como ocurre en otros lenguajes)

- Las **variables globales** son aquellas que se definen fuera de cualquier función
- Las **variables locales** son aquellas que se definen dentro de una función

11.4 Funciones Callback

Las funciones en Javascript son datos, lo que significa que podemos asignarlas a variables igual que cualquier otro valor (y manejarlas como variables)

```
function f(){ return 1; }
var f = function(){ return 1; }
```

Las funciones son datos, pero un tipo especial de datos (`typeof 'function'`) ya que:

- Contienen código
- Podemos ejecutarlas

```
>>> var sum = function(a, b) { return a + b; }
>>> var add = sum;
>>> delete sum
true
>>> typeof sum;
"undefined"
>>> typeof add;
"function"
>>> add(1, 2);
3
```

Las **funciones anónimas** son aquellas que no tienen nombre y se pueden utilizar para:

- Pasar esa función como argumento de una función
- Definir una función y ejecutarla inmediatamente

```
>>> function(a){ return a; }
```

Cuando pasamos una función A como argumento de otra función B y B ejecuta A, decimos que A es una **función callback**

```
>>> function invoke_and_add(a, b){ return a() + b(); }
>>> function one() { return 1; }
>>> function two() { return 2; }
>>> invoke_and_add(one, two);
3
>>> invoke_and_add(one, function(){return 7;})
8
```

11.5 Closures

Si definimos una función `n()` dentro de `f()` , `n()` tendrá acceso tanto a las variables de su scope (ámbito) como las del scope de su padre. Esto es lo que se llama **scope chain** (encadenamiento de ámbitos)

```
var a = 1;
function f(){
  var b = 1;
  function n() {
    var c = 3;
  }
}
```

Las funciones tienen lo que se llama **lexical scope** (ámbito léxico) lo que significa que crean su entorno (scope, a qué variables tienen acceso) cuando son definidas no cuando son ejecutadas

```
>>> function f1(){ var a = 1; return f2(); }
>>> function f2(){ return a; }
>>> f1();
a is not defined
>>> var a = 5;
>>> f1();
5
>>> a = 55;
>>> f1();
55
>>> delete a;
true
>>> f1();
a is not defined
```

```
var a = 123;
function f() {
  alert(a);
  var a = 1;
  alert(a);
}
f();
```

Un **closure** se crea cuando una funcion mantiene un enlace con el ámbito (scope) de la función padre incluso después de que la función padre haya terminado.

```
function f(){
  var b = "b";
  return function(){
    return b;
  }
}
>>> b
b is not defined
>>> var n = f();
>>> n();
"b"
```

```
var n;
function f(){
  var b = "b";
  n = function(){
    return b;
  }
}
>>> f();
>>> n();
```

```
function f(arg) {
  var n = function(){
    return arg;
  };
  arg++;
  return n;
};
>>> var m = f(123);
>>> m();
```

→ Closures and lexical scoping | Mark Story
→ How do JavaScript closures work? | StackOverflow
→ Secrets of JavaScript: closures | kryogenix.org
→ Closing The Book On Javascript Closures | hunlock.com
→ A Graphical Explanation Of Javascript Closures In A jQuery Context | Ben Nadel

12. Arrays

Un **array** es una lista de valores.

Para asignar valores a un array encerramos los elementos entre corchetes (*array literal notation*)

Los elementos de un array *son indexados con números consecutivos a partir de 0*

Para acceder a un elemento del array especificamos el *índice entre corchetes*

```
var a = [1,2,3];
>>> a[0]
1
>>> a[1]
2
```

Podemos declarar un array vacio asi: `var a = [];`

Para *añadir/actualizar* un elemento del array:

```
>>> a[2] = 'three';
"three"
>>> a
[1, 2, "three"]
>>> a[3] = 'four';
"four"
>>> a
[1, 2, "three", "four"]
```

Para *eliminar* un elemento del array podemos utilizar el operador `delete`:

```
>>> var a = [1, 2, 3];
>>> delete a[1];
true
>>> a
[1, undefined, 3]
```

Un array puede contener otros arrays.

```
>>> var a = [1, "two", false, null, undefined];
>>> a
[1, "two", false, null, undefined]
>>> a[5] = [1,2,3]
[1, 2, 3]
>>> a
[1, "two", false, null, undefined, [1, 2, 3]]
```

```
>>> var a = [[1,2,3],[4,5,6]];
>>> a
[[1, 2, 3], [4, 5, 6]]
>>> a[0]
[1, 2, 3]
>>> a[0][0]
1
>>> a[1][2]
6
```

13. Objetos

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja'
};
```

Un **objeto** es como un array pero donde los índices los definimos nosotros

Para definir un objeto utilizamos las llaves `{}` (*object literal notation*)

Los elementos de un objeto (*propiedades*) los separamos por comas

El par *clave/valor* (key/value) lo dividimos con 2 puntos

Las claves (*keys*, nombres de las propiedades) pueden ir entre comillas, pero *no se recomienda* definirlas asi

```
var o = {prop: 1};
var o = {"prop": 1};
var o = {'prop': 1};
```

Cuando una propiedad contiene una función, decimos que esta propiedad es un *método del objeto*

```
var dog = {
  name: 'Benji',
  talk: function(){
    alert('Woof, woof!');
  }
};
```

Hay 2 maneras de acceder a la propiedad de un objeto:

- Con la notación de corchetes: `hero['occupation']`
- Con la notación de puntos: `hero.occupation`

Los objetos pueden contener otros objetos

```
var book = {
  name: 'Catch-22',
  published: 1961,
  author: {
    firstname: 'Joseph',
    lastname: 'Heller'
  }
};
>>> book.author.firstname
"Joseph"
>>> book['author']['lastname']
"Heller"
>>> book.author['lastname']
"Heller"
>>> book['author'].lastname
"Heller"
```

Podemos definir un objeto vacio y luego añadirle (y quitarle) propiedades y métodos

```
>>> var hero = {};
>>> typeof hero.breed
"undefined"
>>> hero.breed = 'turtle';
>>> hero.name = 'Leonardo';
>>> hero.sayName = function() {return hero.name;};
>>> hero.sayName();
"Leonardo"
>>> delete hero.name;
true
>>> hero.sayName();
reference to undefined property hero.name
```

Cuando estamos dentro de un método, con `this` hacemos referencia al objeto al que pertenece (*"this object"*)

```
var hero = {
  name: 'Rafaelo',
  sayName: function() {
    return this.name;
  }
}
>>> hero.sayName();
"Rafaelo"
```

13.1 Funciones Constructoras

Otra manera de crear objetos es mediante **funciones constructoras**

Para crear objetos con estas funciones hay que usar el operador `new`

La ventaja que tiene utilizar estas funciones constructoras es que *aceptan parámetros al crear objetos*

```
1  function Hero(name) {
2    this.name = name;
3    this.occupation = 'Ninja';
4    this.whoAreYou = function() {
5      return "I'm " + this.name + " and I'm a " + this.occupation;
6    }
7  }
8
9  >>> var h1 = new Hero('Michelangelo');
10 >>> var h2 = new Hero('Donatello');
11 >>> h1.whoAreYou();
12 "I'm Michelangelo and I'm a Ninja"
13 >>> h2.whoAreYou();
14 "I'm Donatello and I'm a Ninja"
```

Todos los entornos cliente tienen un **objeto global** y todas las variables globales son propiedades de este objeto global

En el navegador este objeto global se llama `window`

Por lo tanto, podemos acceder a una variable global `a`:

- Como una variable `a`
- Como una propiedad del objeto global: `window['a']` o `window.a`

Si declaramos una función constructora y la llamamos sin `new`

- Devolverá `undefined`
- Todas las propiedades declaradas con `this` se convertirán en propiedades de `window`

```
>>> function Hero(name) {this.name = name;}
>>> var h = Hero('Leonardo');
>>> typeof h
"undefined"
>>> typeof h.name
h has no properties

>>> name
"Leonardo"
>>> window.name
"Leonardo"

>>> var h2 = new Hero('Michelangelo');
>>> typeof h2
"object"
>>> h2.name
"Michelangelo"
```

Hay maneras de evitar estos "accidentes" (llamar a un constructor sin `new`) como por ejemplo activar `strict mode` (lanzaria una excepcion en este caso).

Cuando creamos un objeto, se le asigna siempre la propiedad `constructor` que contiene una referencia a la función constructora utilizada para crear el objeto

```
>>> h2.constructor
Hero(name)

>>> var h3 = new h2.constructor('Rafaello');
>>> h3.name;
"Rafaello"

>>> var o = {};
>>> o.constructor;
Object()
>>> typeof o.constructor;
"function"
```

Con el operador `instanceof` podemos chequear si un objeto fue creado con una determinada función constructora

```
>>> function Hero(){}
>>> var h = new Hero();
>>> var o = {};
>>> h instanceof Hero;
true
>>> h instanceof Object;
false
>>> o instanceof Object;
true
```

13.2 Trabajando con Objetos

Otra forma de crear un objeto, es a través de una función que nos devuelva un objeto.

