

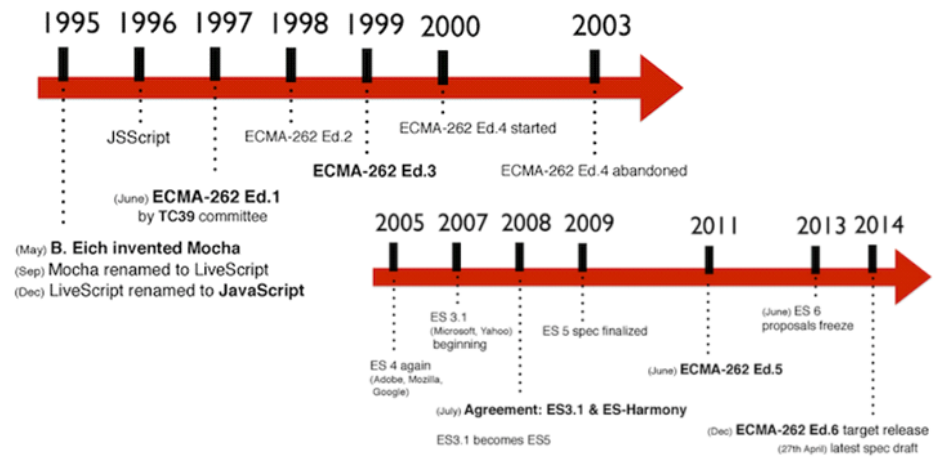
ECMAScript 6 (ES6 / ES2015)

sábado, 9 de septiembre de 2017 13:33

Brendan Eich
Netscape



European Computer Manufacturers' Association

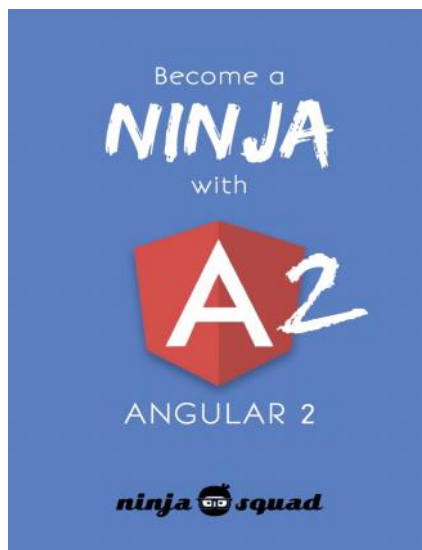
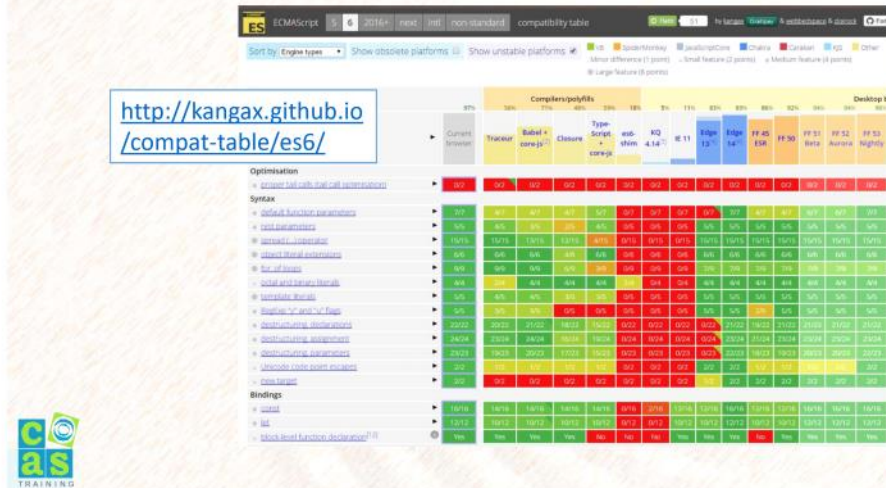


Diciembre de 2014

- Constates (const). Variables con ámbito (let)
- Función Arrow. This "semántico"
- *Template Strings*: interpolación de variables
- Valores por defecto
- Clases (class)
- Módulos (export / import)
- Promesas (promise)
- *Destructuring* ...