```
function factory(name) {
  return {
    name: name
  };
}
>>> var o = factory('one');
>>> o.name
"one"
>>> o.constructor
Object()
```

Podemos utilizar funciones constructoras y devolver objetos distintos de `this`

```
>>> function C() { this.a = 1; }
>>> var c = new C();
>>> c.a
1
>>> function C2() { this.a = 1; return {b: 2}; }
>>> var c2 = new C2();
>>> typeof c2.a
"undefined"
>>> c2.b
2
```

Cuando *copiamos* un objeto o *lo pasamos como parámetro* a una función, realmente estamos pasando una referencia al objeto.

Si hacemos un cambio a esta referencia, modicaremos también el objeto original

```
>>> var original = { howmany: 1 };
>>> var copy = original;
>>> copy.howmany
1
>>> copy.howmany = 100;
100
>>> original.howmany
100

>>> var nullify = function(o) { o.howmany = 0; }
>>> nullify(copy);
>>> original.howmany
0
```

Cuando *comparamos* un objeto sólo obtendremos `true` si comparamos 2 referencias al mismo objeto

```
>>> var fido = { breed: 'dog' };
>>> var benji = { breed: 'dog' };
>>> benji === fido
false
>>> benji == fido
false
>>> var mydog = benji;
>>> mydog === benji
true
>>> mydog === fido
false
```

14. Objetos Globales

Son los objetos que tenemos disponibles en el ámbito global (objetos primitivos)

Los podemos dividir en 3 grupos

- *Objetos contenedores de datos*: Object, Array, Function, Boolean, Number
- *Objetos de utilidades*: Math, Date, RegExp
- *Objetos de errores*: Error

14.1 Object

Object es el padre de todos los objetos Javascript, es decir, cualquier objeto hereda de él

Para crear un objeto vacio podemos usar:

- La notacion literal : `var o = {}`
- La funcion constructora Object(): `var o = new Object();`

Un objeto contiene las siguientes propiedades y metodos:

- La propiedad `o.constructor` con la función constructora
- El método `o.toString()` que devuelve el objeto en formato texto
- El método `o.valueOf()` que devuelve el valor del objeto (normalmente o)

```
>>> var o = new Object();
>>> o.toString()
“[object Object]”

>>> o.valueOf() === o
true
```

14.2 Array

Para crear arrays podemos usar:

- La notacion literal : `var a = []`
- La funcion constructora Array(): `var o = new Array();`

Podemos pasarle parametros al constructor `Array()`

- Varios parametros: Seran asignados como elementos al array
- Un numero: Se considerará el tamaño del array

```
>>> var a = new Array(1,2,3,'four');
>>> a;
[1, 2, 3, "four"]

>>> var a2 = new Array(5);
>>> a2;
[undefined, undefined, undefined, undefined, undefined]
```

Como los arrays son objetos tenemos disponibles los metodos y propiedades del padre `Object()`

```
>>> typeof a;
"object"

>>> a.toString();
"1,2,3,four"
>>> a.valueOf()
[1, 2, 3, "four"]
>>> a.constructor
Array()
```

Los arrays disponen de la propiedad `length`

- Nos devuelve el tamaño del array (numero de elementos)
- Podemos modificarlo y cambiar el tamaño del array

```
>>> a[0] = 1;
>>> a.prop = 2;

>>> a.length
1
>>> a.length = 5
5
>>> a
[1, undefined, undefined, undefined, undefined]

>>> a.length = 2;
2
>>> a
[1, undefined]
```

Los arrays disponen de unos cuantos metodos interesantes:

- `push()`

- pop()
- sort()
- join()
- slice()
- splice()

```
>>> var a = [3, 5, 1, 7, 'test'];

>>> a.push('new')
6
>>> a
[3, 5, 1, 7, "test", "new"]

>>> a.pop()
"new"
>>> a
[3, 5, 1, 7, "test"]

>>> var b = a.sort();
>>> b
[1, 3, 5, 7, "test"]
>>> a
[1, 3, 5, 7, "test"]

>>> a.join(' is not ');
"1 is not 3 is not 5 is not 7 is not test"

>>> b = a.slice(1, 3);
[3, 5]
>>> a
[1, 3, 5, 7, "test"]

>>> b = a.splice(1, 2, 100, 101, 102);
[3, 5]
>>> a
[1, 100, 101, 102, 7, "test"]
```

push()

`push()` inserta elementos al final del array
`a.push('new')` es lo mismo que `a[a.length] = 'new'`
`push()` devuelve el tamaño del array modificado

```
>>> var sports = ['soccer', 'baseball'];
>>> sports
["soccer", "baseball"]
>>> sports.length
2
>>> sports.push('football', 'swimming');
4
>>> sports
["soccer", "baseball", "football", "swimming"]
```

pop()

`pop()` elimina el ultimo elemento
`a.pop()` es lo mismo que `a.length--`;
`pop()` devuelve el elemento eliminado

```
>>> var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];
>>> myFish
["angel", "clown", "mandarin", "sturgeon"]
>>> myFish.pop();
"sturgeon"
>>> myFish
["angel", "clown", "mandarin"]
```

sort()

ordena el array y devuelve el array modificado

```
>>> var fruit = ['apples', 'bananas', 'Cherries'];
>>> fruit
["apples", "bananas", "Cherries"]
>>> fruit.sort();
["Cherries", "apples", "bananas"]

>>> var scores = [1, 2, 10, 21];
>>> scores
[1, 2, 10, 21]
>>> scores.sort()
[1, 10, 2, 21]
```

```
>>> var numbers = [4, 2, 42, 36, 5, 1, 12, 3];
>>> numbers
[4, 2, 42, 36, 5, 1, 12, 3]
>>> numbers.sort()
[1, 12, 2, 3, 36, 4, 42, 5]
>>> numbers.sort( function(a, b) { return a - b; } );
[1, 2, 3, 4, 5, 12, 36, 42]
```

join()

`join()` devuelve una cadena (string) con los valores de los elementos del array

```
>>> var a = ['Wind', 'Rain', 'Fire'];
>>> a.join();
"Wind,Rain,Fire"
>>> a.join(" - ");
"Wind - Rain - Fire"
>>> typeof ( a.join(" - ") )
"string"
```

slice()

`slice()` devuelve un trozo del array sin modficar el original

```
>>> var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
>>> var citrus = fruits.slice(1, 3);
>>> fruits
["Banana", "Orange", "Lemon", "Apple", "Mango"]
>>> citrus
["Orange", "Lemon"]
```

splice()

`splice()` quita un trozo del array, lo devuelve y opcionalmente rellena el hueco con nuevos elementos

```
>>> var myFish = ['angel', 'clown', 'mandarin', 'surgeon'];

>>> myFish
["angel", "clown", "mandarin", "surgeon"]
>>> var removed = myFish.splice(2, 1);

>>> myFish
["angel", "clown", "surgeon"]
>>> removed
["mandarin"]

>>> var removed = myFish.splice(2, 0, 'drum');
>>> myFish
["angel", "clown", "drum", "surgeon"]
>>> removed
[]
```

14.3 Function

Las funciones son objetos

Podemos crear funciones con la función constructora `Function()` (aunque este metodo no se recomienda ya que internamente hace un *eval()*)

```
>>> function sum(a, b) {return a + b;};
>>> sum(1, 2)
3
>>> var sum = function(a, b) {return a + b;};
>>> sum(1, 2)
3
>>> var sum = new Function('a', 'b', 'return a + b;');
>>> sum(1, 2)
3
```

Las funciones disponen de las siguientes propiedades:

- La propiedad `constructor` que contiene una referencia a la funcion constructora *Function()*
- La propiedad `length` que contiene el numero de parametros que acepta la función
- La propiedad `caller` (no standard) que contiene una referencia a la funcion que llamó a esta función
- La propiedad `prototype` que contiene un objeto.
  - Sólo es util cuando utilizamos una funcion como constructora.
  - Todos lo objetos creados con una función constructora mantienen una referencia a su propiedad `prototype` y pueden usar sus propiedades como si fueran propias.

```
>>> function myfunc(a){ return a; }
>>> myfunc.constructor
Function()

>>> function myfunc(a, b, c){ return true; }
>>> myfunc.length
3

>>> function A(){ return A.caller; }
>>> function B(){ return A(); }
>>> B()
B()
```

```
var some_obj = {
  name: 'Ninja',
  say: function(){
    return 'I am a ' + this.name;
  }
}
>>> function F(){}
>>> typeof F.prototype
"object"
>>> F.prototype = some_obj;
>>> var obj = new F();
>>> obj.name
"Ninja"
>>> obj.say()
"I am a Ninja"
```

Las funciones disponen de los siguientes métodos:

- El método `toString()` que devuelve el código fuente de la función
- Los métodos `call()` y `apply()` que ejecutan metodos de otros objetos especificando el contexto (especificamos un `this` diferente)
  - Estos dos métodos hacen lo mismo pero el formato en que reciben los argumentos es diferente

```
>>> function myfunc(a, b, c) {return a + b + c;}
>>> myfunc.toString()
"function myfunc(a, b, c) {
return a + b + c;
}"
```

```
var some_obj = {
  name: 'Ninja',
  say: function(who){
    return 'Haya ' + who + ', I am a ' + this.name;
  }
}
>>> some_obj.say('Dude');
"Haya Dude, I am a Ninja"

>>> my_obj = {name: 'Scripting guru'};
>>> some_obj.say.call(my_obj, 'Dude');
"Haya Dude, I am a Scripting guru"

some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

Las funciones disponen del objeto `arguments` que (ademas de `length`) tiene la propiedad `callee` que contiene una referencia a la funcion llamada (a si misma)

```
>>> function f(){return arguments.callee;}
>>> f()
f()
```

```
(
  function(count){
    if (count <= 5) {
      console.log(count);
      arguments.callee(++count);
    }
  }
)(1)
```