JS

<http://es6-features.org/>



En uno de sus primeros capítulos hace un resumen del nuevo estándar desde la perspectiva de Angular

Nuevos elementos de código

domingo, 30 de julio de 2017 22:19

- Constates (const). Variables con ámbito (let)
- Función Arrow
- *Template Strings*: interpolación de variables

El uso de var sigue siendo idéntico a versiones anteriores, usándose en este caso para declarar un array

const y let

Salida por consola utilizando "template strings" en los que se conservan los saltos de línea

```
// Ejemplo de código en ES6
var data = [{precio: 12}, {precio: 34}, {precio: 19}];

data.forEach( elem => {
  if (true) {

const iva = 1.16
let precioFinal = elem.precio * iva

    console.log(`
Oferta:
El precio final es ${precioFinal}`);
  }

// console.log (iva)
});
```

la función callback del método forEach, propio de ES5 se define con el nuevo formato "Arrow function" con elem como único argumento

línea que daría error por hacer referencia a una variable en un ámbito en el que no existe

Otros elementos:

- Nuevos objetos iterables Map y Set
- Bucle for ... of
- Valores por defecto en funciones y métodos

Nuevos objetos iterables de tipo Map, inicializados de 2 de las formas posibles

```
let map = new Map()
.set("A",1)
.set("B",2)
.set("C",3);

let map2 = new Map([
  [ "A", 1 ],
  [ "B", 2 ],
  [ "C", 3 ]
]);
```

<https://hackernoon.com/what-you-should-know-about-es6-maps-dc66af6b9a1e>

bucles que iteran a través de los elementos de objetos iterables (incluyendo Array, Map, Set, el objeto arguments, etc.),

```
let aDatos = [10,20,30]
let nTotal1 = "";
for (let dato in aDatos) {
  nTotal1 += dato
  console.log(dato);
}
// "0"
// "1"
// "2"
console.log(`Total : ${nTotal1}`);
```

El bucle for...in, adecuado para Objetos, producía resultados incongruentes en el caso de los Arrays

```
let nTotal2 = 0;
for (let dato of aDatos) {
  nTotal2 += dato
  console.log(dato);
}
```

El bucle for...of se comporta igualmente en el caso de Objetos, añadiendo un comportamiento más coherente en caso de Arrays

```
// 10  
// 20  
// 30  
console.log(`Total : ${nTotal2}`);
```

Variables y constantes

sábado, 30 de diciembre de 2017 12:47

la declaración con `var` siempre se "eleva" al inicio del código, independientemente de que acompañe o no a una inicialización

```
(function prueba_var () {  
  console.log(x)  
  var x = 20  
})();
```

Equivale a

```
var x  
(function prueba_var () {  
  console.log(x)  
  x = 20  
})();
```

devuelve undefined

la declaración con `let` no se eleva

```
(function prueba_let () {  
  console.log(x)  
  let x = 20  
})();
```

ReferenceError x is not defined

Además su ámbito de existencia está limitado al bloque en que se declara y los bloques contenidos en el

```
(function bloques () {  
  let x = 0  
  let y = 0  
  {  
    x = 20  
    let y = "Modificada"  
    console.log(x)  
    console.log(y)  
    let z = 25  
  }  
  console.log(x)  
  console.log(y)  
  console.log(z)  
})();
```

20
Modificada
20
0
ReferenceError z is not defined

`const` declara como constantes los tipos elementales o las referencias a los objetos, pero nunca el contenido de los objetos. Para ello disponemos del método de ES5 `Object.freeze()`

```
(function constantes () {  
  const MES = "Enero"
```

```
const DIAS = 31
const USER = {
  name : "",
  apellido : "",
  puesto : ""
}
```

En un objeto "constante" podemos modificar, añadir o eliminar propiedades.

```
USER.name = "Pepe"
USER.apellido = "Perez"
USER.edad = 25
delete USER.puesto
```

```
console.log(USER) → { name: 'Pepe', apellido: 'Perez', edad: 25 }
```

```
USER = {}
})();
```


No podemos reasignar el objeto

TypeError: Assignment to constant variable.

Nuevas sintaxis

sábado, 30 de diciembre de 2017 19:39

- Clases
- *arrow functions*
- *template strings*
- *Map*
- Bucles *for...of*


Nuevos objetos iterables de tipo Map 

mapa clave/valor.
Cualquier valor (tanto objetos como valores primitivos) pueden ser usados como clave o valor.

```
let map = new Map()  
.set("A",1)  
.set("B",2)  
.set("C",3);
```

Alternativa para la creación de dichos objetos

```
let map2 = new Map([  
  [ "A", 1 ],  
  [ "B", 2 ],  
  [ "C", 3 ]  
]);
```

 cualquier otro objeto iterable cuyos elementos son pares clave-valor (arrays de 2 elementos). Cada par clave-valor será agregado al nuevo Map.