14.4 Boolean

El objeto *Boolean* es un contenedor para un valor de tipo booleano

Podemos crear objetos *Boolean* con la función constructora `Boolean()`

```
>>> var b = new Boolean();
>>> typeof b
"object"
>>> typeof b.valueOf()
"boolean"
>>> b.valueOf()
false
```

La función `Boolean` usada como función normal (sin `new`) nos devuelve el valor pasado como parametro convertido a booleano

```
>>> Boolean("test")
true
>>> Boolean("")
false
>>> Boolean({})
true
```

14.5 Number

La función `Number()` puede ser usada:

- Cómo una *función normal* para convertir valores a número
- Cómo una *función constructora* (con `new`) para crear objetos

Los objetos número disponen de los métodos: `toFixed()`, `toPrecision()` y `toExponential()`

```
>>> var n = new Number(123.456)
>>> n.toFixed(1)
"123.5"

>>> (12345).toExponential()
"1.2345e+4"
```

El método `toString()` de un objeto numero nos permite transformar un numero a una base determinada

```
>>> var n = new Number(255);
>>> n.toString();
"255"
>>> n.toString(10);
"255"
>>> n.toString(16);
"ff"
>>> (3).toString(2);
"11"
>>> (3).toString(10);
"3"
```

14.6 String

Podemos crear objetos String con la función constructora `String()`

Un objeto *String* NO es un dato de tipo primitivo string (`valueOf()`)

```
>>> var primitive = 'Hello';
>>> typeof primitive;
"string"
>>> var obj = new String('world');
>>> typeof obj;
"object"

>>> Boolean("")
false
>>> Boolean(new String(""))
true
```

Un objeto string es parecido a un array de caracteres:

- Cada carácter tiene una posición indexada
- Tiene disponible la propiedad length

```
>>> obj[0]
"w"
>>> obj[4]
"d"
>>> obj.length
5
```

Aunque los métodos pertenezcan al objeto String, podemos utilizarlos también directamente en datos de tipo primitivo string (se crea el objeto internamente)

```
>>> "potato".length
6
>>> "tomato"[0]
"t"
>>> "potato"["potato".length - 1]
"o"
```

Los objetos string disponen de los siguientes métodos:

- `toUpperCase()` devuelve el string convertido a mayúsculas
- `toLowerCase()` devuelve el string convertido a minusculas
- `charAt()` devuelve el carácter encontrado en la posición indicada
- `indexOf()` busca una cadena de texto en el string y devuelve la posición donde la encuentra
  - Si no encuentra nada devuelve -1
- `lastIndexOf()` empieza la búsqueda desde el final de la cadena
  - Si no encuentra nada devuelve -1

Por tanto la manera de correcta de chequear si existe una cadena de texto en otra es → `if ( s.toLowerCase().indexOf('couch') !== -1 ) {...}`

- `slice()` devuelve un trozo de la cadena de texto
- `split()` transforma el string en un array utilizando un string como separador
- `concat()` une strings

```
>>> var s = new String("Couch potato");
>>> s.toUpperCase()
"COUCH POTATO"
>>> s.toLowerCase()
"couch potato"
>>> s.charAt(0);
"C"
>>> s.indexOf('o')
1
>>> s.lastIndexOf('o')
11
>>> s.slice(1, 5)
"ouch"
>>> s.split(" ")
["Couch", "potato"]
>>> s.concat("es")
"Couch potatoes"
```

14.7 Math

Math es un objeto con propiedades y métodos para usos matemáticos No es constructor de otros objetos

Algunos métodos interesantes de Math son:

- `random()` genera números aleatorios entre 0 y 1
- `round()`, `floor()` y `ceil()` para redondear numeros
- `min()` y `max()` devuelven el minimo y el máximo de una serie de números pasados como parametros
- `pow()` y `sqrt()` devuelve la potencia y la raíz cuadrada respectivamente

14.8 Date

`Date()` es una función constructora que crea objetos Date
Podemos crear objetos Date nuevo pasándole:

- *Nada* (tomará por defecto la fecha actual)
- *Una fecha* en formato texto
- *Valores separados* que representan: Año, Mes (0-11), Dia (1-31), Hora (0-23), Minutes (0-59), Segundos(0-59) y Milisegundos (0-999)
- Un valor *timestamp*

Ejemplo (Firebug muestra el resultado del metodo toString sobre un objeto date):

```
>>> new Date()
Tue Jan 08 2008 01:10:42 GMT-0800 (Pacific Standard Time)
>>> new Date('2009 11 12')
Thu Nov 12 2009 00:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date(2008, 0, 1, 17, 05, 03, 120)
Tue Jan 01 2008 17:05:03 GMT-0800 (Pacific Standard Time)
>>> new Date(1199865795109)
Wed Jan 09 2008 00:03:15 GMT-0800 (Pacific Standard Time)
```

Algunos métodos para trabajar con objetos Date son:

- `setMonth()` y `getMonth()` escriben y leen el mes en un objeto date respectivamente (lo mismo hay para year, day, hours, minutes, etc...)
- `parse()` dado un string, devuelve su timestamp
- `UTC()` produce un timestamp dados un año, mes, dia, etc..
- `toDateString()` devuelve el contenido de un objeto date en formato americano

```
>>> var d = new Date();
>>> d.toString();
"Wed Jan 09 2008 00:26:39 GMT-0800 (Pacific Standard Time)"
>>> d.setMonth(2);
1205051199562
>>> d.toString();
"Sun Mar 09 2008 00:26:39 GMT-0800 (Pacific Standard Time)"
>>> d.getMonth();
2

>>> Date.parse('Jan 1, 2008')
1199174400000
>>> Date.UTC(2008, 0, 1)
1199145600000

>>> var d = new Date(2012, 5, 20);
>>> d.getDay();
3
>>> d.toDateString();
"Wed Jun 20 2012"
```

15. El entorno del Navegador

> → El gran lío: versiones, DOM y BOM
> → JavaScript Vs DOM Vs BOM, relationship explained | Rajakvk's Blog
> → Browser Object Model and Document Object Model | JavaScript By Example
> → What is the DOM and BOM in JavaScript? | StackOverflow

Javascript puede ser utilizado en diferentes entornos, pero su entorno más habitual es el navegador

El código Javascript de una pagina tiene acceso a unos cuantos objetos. Estos objetos los podemos agrupar en:

- Objetos que tienen relación con la pagina cargada (el document). Estos objetos conforman el **Document Object Model (DOM)**
- Objetos que tienen que ver con cosas que están fuera de la pagina (la ventana del navegador y la pantalla). Estos objetos conforman el **Browser Object Model (BOM)**

El DOM es un standard y tiene varias versiones (llamadas levels). La mayoria de los navegadores implementan casi por completo el DOM Level 1.

El BOM no es un standard, asi que algunos objetos están soportados por la mayoría de navegadores y otros solo por algunos.

15.1 Deteccion de Funcionalidades

Debido a estas diferencia entre navegadores surge la necesidad de averiguar (desde código JS) que caracteristicas soporta nuestro navegador (DOM y BOM)

Una solución seria la llamada **Browser Sniffing** que consiste en detectar el navegador que estamos utilizando

Esta técnica no se recomienda por:

- Hay demasiados navegadores para controlar
- Dificil de mantener (surgen nuevas versiones y nuevos navegadores)
- El parseo de cadenas puede ser complicado y no es fiable del todo

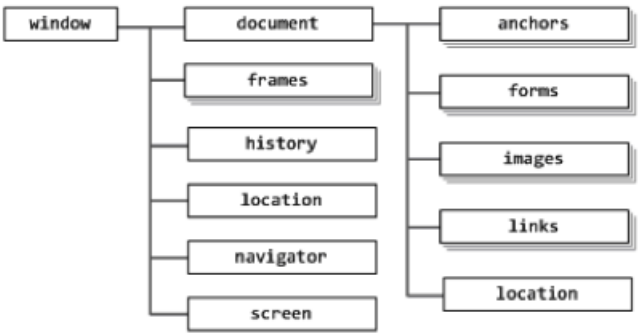
```
if (navigator.userAgent.indexOf('MSIE') !== -1) {
    // this is IE
} else {
    // not IE
}
```

La mejor solucion para detectar funcionalidades de nuestro navegador es hacer **Feature Sniffing**, es decir chequear la existencia del objeto (método, array o propiedad) que queremos utilizar

```
if (typeof window.addEventListener === 'function') {
    // feature is subported, let's use it
} else {
    // hmm, this feature is not subported, will have to
    // think of another way
}
```

16. BOM

El **BOM (Browser Object Model)** lo conforman todos los objetos que están fuera del documento cargado (document) y forman parte del objeto window



BOM

El objeto `window` ademas de servir de contenedor de las variables globales y de ofrecer los metodos nativos de JS (`window.parseInt`), contiene informacion sobre el entorno del navegador (frame, iframe, popup, ventana o pestaña)

16.1 Propiedades de `window`

Algunos de los *objetos* que tenemos disponibles en window son:

`window.navigator`

Es un objeto que contiene información sobre el navegador

```
>>> window.navigator.userAgent
"Mozilla/5.0 (Windows; U; Windows NT 5.1; es-ES; rv:1.9.2.12)
Gecko/20101026 Firefox/3.6.12 ( .NET CLR 3.5.30729)"
```