Bucles *for...of*

```
let aDatos = [10,20,30]  
  
for (let dato of aDatos) {  
  nTotal2 += dato  
  console.log(dato);  
}  
// 10  
// 20  
// 30  
console.log(`Total : ${nTotal2}`);
```

Operador de propagación

domingo, 28 de enero de 2018 23:51

El operador de propagación *spread operator* permite que una expresión sea expandida en situaciones donde se esperan múltiples argumentos (llamadas a funciones) o múltiples elementos (*arrays* literales).

Llamadas a funciones:

```
f(...iterableObj);
```

Arrays literales:

```
[...iterableObj, 4, 5, 6]
```

Desestructuración *destructuring*:

```
[a, b, ...iterableObj] = [1, 2, 3, 4, 5];
```

Se define una función que espera múltiples argumentos

```
function f(x, y, z) { }
```

La función es invocada pasándole un array con el operador de propagación, que será expandido a los múltiples argumentos que espera la función

```
var args = [0, 1, 2];  
f(...args);
```

<http://www.etnassoft.com/2014/06/03/el-operador-de-propagacion-en-javascript-ecmascript-6-y-polyfill/>

Desestructuración

lunes, 29 de enero de 2018 0:02

La desestructuración no es un concepto nuevo en programación. De hecho, eso es algo que se ha tenido muy en cuenta a la hora de fijar el estándar: incorporar de forma progresiva al lenguaje Javascript lo mejor de otros.

- en Python o en OCaml, tendríamos las tuplas
- en PHP las listas
- sus correspondientes en Perl y Clojure...


una expresión que permite asignar valores a nombres conforme a una estructura de tabla dada

Desestructuración de ARRAYS

```
const aNumbers = [1, 2, 3, 4]

const [uno, dos, tres] = aNumbers
const [uno, dos, tres] = [1, 2, 3]
console.log(uno, dos, tres)
```

Desestructuración de un array declarando las variables




Desestructuración de un objeto

```
{
  const oNumbers = {
    uno: 1,
    dos: 2,
    tres: 3,
    cuatro: 4
  }

  const {uno, cuatro, tres, dos} = oNumbers

  console.log(uno, dos, tres, cuatro)
}
```

Cada una de las variables recibe el valor de uno de los miembros del objeto



Ejemplos en el fichero basicos.4.destructuring.js

<http://www.etnassoft.com/2016/07/04/desestructuracion-en-javascript-parte-1/>

Valores por defecto

lunes, 29 de enero de 2018 0:11

ES6: parámetros por defecto y
desestructuración del paso de parámetros

```
function drawCircleES6( {radius = 30,  
                        coords = { x: 0, y: 0}  
                        } = {}) {  
    console.log(radius, coords);  
};
```

Desestructuración:

```
{radius = 30,  
 coords = { x: 0, y: 0}} = {}
```

valores por defecto
desestructurados:
- radius
- coords

parámetro real
recibido como objeto

Diversas llamadas a la función

```
drawCircleES6(); // radius: 30, coords.x: 0, coords.y: 0 }  
drawCircleES6({radius: 10}); // radius: 10, coords.x: 0, coords.y: 0 }  
drawCircleES6({coords: {y: 10, x: 30}, radius: 10}); // radius: 10, coords.x: 30, coords.y: 10 }
```

Ejemplos en el fichero basicos.4.default.js

Función Arrow. This "semántico"

sábado, 14 de octubre de 2017 11:38

```
let oPrueba = {
  precio: 12,
  iva : 1.16,
};
oPrueba.calculaIvaAsiync = function () {
  setTimeout (function () {
    let precioFinal = this.precio * this.iva
    console.log(`
      Usando una funcion clásica:
      El precio final es ${precioFinal}
    `);
  }, 1000)
}
oPrueba.calculaIvaAsiync()
```

```
// la función callback del método
setTimeout
// interpreta this como una
// llamada al sistema,
// no como el objeto en el que se
// ha definido
```

```
// Versión alternativa
// unando una arrow function
```

```
oPrueba.calculaIvaAsiync_Arrow = function () {
  setTimeout (() => {
    let precioFinal = this.precio * this.iva;
    console.log(`
      Usando una arrow function:
      El precio final es ${precioFinal}
    `);
  }, 1000)
}
oPrueba.calculaIvaAsiync_Arrow();
```

```
// la función callback del método
setTimeout
// interpreta this semanticamente,
// según donde se ha definido la
// función que lo usa
// y no según donde se utiliza, que
// supondría hacerlo como una llamada
// al sistema.
```

Clases

sábado, 29 de julio de 2017 15:30

// Ejemplo de código en ES6

Clase "padre" → `class Libro {}`

Clase que hereda de la anterior → `class LibroTecnico extends Libro {`

Constructor → `constructor(tematica, paginas) {`
`super(tematica, paginas);`
`this.capitulos = [];`
`this.precio = "";`
`// ...`
`}`

Método que define valores por defecto → `metodo(pValor = "foo") {`
`// ...`
`}`

NO EXISTEN

- Propiedades definidas fuera de los métodos
- Modificadores de acceso (*private*, *protected*, *public*)
- Interfaces

Estos elementos se añaden en la implementación de las clases propia de *TypeScript*

También existe el modificador **Static**

Azúcar sintáctico.
En JS NO EXISTEN CLASES
Sólo hay PROTOTYPES

La nueva forma de escribir en ES6
hace más sencillo el uso de los
prototipos al asimilarlos a la forma
habitual de trabajar con clases

Ejemplo de Clases

sábado, 14 de octubre de 2017 17:33

Declaración de una clase

```
class Libro {  
  constructor(tematica, paginas) {  
    this.tematica = tematica  
    this.paginas = paginas  
  }  
}
```

constructor

Declaración de una clase que hereda de la anterior

```
class LibroTecnico extends Libro {  
  constructor(tematica, paginas, precio) {  
    super(tematica, paginas);  
    this.capitulos = [];  
    this.precio = precio;  
    // ...  
  }  
  
  precioFinal(pIva = 16) {  
  
    return this.precio * (1 + pIva/100)  
  }  
}
```

constructor que invoca el constructor de la clase padre

ejemplo de método con parámetros con valor por defecto

Instanciación de objetos

```
let libro1 = new LibroTecnico("Informatica", 250, 30)  
console.dir(libro1)  
console.dir(`Precio final: ${libro1.precioFinal()} €`)  
console.dir(`En Canarias : ${libro1.precioFinal(0)} €`)
```

```
LibroTecnico {  
  tematica: 'Informatica',  
  paginas: 250,  
  capitulos: [],  
  precio: 30 }  
'Precio final: 34.8'  
'En Canarias : 30'
```

Módulos

sábado, 29 de julio de 2017 15:30

Sistemas previos

- *CommonJS* (e.g. Node)
- *Asynchronous Module Definition* - AMD (e.g. require.js, Dojo)

- *require*
- *exports*

- *define*

<https://auth0.com/blog/javascript-module-systems-showdown/>

Módulos en ES6

- *import*
- *export*

Síncrono y asíncrono

Creación de un módulo en el que se exporta una función, escrita en el nuevo formato "arrow function"

definición de un módulo:
corresponde al fichero
que lo incluye

función
exportada

```
//File: modulo.js  
export const hello = (nombre) => {  
  return "Hola " + nombre;  
}
```

Uso del módulo anteriormente creado

importación de
una función

objeto en el que se usa la
función importada

```
//File: app.js  
import { hello } from "./modulo.js";  
  
var app = {  
  saludo : () => {console.log(hello("Carlos"))};  
}  
  
app.saludo()
```

Problema:

El estándar ES6 describe como se declaran los módulos,
pero no especifica cómo deben ser cargados

Hasta muy recientemente NO DISPONIBLE EN LOS NAVEGADORES

Tampoco es soportado en *NodeJS*, que continua usando su propia definición de módulos

Actualmente, se puede indicar al navegador que procese correctamente los archivos JavaScript que usan módulos utilizando el atributo `type="module"`

```
<script src="app.js" type="module"></script>
```

Una promesa representa el resultado eventual de una operación.
Se utiliza para especificar que se hará cuando esa eventual operación de un resultado de éxito o fracaso.

Promesas

JS

Un objeto promesa representa un valor que todavía no está disponible pero que lo estará en algún momento en el futuro

Permiten escribir código asíncrono de forma más similar a como se escribe el código síncrono:

La función asíncrona retorna inmediatamente y ese retorno se trata como un proxy cuyo valor se obtendrá en el futuro

El API de las promesas en Angular corresponde al **servicio \$q**

la biblioteca Q desarrollada por **Kris Kowal**

<https://github.com/kris/kowal/q>



Promesas: \$q

JS

```
function getPromise()
```

```
    var deferred=$q.defer();
```

crea una promesa

```
        deferred.resolve()  
        deferred.reject()
```

resuelve la promesa en un sentido u otro al cabo del tiempo

```
    return deferred.promise
```

devuelve la promesa

```
var promise = getPromise();
```

```
    promise.then(successCallback,failureCallback,notifyCallback);
```

```
    promise.catch(errorCallback)
```

```
    promise.finally(callback)
```