`window.location`

Es un objeto que contiene info (y metodos) sobre la URL actual

```
>>> window.location.href = 'http://www.packtpub.com'
>>> location.href = 'http://www.packtpub.com'
>>> location = 'http://www.packtpub.com'
>>> location.assign('http://www.packtpub.com')
>>> location.reload()
>>> window.location.href = window.location.href
>>> location = location
```

window.history

Es un objeto que contiene el historial de paginas visitadas y tiene métodos para movernos en él (sin poder ver las URL's)

```
>>> window.history.length
5
>>> history.forward()
>>> history.back()
>>> history.go(-2);
```

window.frames

Es una colección de todos los frames que tenemos en la página
Cada frame tendrá su propio objeto window
Podemos utilizar `parent` para acceder desde el frame hijo al padre
Con la propiedad `top` accedemos a la pagina que está por encima de todos los frames
Podemos acceder a un frame concreto por su nombre.

```
<iframe name="myframe" src="about:blank" />
>>> window.frames[0]
>>> frames[0].window.location.reload()
>>> frames[0].parent === window
true
>>> window.frames[0].window.top === window
true
>>> self === window
true
>>> window.frames['myframe'] === window.frames[0]
true
```

window.screen

Ofrece info sobre la pantalla (general, fuera del browser)

```
>>> window.screen.colorDepth
32
>>> screen.width
1440
>>> screen.availWidth
1440
>>> screen.height
900
>>> screen.availHeight
847
```

16.2 Métodos de `window`

Algunos de los *métodos* que tenemos disponibles en `window` son:

`window.open()`, `window.close()`

Nos permiten abrir (y cerrar) nuevas ventanas (popups)
`window.open()` devuelve una referencia a la ventana creada (si devuelve false es que no la ha podido crear - popups blocked)
No se recomienda su uso ;-)

```
>>> var win = window.open('http://www.packtpub.com', 'packt',
'width=300,height=300,resizable=yes');
>>> win.close()
```

`window.moveTo()`, `window.moveBy()`, `window.resizeTo()`, `window.resizeBy()`

Nos permiten mover y redimensionar las ventanas
No se recomienda su uso ;-)

```
>>> window.moveTo(100, 100)
>>> window.moveBy(10, -10)
>>> window.resizeTo(300, 300)
>>> window.resizeBy(20, 10)
```

`window.alert()`, `window.prompt()`, `window.confirm()`

Nos permiten interactuar con el usuario a traves de mensajes del sistema

```
if (confirm('Are you sure you want to delete this item?')) {
// delete
} else {
// abort
}
>>> var answer = prompt('And your name was?');
console.log(answer);
```

window.setTimeout(), window.setInterval()

Nos permiten ejecutar código después de un intervalo de tiempo (y en su caso, repetirlo)

```
>>> function boo(){alert('Boo!');}
>>> setTimeout(boo, 2000);
>>> var id = setTimeout(boo, 2000);
>>> clearTimeout(id);
>>> function boo() { console.log('boo') };
>>> var id = setInterval( boo, 2000 );
boo
boo
boo
>>> clearInterval(id)
var id = setInterval( "alert('boo, boo')", 2000 );
var id = setInterval( function(){ alert('boo, boo')}, 2000 );
```

16.3 El objeto 'document'

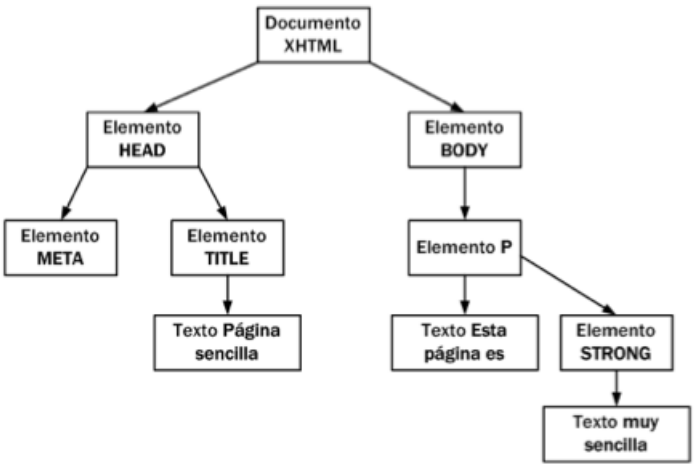
`window.document` es un objeto del BOM con info sobre el documento actual

Todos los métodos y propiedades que estan dentro de `window.document` pertenecen a la categoría de objetos DOM

17. DOM

El **DOM (Document Object Model)** es una forma de representar un documento HTML (o XML) como un árbol de nodos.
Utilizando los métodos y propiedades del DOM podremos acceder a los elementos de la página, modificarlo, eliminarlos o añadir nuevos

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-\
1" />
        <title>Página sencilla</title>
    </head>
    <body>
        <p>Esta página es <strong>muy sencilla</strong></p>
    </body>
</html>
```



DOM

En el *DOM Level 1* se diferencia entre:

- El **Core DOM** es la especificación común que se aplica a todos los documentos (XML, HTML,...)
- El **Core HTML** es la especificación que se aplica sólo a documentos HTML

17.1 Accediendo a los nodos

```
<body>
    <p class="opener">first paragraph</p>
    <p><em>second</em> paragraph</p>
    <p id="closer">final</p>
    <!-- and that's about it -->
</body>
```

El nodo `document` nos da acceso al documento (es el punto de partida)

Todos los nodos tienen las propiedades:

- `nodeType`: Hay 12 tipos de nodos representados por números (1=element, 2=attribute, 3=text, ...)
- `nodeName`: Para tags HTML es el nombre del tag y para nodos texto es #text
- `nodeValue`: Para nodos de texto el valor será el texto

El nodo `documentElement` es el nodo raíz. Para documentos HTML es el tag `<html>`

```
>>> document.documentElement
<html>
>>> document.documentElement.nodeType
1
>>> document.documentElement.nodeName
"HTML"
>>> document.documentElement.tagName
"HTML"
```

Cada nodo puede tener nodos-hijo:

- `hasChildNodes()` : Este método devolverá true si el nodo tiene nodos-hijo
- `childNodes`: Devuelve en un array los nodos-hijo de un elemento.
  Al ser un array podemos saber el numero de nodos-hijo con `childNodes.length`
- `parentNode`: Nos da el nodo-padre de un nodo-hijo

```
>>> document.documentElement.hasChildNodes()
True
>>> document.documentElement.childNodes.length
2
>>> document.documentElement.childNodes[0]
<head>
>>> document.documentElement.childNodes[1]
<body>
>>> document.documentElement.childNodes[1].parentNode
<html>
>>> var bd = document.documentElement.childNodes[1];
>>> bd.childNodes.length
9
```

Podemos chequear la existencia de attributes y acceder a sus atributos:

- `hasAttributes()`:Devuelve true si el elemento tiene atributos
- `getAttribute()`: Devuelve el contenido de un atributo

```
>>> bd.childNodes[1]
<p class="opener">
>>> bd.childNodes[1].hasAttributes()
True
>>> bd.childNodes[1].attributes.length
1
>>> bd.childNodes[1].attributes[0].nodeName
"class"
>>> bd.childNodes[1].attributes[0].nodeValue
"opener"
>>> bd.childNodes[1].attributes['class'].nodeValue
"opener"
>>> bd.childNodes[1].getAttribute('class')
"opener"
```

Podemos acceder al contenido de un tag:

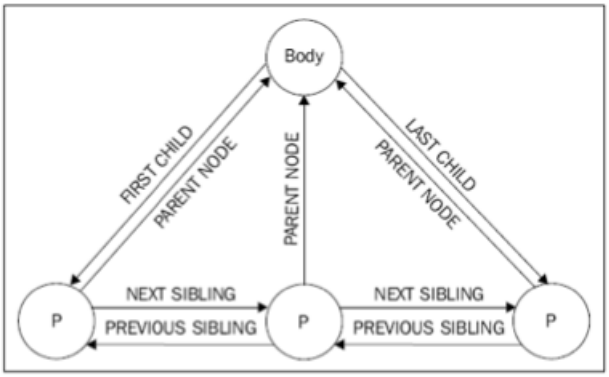
- `textContent`: Esta propiedad nos da el texto plano dentro de una etiqueta
  En IE no existe esta propiedad (hay que usar innerText)
- `innerHTML`: Esta propiedad nos da el contenido (en HTML) de un tag

```
>>> bd.childNodes[1].nodeName
"P"
>>> bg.childNodes[1].textContent
"first paragraph"
>>> bd.childNodes[1].innerHTML
"first paragraph"
>>> bd.childNodes[3].innerHTML
"<em>second</em> paragraph"
>>> bd.childNodes[3].textContent
"second paragraph"
>>> bd.childNodes[1].childNodes.length
1
>>> bd.childNodes[1].childNodes[0].nodeName
"#text"
>>> bd.childNodes[1].childNodes[0].nodeValue
"first paragraph"
```

Podemos acceder directamente a algunos elementos sin necesidad de recorrer todo elárbol:

- `getElementsByTagName()`: Nos devuelve un array con todos los elementos con el tag que se le pasa por parámetro
- `getElementsByName()`: Nos devuelve un array con todos los elementos con el name que se le pasa por parámetro
- `getElementById()`: No devuelve el elemento con el id que se le pasa por parámetro