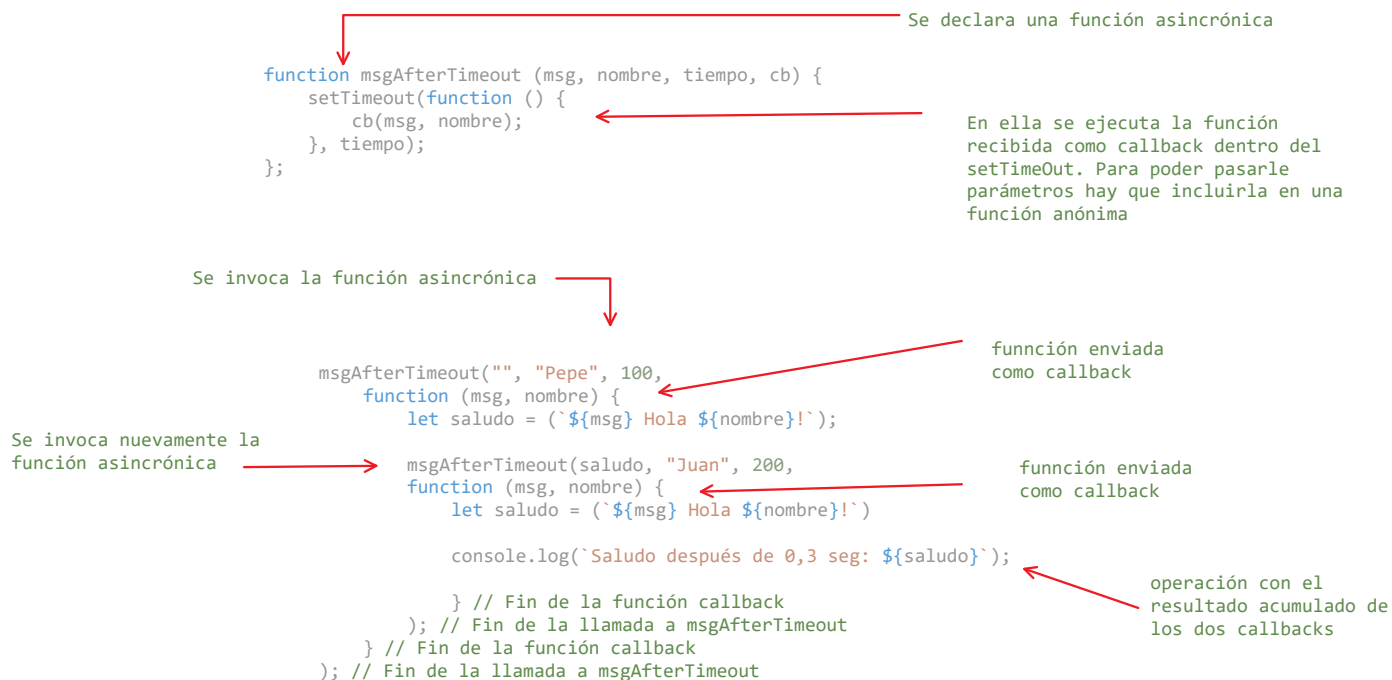


```
promise.catch(errorCallback)  
promise.finally(callback)
```



Callbacks anidados

sábado, 14 de octubre de 2017 13:52



Implementación new Promise

el objeto promesa recibe como parámetros dos funciones:

- La función *"resolve"*: se ejecutará cuando queramos finalizar la promesa con éxito.
- La función *"reject"*: se ejecutará cuando queramos finalizar una promesa informando de un caso de fracaso.

```
function hacerAlgoPromesa (){  
  return new Promise (function (resolve, reject) {  
    console.log ( 'hacer algo que ocupa un tiempo...');  
    setTimeout (resolve(), 1000);  
  })  
}
```

En este caso la promesa siempre se resuelve correctamente; la función admite como parámetro los datos que la promesa deba retornar

Utilización

a la función que retorna el objeto promesa se le encadenan el método then con dos funciones como parámetros,

- la función que se ejecutará cuando la promesa haya finalizado con éxito.
- la función que se ejecutara cuando la promesa haya finalizado informando de un caso de fracaso.

```
hacerAlgoPromesa()  
  .then (  
    function (){ console.log ( ' la promesa terminó.');},  
    function (){ console.log ( ' la promesa fracasó.');}  
  )
```

Alternativamente puede utilizarse el método catch para declarar la función que se ejecutara cuando la promesa haya finalizado informando de un caso de fracaso.

```
hacerAlgoPromesa()  
  .then (function (){ console.log ( ' la promesa terminó.');})  
  .catch(function (){ console.log ( ' la promesa fracasó.');})
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise

Ejemplo de promesas en ES6

sábado, 14 de octubre de 2017 14:43

```
function msgAfterTimeout (msg, nombre, tiempo) {  
  return new Promise((resolve, reject) => {  
    setTimeout(  
      () => resolve(`${msg} Hola ${nombre}!`),  
      tiempo)  
    })  
  })  
}
```

Función que crea y devuelve un **objeto promesa**

En este caso, la promesa siempre se resuelve correctamente, creando un mensaje de saludo a un usuario

Utilización de las promesas, encadenando las llamadas a ellas

msg almacena el resultado del primer proceso asincrónico

msg almacena los sucesivos resultados de los procesos asincrónicos

```
msgAfterTimeout("", "Pepe", 100)  
  .then((msg) =>  
    msgAfterTimeout(msg, "Juan", 200))  
  .then((msg) => {  
    console.log(`Saludo después de 0,3 seg: ${msg}`)  
  })
```

operamos finalmente con *msg*

MENSAJES

Saludo despues de 0,3 seg: Hola Pepe! Hola Juan!

PS D:\Desarrollo\Front_End_alce65\Angular\angular_4_2017\02_tecnologias\ES6>

Generadores

lunes, 29 de enero de 2018

23:47

Procesar cada uno de los elementos en una colección es un tipo de operación muy común. JavaScript proporciona diversas formas de iterar a través de los elementos de una colección, desde simples bucles for hasta map(), y filter(). Los iteradores y los generadores acercan el concepto de iteración directamente al núcleo del lenguaje y proporcionan un mecanismo para personalizar el comportamiento de los bucles for...of.

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_Generators

- Iteradores
- Iterables
- Generadores



Son **funciones especiales** (se declaran con function *) de las que se puede salir y entrar varias veces con resultados diferentes

- devuelven valores gracias a la palabra reservada yield
- continúan cuando se ejecuta el método next

<https://carlosazaustre.es/funciones-generadoras-en-ecmascript6/>

<https://speakerdeck.com/serabe/generadores-en-javascript>

Operaciones con arrays en JS 1.5



- `map()`,
- `filter()`,
- `some()`,
- `every()`,
- `forEach()`,
- `reduce()`,
- `reduceRight()`,

En todos los nuevos métodos de utiliza una **función callback**, es decir una función que es pasada como parámetro para que el método la utilice de la forma en que tiene previamente definida.

Los parámetros de dicha función son (element, index, array)



Arrays en JS 1.5 (1)



`map()` una proyección (como `select` en C#): el argumento es una función que transforma cada uno de los elementos.

```
array.map(function(i){return i.toUpperCase()});
```

convertiría cada elemento del array en mayúsculas

```
var numbers = [1, 4, 9];  
var roots = numbers.map(Math.sqrt);  
document.write("roots is : " + roots );
```

mostraría las raíces cuadradas de los elementos del array

Un ejemplo más complejo, podría transformar en objetos cada uno de los elementos del array

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (2)



`filter()` tiene como argumento una función lambda que evalúa cada elemento del array y devuelve un booleano. Se devuelve un **nuevo array** sólo con los elementos que hayan dado verdadero en la función callback

```
array.filter(function(i){return i.[0]=="a"});
```

```
function isBigEnough(element, index, array) {  
    return (element >= 10);  
}  
var aDatos = [12, 5, 8, 130, 44]  
var aFiltrado = aDatos.filter(isBigEnough);  
console.log(aFiltrado);
```

Ejemplo que devuelve [12,130,44]

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (3)



`some()` y `every()` utilizan funciones del mismo tipo para evaluar si el array en su conjunto las cumple, y devolver en consecuencia verdadero a falso.

- `some()` devuelve verdadero si algún elemento del array lo devuelve
- `every()` devuelve verdadero si todos los elementos del array lo devuelven

```
function isBigEnough(element, index, array) {  
    return (element >= 10);  
}  
var aDatos = [12, 5, 8, 130, 44]  
var isVal_some = aDatos.some(isBigEnough);  
var isVal_every = aDatos.every(isBigEnough);  
console.log(isVal_some);  
console.log(isVal_every);
```

true
false

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (4)



`forEach()` permite indicar cualquier función , booleana o no, que modifique o no los elementos, pero que en cualquier caso se aplica sobre cada uno de ellos.