```
>>> document.getElementsByTagName('p').length
3
>>> document.getElementsByTagName('p')[0]
<p class="opener">
>>> document.getElementsByTagName('p')[0].innerHTML
"first paragraph"
>>> document.getElementsByTagName('p')[2]
<p id="closer">
>>> document.getElementsByTagName('p')[2].id
"closer"
>>> document.getElementsByTagName('p')[0].className
"opener"
>>> document.getElementById('closer')
<p id="closer">
```



Parent & Childs

Desde un nodo también podemos acceder a sus hermanos y al primero y último de sus hijos

- `nextSibling`: Nos devuelve el siguiente hermano
- `previousSibling`: Nos devuelve el anterior hermano
- `firstChild`: Nos devuelve el primer hijo
- `lastChild`: Nos devuelve el último hijo

```
>>> var para = document.getElementById('closer')
>>> para.nextSibling
"\n "
>>> para.previousSibling
"\n "
>>> para.previousSibling.previousSibling
<p>
>>> para.previousSibling.previousSibling.previousSibling
"\n "
>>>
para.previousSibling.previousSibling.nextSibling.nextSibling
<p id="closer">
>>> document.body.previousSibling
<head>
>>> document.body.firstChild
"\n "
>>> document.body.lastChild
"\n "
>>> document.body.lastChild.previousSibling
Comment length=21 nodeName=#comment
>>> document.body.lastChild.previousSibling.nodeValue
" and that's about it "
```

17.2 Modificando los nodos

Para cambiar el contenido de una etiqueta cambiamos el contenido de innerHTML

```
>>> var my = document.getElementById('closer');
>>> my.innerHTML = '<em>my</em> final';
>>> my.firstChild
<em>
>>> my.firstChild.firstChild
"my"
>>> my.firstChild.firstChild.nodeValue = 'your';
"your"
```

Los elementos tienen la propiedad `style` que podemos utilizar para modificar sus estilos

```
>>> my.style.border = "1px solid red";
"1px solid red"
· Ademas podemos modificar los atributos existan ya o no
Ejemplo:
>>> my.align = "right";
"right"
>>> my.name
>>> my.name = 'myname';
"myname"
>>> my.id
"closer"
>>> my.id = 'further'
"further"
```

17.3 Creando y Eliminando nodos

Para crear nuevos elementos podemos utilizar los métodos `createElement` y `createTextNode`.

Una vez creados los podemos añadir al DOM con `appendChild`

```
>>> var myp = document.createElement('p');
>>> myp.innerHTML = 'yet another';
"yet another"
>>> myp.style
CSSStyleDeclaration length=0
>>> myp.style.border = '2px dotted blue'
"2px dotted blue"
>>> document.body.appendChild(myp)
<p style="border: 2px dotted blue;">
```

Tambien podemos copiar elementos existentes con `cloneNode()`

`cloneNode` acepta un parámetro booleano (true copiará el nodo con todos sus hijos y false solo el nodo)

```
>>> var el = document.getElementsByTagName('p')[1];
<p><em>second</em> paragraph</p>
>>> document.body.appendChild(el.cloneNode(false))
>>> document.body.appendChild(document.createElement('p'));
>>> document.body.appendChild(el.cloneNode(true))
```

Con `insertBefore()` podemos especificar el elemento delante del cual queremos insertar el nuestro

```
document.body.insertBefore(
    document.createTextNode('boo!'),
    document.body.firstChild
);
```

Para eliminar nodos del DOM podemos utilizar `removeChild()` o `replaceChild()`

`removeChild()` elimina el elemento y replaceChild() lo sustituye por otro que se le pasa como parámetro

Tanto `replaceChild()` como `removeChild()` devuelven el nodo eliminado

```
>>> var myp = document.getElementsByTagName('p')[1];
>>> var removed = document.body.removeChild(myp);
>>> removed
<p>
>>> removed.firstChild
<em>
>>> var p = document.getElementsByTagName('p')[1];
>>> p
<p id="closer">
>>> var replaced = document.body.replaceChild(removed, p);
>>> replaced
<p id="closer">
```

17.4 Objetos DOM sólo de HTML

En el DOM tenemos disponibles una serie de selectores directos y de colecciones exclusivos de HTML (no XML):

- `document.body`: `document.getElementsByTagName('body')[0]`
- `document.images`: `document.getElementsByTagName('img')`
- `document.applets`: `document.getElementsByTagName('applet')`
- `document.links`: Nos devuelve un array con todos los links con atributo `href`
- `document.anchors`: Nos devuelve un array con todos los links con atributo name
- `document.forms`: `document.getElementsByTagName('form')`
  Podemos acceder a los elementos del form (inputs, buttons) con elements

```
>>> document.forms[0]
>>> document.getElementsByTagName('forms')[0]
>>> document.forms[0].elements[0]
>>> document.forms[0].elements[0].value = 'me@example.org'
"me@example.org"
>>> document.forms[0].elements[0].disabled = true;
>>> document.forms[0].elements['search']; // array notation
>>> document.forms[0].elements.search; // object property
```

Tambien tenemos disponible el método `document.write()`
No se recomienda su uso ;-)

Algunas propiedades del objeto document son:

- `document.cookies`: Contiene una cadena de texto con las cookies asociadas al documento
- `document.title`: Permite cambiar el titulo de la pagina que aparece en el navegador
  Esto no cambia el contenido del tag title
- `document.referrer`: Contiene la URL desde donde hemos llegado a la página
- `document.domain`: Contiene el dominio de la pagina

17.5 Selección Avanzada de Elementos

> → CSS2: Syntax and basic data types | w3.org
> → CSS contents and browser compatibility | quirksmode.org

`document.images`

```
`document.getElementsByTagName('img')`
```

`rows` y `cells`

Dado un elemento table podemos acceder a sus filas y dado un row podemos acceder a sus celdas con estos selectores

```
>>> oTable = document.getElementsByTagName('table')[0];
>>> aRows = oTable.rows;n
>>> oFirstRow = aRows[0];
>>> oLastRow = aRows[oRows.length-1];
>>> aCells = oFirstRow.cells;
>>> oFirstCell = aCells[0];
>>> oLastCell = aCells[aCells.length-1];
```

options

Dado un elemento `select` podemos acceder al array de sus `options`

```
>>> document.getElementsByTagName('select')[0].options;
[option.windows, option.movil, option.aplicaciones-web, option.mac,
option.linux, option.palm, option.pocketpc, option.blog]
```

querySelector y querySelectorAll

Devuelve elementos del DOM a partir de una selección CSS

- `querySelector()` devuelve el primer elemento encontrado
- `querySelectorAll()` devuelve un array de elementos

Funciones nativas disponibles a partir de IE8 y FF3.5

```
>>> oMyElem = document.querySelector("#myid");
>>> aMyHiddenElems = document.body.querySelectorAll(".hidden");
```

$() o jQuery()

Con jQuery disponemos de una potente herramienta de selección de elementos

> → How jQuery selects elements using Sizzle | blog.bigbinary.com
> → jQuery Selectors | refcardz.dzone.com

Para obtener los elementos utilizamos `$()` o `jQuery()` pasándole nuestra selección CSS entre comillas

`$()` devuelve un objeto jQuery (que no es un elemento DOM y tiene acceso a métodos propios de jQuery)

Podemos pasar de objeto jQuery a selección DOM:

- Para un elemento `$('#container') -> $('#container')[0]`
- Para un grupo de elementos `$('.hidden') -> $('.hidden').toArray()`

Tambien podemos pasar de selección DOM a objeto jQuery:

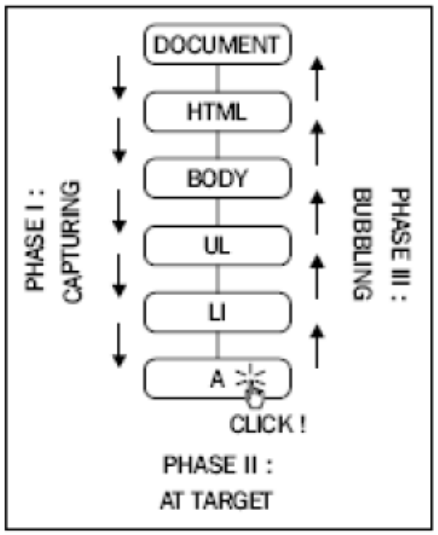
- Para un elemento: `document.getElementById('container') -> $(document.getElementById('containe`
- Para un grupo de elementos: `document.links -> $(document.links);`

```
>>> $('#container');
jQuery(div#container)
>>> $('#container')[0]

<div id="container">
>>> $('#footer_contents')
jQuery(div#footer_contents.clearfix)
>>> $('#footer_contents')[0]

<div id="footer_contents" class="clearfix">
>>> $('#footer_contents').attr("class");
"clearfix"
>>> $('#footer_contents').className
undefined
>>> $('#footer_contents')[0].className
"clearfix"
>>> $('#footer_contents')[0].attr("class");
 TypeError: $("#footer_contents")[0].attr is not a function

 >>> $('div.hidden')
 jQuery(div#ads_section_textlinks.clearfix, div#top_sales.top_box,
 div#top_valuated.top_box, div.list_container, div.ac_results)

 >>> $('div.hidden').toArray()
 [div#ads_section_textlinks.clearfix, div#top_sales.top_box,
 div#top_valuated.top_box, div.list_container, div.ac_results]

 >>> $('div.hidden').toArray()[0]
 <div id="ads_section_textlinks" class="clearfix
 hidden" style="display: block;">

 >>> document.getElementById('ads_section_textlinks');
 <div id="ads_section_textlinks" class="clearfix
 hidden" style="display: block;">

 >>> $(document.getElementById('ads_section_textlinks'));
```

```
 jQuery(div#ads_section_textlinks.clearfix)

 >>> $(document.querySelectorAll('div.hidden')[0]);
 jQuery(div#ads_section_textlinks.clearfix)
```

18. Eventos

Cada acción (*click*, *change*, ...) que ocurre en el navegador es comunicada (a quien quiera escuchar) en forma de **evento** Desde Javascript podemos *escuchar* estos eventos y engancharle una función (*event handler*) que se ejecutará cuando ocurra este evento



Events

Cuando hacemos click en un link (`a`), tambien hacemos click en su contenedor (`li`,`ul`), en el `body` y en ultima instancia en el `document`. Esto es lo que se llama la **propagación del evento**

La especificación de eventos DOM Level 2 define que el evento se propaga en 3 fases: *Capturing*, *Target* y *Bubbling*

- **Capturing**: El click ocurre en el document y va pasando por todos los elementos hasta llegar al link (a)
- **Bubbling**: El click ocurre en el link (a) y va emergiendo hasta el document.

Firefox, Opera y Safari implementan las 3 fases pero IE sólo el Bubbling

18.1 Capturando eventos

→ Flexible Javascript Events | John Resig blog
→ JavaScript Events | webmonkey.com
→ Events and Event Handlers | elated.com
→ Gestión de eventos en Javascript | anieto2k.com

```
function callback(evt) {
    // prep work
    evt = evt || window.event;
    var target = (typeof evt.target !== 'undefined') ? evt.target :
    evt.srcElement;

    // actual callback work
    console.log(target.nodeName);
}
// start listening for click events
if (document.addEventListener){ // FF
    document.addEventListener('click', callback, false);
} else if (document.attachEvent){ // IE
    document.attachEvent('onclick', callback);
} else {
    document.onclick = callback;
}
```

Modelo Tradicional

Podemos capturar eventos con el *modelo tradicional*

Este modelo consiste en asignar una función a la propiedad `onclick`, `onchange`,... del elemento del DOM

```
<button onclick="myFunction()">Click me</button>
<button onclick="document.bgColor='lightgreen'">Click me</button>
<button onclick="getElementById('demo').innerHTML=Date()">What is the time?</\
button>

<A HREF="#" onClick="alert('Hello out there!')";>Some Text</A>

window.onclick = myFunction;
// If the user clicks in the window, set the background color of <body> to ye\
llow
function myFunction() {
    document.getElementsByTagName("BODY")[0].style.backgroundColor = "yellow";
}
```

Con este metodo sólo podemos asignar UNA funcion a cada evento
Este metodo funciona igual en TODOS los navegadores

Modelo Avanzado

Tambien podemos capturar eventos con el *modelo avanzado*

Con este metodo podemos asignar varias funciones a un mismo evento

Este modelo se aplica distinto según el navegador

Para enganchar/desenganchar una funcion a un evento con este modelo se utiliza:

- `addEventListener` y `removeEventListener` en Firefox, Opera y Safari (W3C way)
  Le pasamos 3 parametros:
  1. *Tipo de Evento* : `click`, `change`,...
  2. *Funcion a ejecutar* (*handler*, *callback*) : Recibe un objeto `e` con info sobre el evento
     En `e.target` tenemos el elemento que lanzó el evento
  3. *¿Utilizo Capturing?* : Poniendolo a `false` utilizariamos sólo *Bubbling*
- `attachEvent` y `detachEvent` en IE (Microsoft way)
  Le pasamos 2 parametros:
  1. *Tipo de Evento*: `onclick`, `onchange`,...
  2. *Funcion a ejecutar* (*handler*, *callback*): Para acceder a la info del evento hay que mirar el objeto global `window.event`
     En `event.srcElement` tenemos el elemento que lanzó el evento

18.2 Deteniendo el flujo de los eventos

Algunos elementos tienen un comportamiento por defecto (*por ejemplo al hacer click sobre un link nos lleva a su URL*).

Esta acción por defecto se ejecuta al final (si tenemos otras funciones asignadas al evento)

Para desactivar la acción por defecto utilizamos el metodo `e.preventDefault()`. En IE pondriamos a `false` la propiedad `returnValue` de `window.event`

Podemos detener la propagacion del evento con el metodo `e.stopPropagation()`

En IE pondriamos a `true` la propiedad `cancelBubble` de `window.event`

Cuando la función asignada al evento devuelve `false` se aplica automaticamente `e.preventDefault()` y `e.stopPropagation()`

18.3 Delegación de eventos

→ Gestión de eventos vs Delegación de eventos | anieto2k.com
→ JavaScript Event Delegation is Easier than You Think | sitepoint.com

Aprovechando el *bubbling* y la deteccion del *target* podemos optimizar (en algunos casos) nuestra gestión de eventos con la **delegación de eventos**

Para el caso en que tengamos que capturar los eventos de muchos elementos (*por ejemplo los clicks en celdas de una tabla*), podemos capturar el evento de su contenedor (*la tabla*) y detectar luego cual de sus hijos (*qué celda*) provocó el evento,

Las principales ventajas de este sistema son:

- Hay muchas menos definiciones de eventos: menos espacio en memoria y mayor performance
- No hay que re-capturar los eventos de los elementos añadidos dinamicamente

18.4 Eventos con jQuery

→ Events | api.jquery.com
→ Events and Event Delegation | jqfundamentals.com

Con *jQuery* podemos realizar nuestra gestion de eventos sin tener que preocuparnos de las diferencias entre navegadores:

`$().bind()` y `$().unbind()`

El `addEventListener`/`removeEventListener` cross-browsing

```
.bind( eventType, [ eventData ], handler(eventObject) )
.unbind( [ eventType ], [ handler(eventObject) ] )
```

El `handler` recibe un objeto `event` propio de jQuery

Los tipos de eventos que podemos capturar son `blur`, `focus`, `focusin`, `focusout`, `load`, `resize`, `scroll`, `unload`, `click`, `dblclick`, `mousedown`, `mouseup`, `mousemove`, `mouseover`, `mouseout`, `mouseenter`, `mouseleave`, `change`, `select`, `submit`, `keydown`, `keypress`, `keyup`, `error`

Tambien podemos crearnos nuestros propios tipos de eventos

`$().on()` y `$().off()`

El `$().bind()`/`$().unbind()` cross-functional

```
.on( events [, selector ] [, data ], handler )
.off( events [, selector ] [, handler ] )
```

$().trigger()

Nos permite ejecutar todos los handlers asociados a un evento

```
.trigger( eventType, extraParameters )
```

$().toggle()

Adjunta varias funciones a un elemento que seran ejecutadas en sucesivos clicks

```
.toggle( handler(eventObject), handler(eventObject), [ handler(eventObject) ]\
 )
```

event.preventDefault()

El `e.preventDefault` cross-browsing

event.stopPropagation()

El `e.stopPropagation` cross-browsing

event.stopImmediatePropagation()

Además de hacer `e.stopPropagation` cross-browsing, detiene el resto de handlers asociados al mismo evento

event.target

El `e.target` (elemento que provocó el evento) cross-browsing

event.type

El tipo de evento lanzado

19. JSON

→ Introducción a JSON | json.org
→ Tutorial JSON | etnassoft.com
→ JSON and JSONP | Angus Croll blog
→ Toma de contacto con JSON | anieto2k.com

```
{
"name": "Eric Clapton",
"occupation": "Guitar Hero",
"bands": ["Cream", "Blind Faith"]
}
```

JSON (Javascript Object Notation) es un formato de intercambio de datos basado en la notación literal de Javascript para la representación de objetos, arrays, cadenas, booleanos y números

Ventajas de este formato de datos frente a XML:

- Más ligero (su estructura necesita menos elementos que XML) por lo que es ideal para peticiones AJAX
- Más fácil de transformar a objeto Javascript (con eval se haria directo)

Particularidades del formato JSON frente a la notación literal de Javascript:

- Los pares nombre-valor van siempre con comillas dobles
- JSON puede representar 6 tipos de valores: objetos, arrays, números, cadenas, booleanos y null
- Las fechas no se reconocen como tipo de dato
- Los números no pueden ir precedidos de 0 (salvo los decimales)

Las cadenas JSON deben ser convertidas a objetos Javascript para poder utilizarlas (y viceversa). Para ello podemos utilizar:

- `eval()`: No se recomienda utilizarlo directamente
- `JSON.parse`: Convierte una cadena JSON en un objeto Javascript
  hace eval pero comprueba el formato antes de hacerlo
- `JSON.stringify`: Convierte un objeto Javascript en una cadena JSON
- `jQuery.parseJSON`: con jQuery también podemos hacer el parseo del JSON

El objeto JSON está disponible de forma nativa en los navegadores compatibles con ECMAScript 5