```
function printBr(element, index, array) {  
    document.write("<br />[" + index + "] is " + element );  
}  
aDatos = [12, 5, 8, 130, 44];  
aDatos.forEach(printBr);
```

[0] is 12
[1] is 5
[2] is 8
[3] is 130
[4] is 44

```
function cuad(element, index, array) {  
    array[index] = element * element;  
}  
aDatos = [12, 5, 8, 130, 44];  
aDatos.forEach(cuad);  
console.log(aDatos);
```

[144, 25, 64, 16900, 1936]

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (5)



`reduce()`, aplica una función simultáneamente a pares de valores del array. desde la izquierda a la derecha, sucesivas veces, hasta reducir el array a un único valor

`reduceRight()`, realiza el mismo proceso desde la derecha a la izquierda, de nuevo sucesivas veces, hasta reducir el array a un único valor

```
var aDatos = [1, 2, 3, 4];  
var nTotal = aDatos.reduce(function(a, b){ return a * b; });  
console.log(aDatos);  
console.log(nTotal);
```

```
[1, 2, 3, 4]  
24
```

ECMAScript 5.1 (ECMA-262)

Inmutabilidad

viernes, 4 de enero de 2019 9:41

Datos primitivos

viernes, 4 de enero de 2019 9:45

A los datos **primitivos** se accede **por valor**

string
number
boolean

Las variables correspondientes a los tipos primitivos son accedidas por valor, es decir que se manipula directamente el valor real almacenado en la variable, lo único que importa es el valor en sí

Técnicamente, cada variable es un espacio de memoria diferente, donde se almacena un cierto valor, con independencia de cualquier otro

Inmutabilidad

Los datos primitivos son **inmutables**. Para cambiar el valor de una variable, se destruye el espacio de memoria antiguo y se selecciona uno nuevo, en el que se almacena el valor actual

```
let var_1 = 3
```

```
var_1 = 8
```

var_1

3



var_1

var_1

3
8

Asignación a otras variables

Cuando se **asigna** una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda. (Sería el procedimiento supondríamos por defecto, intuitivamente)

```
var var_1 = 3;
```

```
var var_2 = var_1;  
var var_3 = var_2 + 5;
```

// Ahora var_3 = 8 y var_1 sigue valiendo 3

var_1

3



var_1

var_2

var_3

3
3
8

Datos referenciados

viernes, 4 de enero de 2019 9:49

A los tipos de referencia se
accede por referencia

→ objects
(arrays)

Técnicamente, el conjunto de valores que constituyen un objeto (o cualquier tipo referencia) se almacenan en un espacio de memoria al que no se puede acceder directamente, sino mediante referencias.

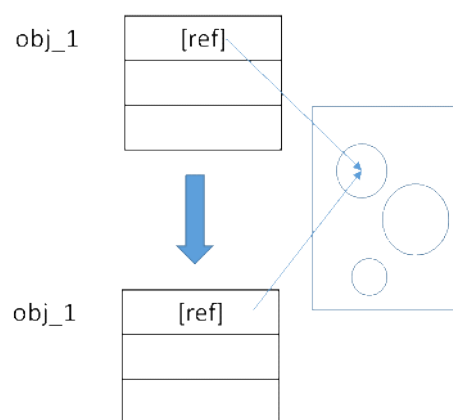
Al manipular un objeto, realmente trabajamos siempre con las referencias a ese objeto, más que con el propio objeto real

Mutabilidad

Al modificar cualquier propiedad de un objeto, añadir propiedades nuevas o eliminar las que existen, cambian realmente los valores almacenados en el espacio de memoria del objeto, por lo que los objetos son **mutables**

```
var obj_1 = {user: "Pepe"}
```

```
var obj_1.edad = 33
```



Asignación a otras variables

Cuando asignamos a un tipo de referencia otra variable del mismo tipo, la asignación se realiza por referencia: las dos variables hacen referencia (apuntan) a un mismo espacio mencionado antes.
Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo referencia. Si se modifica el valor de una de ellas, el valor de la otra variable se verá automáticamente modificado

```
var obj_1 = new Date(2009,11,25);
```

```
// obj_1 = 25 diciembre 2009
```

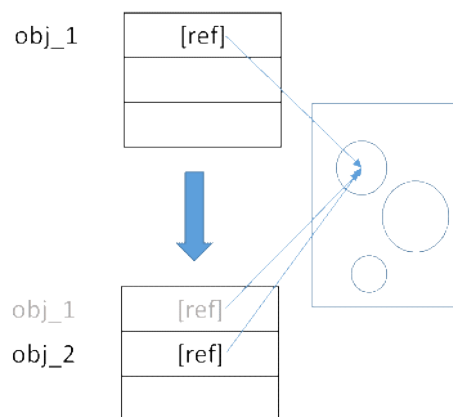
```
var obj_2 = obj_1;
```