```
>>> JSON.parse('{"bar":"new property","baz":3}')
Object { bar="new property", baz=3}
```

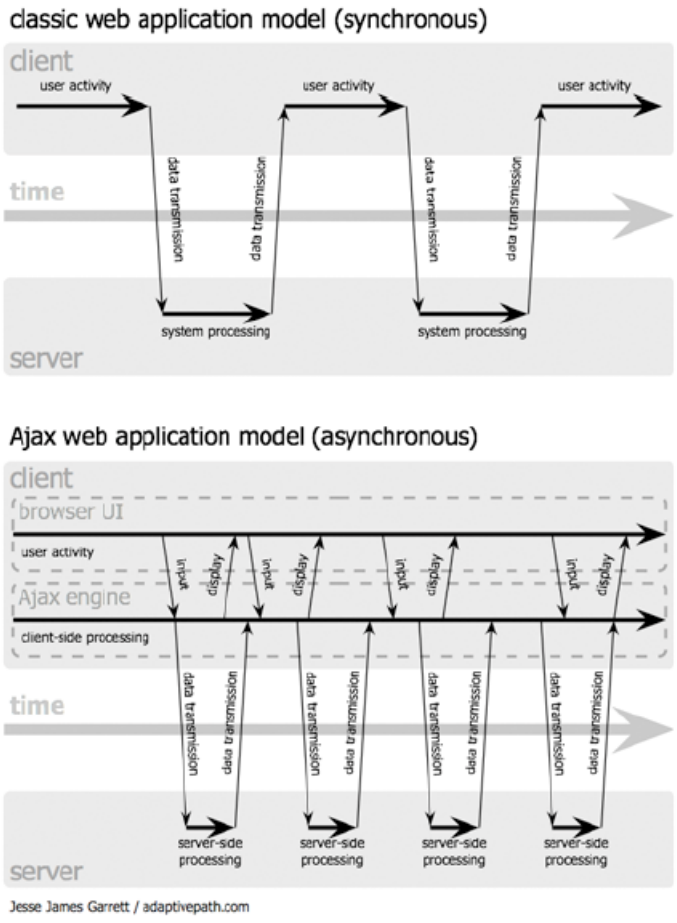
```
>>> JSON.stringify({ breed: 'Turtle', occupation: 'Ninja' });
"{"breed":"Turtle","occupation":"Ninja"}"
```

20. AJAX

> → Ajax: A New Approach to Web Applications | 18/02/2005 Jesse James Garrett (adaptive-path.com)

Agrupa un conjunto de tecnologias cuyo eje central son las peticiones asincronas al servidor a traves del objeto `XMLHttpRequest()` (XHR)

La aplicación de esta técnica dio lugar a las llamadas aplicaciones AJAX, donde no es necesario refrescar la pagina para obtener nuevo contenido.



AJAX

AJAX son las siglas de **Asynchronous JavaScript + XML**:

- *Asynchronous* porque despues de hacer una peticion HTTP no necesita esperar a una respuesta, sino que puede seguir haciendo otras cosas y ser notificado cuando llegue la respuesta
- *JavaScript* porque creamos los objetos XHR con Javascript
- *XML* porque inicialmente era el formato standard de intercambio de datos. Actualmente una petición HTTP suele devolver JSON (o HTML)

La mayor limitación de una petición AJAX es que no puede acceder a datos de un dominio diferente al que estamos (*cross-domain*)

Pero hay maneras de solucionar este problema: *JSONP* y *CORS*

20.1 Haciendo la petición

> → XMLHttpRequest | MDN
> → Using the XML HTTP Request object | http://jibbering.com/

```
1 var xhr = new XMLHttpRequest();
2 xhr.onreadystatechange = myCallback;
3 xhr.open('GET', url, true);
4 xhr.send('');
```

Las peticiones AJAX las hacemos a traves de objetos XHR.

Para crear un objeto XHR utilizamos el objeto `XMLHttpRequest()` que es nativo en IE7+, Safari, Opera y Firefox

```
1 var xhr = new XMLHttpRequest();
```

Una vez creado el objeto XHR, capturamos el evento `readystatechange` de este objeto y le enganchamos una función.

```
1 xhr.onreadystatechange = myCallback;
```

Despues hay que llamar al metodo `open()` pasandole los parametros de la peticion

```
1 xhr.open('GET', 'somefile.txt', true);
```

- *El 1er parametro es el tipo de peticion (GET, POST,...).*
- *El 2º parametro es la URL*
- *El 3er parametro indica si la petición es asíncrona (true) o síncrona (false)*

Despues hay que llamar al metodo `send()` para hacer la petición

```
xhr.send('');
```

20.2 Procesando la respuesta

```
function myCallback() {
    if (xhr.readyState < 4) {
        return; // not ready yet
    }
    if (xhr.status !== 200) {
        alert('Error!'); // the HTTP status code is not OK
        return;
    }
    // all is fine, do the work
    alert(xhr.responseText);
}
```

El objeto XHR tiene una propiedad llamada `readyState` y cada vez que cambia esta propiedad se dispara el evento `readystatechange`

Los posibles valores de readyState son:

- 0—uninitialized
- 1—loading
- 2—loaded
- 3—interactive
- **4—complete**

Cuando `readyState` llega a 4 significa que ya se ha recibido una respuesta

Una vez hemos recibido la respuesta hay que chequear el estado de la misma en la propiedad `status` del objeto XHR

El valor que nos interesa para esta propiedad es 200 (OK)

20.3 AJAX con jQuery

```
$.ajax({
    url: 'ajax/test.html',
    success: function(data) {
        $('.result').html(data);
        alert('Load was performed.');
    }
});
```

Con jQuery podemos realizar nuestra petición por AJAX con el metodo `$.ajax()`

El metodo `$.ajax()` devuelve un objeto `jqXHR` que viene a ser una version mejorada del objeto nativo `XMLHttpRequest`

Parametros `$.ajax()`

Al metodo `$.ajax()` le pasamos los parametros de nuestra petición AJAX que son, entre otros:

url

La URL donde hacemos la peticion

type

El tipo de petición, `POST` o `GET` (por defecto)

Las peticiones `GET` las utilizamos normalmente para recibir datos (ya que se pueden cachear)

Las peticiones `POST` las utilizamos para mandar datos al servidor

data

Los datos que enviaremos al servidor

dataType

El tipo de datos que esperamos recibir del servidor (json, html, xml, jsonp, ...)

Funciones callback `$.ajax()`

Al `$.ajax()` le podemos pasar tambien unas cuantas funciones callback que se ejecutaran dependiendo del resultado de la petición

success

La función que queremos ejecutar cuando recibamos la respuesta.

Si los datos recibidos estan en formato JSON, la función los recibe directamente transformados en objeto Javascript

A esta función le llega, ademas de los datos recibidos, el status de la petición y el objeto `jqXHR` que maneja la petición.

error

Esta función se ejecutará si falla la petición.

A esta función le llega el objeto `jqXHR` que maneja la petición y el error.

complete

Esta función se ejecutará cuando finalize la petición

A esta función le llega el objeto `jqXHR` que maneja la petición y el error o éxito de la operación.

beforeSend

Esta función se ejecuta antes de hacer la petición

A esta función le llega el objeto jqXHR que maneja la petición y los parametros de la petición

dataFilter

Esta función se ejecuta inmediatamente despues de la recepción exitosa de los datos

A esta función le llega la información recibida y el valor del dataType, y lo que devuelve le llega a success

21. Expresiones Regulares

> regexpal.com | a JavaScript regular expression tester
> Regular-Expressions.info/
> Javascript: Expresiones Regulares | mundogeek.net
> Expresiones Regulares | javascript.espaciolatino.com
> Programmer's Guide to Regular Expressions | javascriptkit.com

Las expresiones regulares nos permiten buscar y manipular texto de una forma muy potente. Un expresión regular consiste en: - Un **patron** (*pattern*) que se usa para localizar textos que se ajusten a él - **Modificadores** (opcionales) que nos indican como aplicar el patron

En Javascript tenemos disponibles **objetos de expresiones regulares** que podemos crear: - Con la función constructora `RegExp` : `new RegExp("j.*t")` - Con la notacion literal: `/j.*t/`;

21.1 Propiedades de los Objetos RegExp

Los objetos de expresiones regulares tienen las siguientes propiedades:

- `global`: Con `false` (por defecto) devuelve solo el primer elemento encontrado. Con `true` devuelve todos los elementos encontrados
- `ignoreCase`: Si está a `true` haremos el matching sensible a mayusculas (por defecto a `false`)
- `multiline`: Si está a `true` realizará la busqueda entre varias lineas (por defecto a `false`)
- `lastIndex`: La posición por la que empezar la busqueda (por defecto a `0`)
- `source`: Contiene la expresion regular

Estas propiedades (excepto `lastIndex`) no pueden ser modificadas despues de creado el objeto

Las 3 primeras propiedades representan a los *modificadores* de la expresion regular: - **g**: global - **i**: ignoreCase - **m**: multiline

```
>>> var re = new RegExp('j.*t', 'gmi');
undefined
>>> re.global
true
>>> re.global = false;
false
>>> re.global
true
>>> var re = /j.*t/ig;
undefined
>>> re.global
true
>>> re.source
"j.*t"
```

21.2 Métodos de los Objetos RegExp

Los objetos de expresiones regulares tienen los siguientes metodos:

- `test()`:Devuelve `true` si encuentra algo y `false` en caso contrario
- `exec()`: Devuelve un array de cadenas que cumplan el patron

```
>>> /j.*t/.test("Javascript")
false
>>> /j.*t/i.test("Javascript")
true
>>> /s(amp)le/i.exec("Sample text")
["Sample", "amp"]
>>> /a(b+)a/g.exec("_abbba_aba_")
["abbba", "bbb"]
```

21.3 Métodos de String que aceptan Expresiones Regulares

Tenemos disponibles los siguientes métodos del objeto `String` para buscar dentro de un texto mediante expresiones regulares:

- `match()`: Devuelve un array de ocurrencias
- `search()`: Devuelve la posición de la primera ocurrencia
- `replace()`: Nos permite sustituir la cadena encontrada por otra cadena
- `split()`: Acepta una expresión regular para dividir una cadena de texto en elementos de un array

replace()

- Si omitimos el modificador **g** solo reemplazamos la primera ocurrencia
- Podemos incluir en la sustitución la cadena encontrada con **$&**
- Cuando la expresión regular contiene grupos podemos acceder a la ocurrencia de cada grupo con **$1**, **$2**, etc...
- Al especificar la sustitución podemos pasar una función donde:
  1. El *primer parametro* es la cadena encontrada
  2. El *último parametro* es la cadena donde se está buscando
  3. El *antepenultimo parametro* es la posición de la ocurrencia
  4. El *resto de parametros* son las ocurrencias de cada grupo del patron

```
>>> var s = "HelloJavaScriptWorld"
undefined
>>> s.match(/a/);
["a"]
>>> s.match(/a/g);
["a", "a"]
>>> s.match(/j.*a/i); ["Java"]
>>> s.search(/j.*a/i); 5
>>> s.replace(/[A-Z]/g, '');
"elloavacriptorld"
>>> s.replace(/[A-Z]/, ''); "elloJavaScriptWorld"
>>> s.replace(/[A-Z]/g, "_$&"); "_Hello_Java_Script_World"
>>> s.replace(/([A-Z])/g, "_$1"); "_Hello_Java_Script_World"
>>> "juanmanuel.garrido@softonic.com".replace(/(.*)@.*/, "$1");
"juanmanuel.garrido"
>>> function replaceCallback(match){return "_" + match.toLowerCase();}
undefined
>>> s.replace(/[A-Z]/g, replaceCallback); "_hello_java_script_world"
>>> var sMail = "juanmanuel.garrido@softonic.com";
undefined
>>> var rRegExp = /(.*)@(.*)\.(.*)/;
undefined
>>> var fCallback = function () { args = arguments; return args[1] + " de " +\
 args[2].toUpperCase(); }
undefined
>>> sMail.replace( rRegExp, fCallback);
"juanmanuel.garrido de SOFTONIC"
>>> args
["juanmanuel.garrido@softonic.com", "juanmanuel.garrido",
"softonic", "com", 0, "juanmanuel.garrido@softonic.com"]
>>> var csv = 'one, two,three ,four';
>>> csv.split(',');
["one", " two", "three ", "four"]
```

```
>>> csv.split(/\s*,\s*/)
["one", "two", "three", "four"]
>>> "test".replace('t', 'r')
"rest"
>>> "test".replace(new RegExp('t'), 'r')
"rest"
>>> "test".replace(/t/, 'r')
"rest"
```

21.4 Sintaxis de las Expresiones Regulares

> http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular
> http://www.addedbytes.com/cheat-sheets/regular-expressions-cheat-sheet/
> http://www.visibone.com/regular-expressions/

[abc]

Busca coincidencias en los caracteres del patron

```
>>> "some text".match(/[otx]/g)
["o", "t", "x", "t"]
```

[a-z]

Busca coincidencias en el rango de caractares
`[a-d]` es lo mismo que `[abcd]`
`[a-z]` busca todas los caracteres en minuscula
`[a-zA-Z0-9_]` busca todo los caracteres, numeros y el guión bajo

```
>>> "Some Text".match(/[a-z]/g)
["o", "m", "e", "e", "x", "t"]
>>> "Some Text".match(/[a-zA-Z]/g)
["S", "o", "m", "e", "T", "e", "x", "t"]
```

[^abc]

Devuelve lo que NO coincida con el patron

```
>>> "Some Text".match(/[^a-z]/g)
["S", " ", "T"]
```

a|b

Devuelve *a* o *b* (la barra indica OR)

```
>>> "Some Text".match(/t|T/g);
["T", "t"]
>>> "Some Text".match(/t|T|Some/g);
["Some", "T", "t"]
```

a(?=b)

Devuelve *a* solamente si está seguida de *b*

```
>>> "Some Text".match(/Some(?=Tex)/g);
null
>>> "Some Text".match(/Some(?= Tex)/g);
["Some"]
```

a(?!b)

Devuelve *a* solamente si NO está seguida de *b*

```
>>> "Some Text".match(/Some(?! Tex)/g);
null
>>> "Some Text".match(/Some(?!Tex)/g);
["Some"]
```

\

Carácter de escape utilizado para localizar caracteres especiales utilizados en el patron como literales

```
>>> "R2-D2".match(/[2-3]/g)
["2", "2"]
>>> "R2-D2".match(/[2\-3]/g)
["2", "-", "2"]
```

\n

Nueva linea

\r

Retorno de carro (Para comenzar una nueva linea se usa `\r\n` en Windows, `\n` en Unix y `\r` en Mac)

\f

Salto de pagina

\t

Tabulación

\v

Tabulación Vertical

\s

Espacio en blanco o cualquiera de las 5 secuencias de escape de arriba

```
>>> "R2\n D2".match(/\s/g)
["\n", " "]
```

\S

Lo contrario de lo de arriba. Devuelve todo excepto espacios en blanco y las 5 secuencias de escape de antes. Lo mismo que `[^\s]`

```
>>> "R2\n D2".match(/\S/g)
["R", "2", "D", "2"]
```

\w

Cualquier letra, numero o guión bajo. Lo mismo que `[A-Za-z0-9_]`

```
>>> "Some text!".match(/\w/g)
["S", "o", "m", "e", "t", "e", "x", "t"]
```

\W

Lo contrario de `\w`

```
>>> "Some text!".match(/\W/g)
[" ", "!"]
```

\d

Localiza un numero. Lo mismo que `[0-9]`

```
>>> "R2-D2 and C-3PO".match(/\d/g)
["2", "2", "3"]
```

\D

Lo contrario de `\d`. Localiza caracteres no-numericos. Lo mismo que `[^0-9]` o `[^\d]`

```
>>> "R2-D2 and C-3PO".match(/\D/g)
["R", "-", "D", " ", "a", "n", "d", " ", "C", "-", "P", "O"]
```

\b

Coincide con un limite de palabra (espacio, puntuación, guión...)

```
>>> "R2D2 and C-3PO".match(/[RD]2/g)
["R2", "D2"]
>>> "R2D2 and C-3PO".match(/[RD]2\b/g)
["D2"]
>>> "R2-D2 and C-3PO".match(/[RD]2\b/g)
["R2", "D2"]
```

\B

Lo contrario de `\b`

```
>>> "R2-D2 and C-3PO".match(/[RD]2\B/g)
null
>>> "R2D2 and C-3PO".match(/[RD]2\B/g)
["R2"]
```

^

Representa el principio de la cadena donde se está buscando. Si tenemos el modificador `m` representa el principio de cada linea.

```
>>> "regular\nregular\nexpression".match(/r/g);
["r", "r", "r", "r", "r"]
>>> "regular\nregular\nexpression".match(/^r/g);
["r"]
>>> "regular\nregular\nexpression".match(/^r/mg);
["r", "r"]
```

$

Representa el final de la cadena donde se está buscando. Si tenemos el modificador `m` representa el final de cada linea.

```
>>> "regular\nregular\nexpression".match(/r$/g);
null
>>> "regular\nregular\nexpression".match(/r$/mg);
["r", "r"]
```

.

Representa a cualquier carácter excepto la nueva linea y el retorno de carro

```
>>> "regular".match(/r./g);
["re"]
>>> "regular".match(/r.../g);
["regu"]
```

*

Hace matching si el patron precedente ocurre 0 o más veces.
`/.*/` devolverá todo incluido nada (cadena vacia)

```
>>> "".match(/.*/)
[""]
>>> "anything".match(/.*/)
["anything"]
>>> "anything".match(/n.*h/)
["nyth"]
```

?

Hace matching si el patron precedente ocurre 0 o 1 vez.

```
>>> "anything".match(/ny?/g)
["ny", "n"]
```

+

Hace matching si el patron precedente ocurre 1 o más veces (al menos una vez).

```
>>> "anything".match(/ny+/g)
["ny"]
>>> "R2-D2 and C-3PO".match(/[a-z]/gi)
["R", "D", "a", "n", "d", "C", "P", "O"]
>>> "R2-D2 and C-3PO".match(/[a-z]+/gi)
["R", "D", "and", "C", "PO"]
```

{n}

Hace matching si el patron precedente ocurre exactamente n veces.

```
>>> "regular expression".match(/s/g)
["s", "s"]
>>> "regular expression".match(/s{2}/g)
["ss"]
>>> "regular expression".match(/\b\w{3}/g)
["reg", "exp"]
```

{min,max}

Hace matching si el patron precedente ocurre entre min y max veces.
Se puede omitir max (solo tendrá minimo) No se puede omitir min

```
>>> "doooooooooodle".match(/o/g)
["o", "o", "o", "o", "o", "o", "o", "o", "o", "o"]
>>> "doooooooooodle".match(/o{2}/g)
["oo", "oo", "oo", "oo", "oo"]
>>> "doooooooooodle".match(/o{2,}/g)
["oooooooooo"]
>>> "doooooooooodle".match(/o{2,6}/g)
["oooooo", "oooo"]
```

(pattern)

Cuando el patrón está en parentesis, se captura y se guarda para poder utilizarlo en sustituciones (captura de patrones).
Estas capturas estan disponibles en `$1`, `$2`,... `$9`

```
>>> "regular expression".replace(/(r)/g, '$1$1')
"rregularr exprression"
>>> "regular expression".replace(/(r)(e)/g, '$2$1')
"ergular experssion"
```

{?:pattern}

Patrón no capturable (no disponible en `$1`, `$2`, ...)

```
>>> "regular expression".replace(/(?:r)(e)/g, '$1$1')
"eegular expeession"
```