```
// obj2 = 25 diciembre 2009
```

```
obj_2.setFullYear(2010,11,31);
```

```
// obj2 = 31 diciembre 2010
```

```
// Ahora obj1 también es 31 diciembre 2010
```



Paso de parámetros

viernes, 4 de enero de 2019 9:57

El paso de parámetros es SIEMPRE por valor

- para los **tipos primitivos** esto supone que **no existe paso por referencia**
- en los **tipos referenciados (objetos)**, el parámetro formal crea una nueva referencia al objeto original, lo que implícitamente supone **siempre un paso por referencia**.

Parámetros y tipos primitivos

```
function addTen(num) {  
    num += 10  
    return num  
}  
  
let count = 20  
let result = addTen(count)  
  
alert(count) //20 - no change  
alert(result) //30
```

Parámetros y tipos referenciados

```
function setName(obj) {  
    obj.name = "Nicholas"  
}  
  
var person = new Object()  
setName(person)  
  
alert(person.name) //"Nicholas"
```

Técnicas de clonado

viernes, 4 de enero de 2019 10:03

Tipos de clonado

deep clone -> se clona el original y todas las referencias incluidas en él

shallow copy -> se clona el original pero se mantienen intactas todas las referencias incluidas en él

Clonado de Arrays (shallow copy)

Usa el método slice() del prototype de Array, con el primer parámetro como 0, y sobreentendiéndose como segundo parámetro la longitud del array

```
let aNumbers1 = [1,2,3]
let aNumbers2 = aNumbers1.slice(0)
```

Comprobamos el resultado

```
aNumbers2.push(4)
console.log(aNumbers1, aNumbers2)

// [ 1, 2, 3 ] [ 1, 2, 3, 4 ]
```

Se trata de una shallow copy

```
let aDatos1 = [{user: 'Pepe'}, {user: 'Juan'}]
let aDatos2 = aDatos1.slice(0)
aDatos2[0].user = 'Jose'
console.log(aDatos1, aDatos2)

// [ { user: 'Jose' }, { user: 'Juan' } ] [ { user: 'Jose' }, { user: 'Juan' } ]
```

Clonado de Arrays y Objetos (deep clone)

Combina los métodos del Objeto JSON, parse() y stringify()
Su limitación es que no existan propiedades que no se serialicen al string de tipo JSON

Se serializa → objetos, arrays, strings, números finitos, true, false, null

Se serializa con limitaciones → NaN, Infinity / -Infinity = null
Objetos Date = formato ISO (se reconstruyen como string)

No se serializa → Funciones, RegExp, errores, undefined

```
aDatos1 = [{user: 'Pepe'}, {user: 'Juan'}]
let aDatos2 = JSON.parse(JSON.stringify(aDatos1))
aDatos2[0].user = 'Jose'
console.log(aDatos1, aDatos2)

// [ { user: 'Pepe' }, { user: 'Juan' } ]
   [ { user: 'Jose' }, { user: 'Juan' } ]
```

```
let oDatos1 = {user: {name: 'Pepe', apell: 'Perez'}, edad: 23}
let oDatos2 = JSON.parse(JSON.stringify(oDatos1))
oDatos1.user.name = 'Jose'
console.log(oDatos1, oDatos2)

// { user: { name: 'Jose', apell: 'Perez' }, edad: 23 }
   { user: { name: 'Pepe', apell: 'Perez' }, edad: 23 }
```

Clonado de Objetos con assign (shallow copy)

El primer parámetro es un objeto vacío, que será retornado por la función asignándole las propiedades del segundo parámetro

```
let oDatos2 = {user: {name: 'Pepe', apell: 'Perez'}, edad: 23}
oDatos3 = Object.assign({}, oDatos2)
oDatos2.user.name = 'Jose'
console.log(oDatos2, oDatos3)

// { user: { name: 'Jose', apell: 'Perez' }, edad: 23 }
// { user: { name: 'Jose', apell: 'Perez' }, edad: 23 }
```

immutableJS

viernes, 4 de enero de 2019 11:11

Librerías: immutableJS (<https://facebook.github.io/immutable-js/>)

Proporciona diversas estructuras de datos persistentes e inmutables:

- List
- Stack
- Map
- OrderedMap
- Set
- OrderedSet
- Record

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 50)

console.log(map1, map2)
// Map { "a": 1, "b": 2, "c": 3 }
   Map { "a": 1, "b": 50, "c": 3 }
```