**Aprende ECMAScript 6 (ES6 o ES2015), el nuevo estándar de JavaScript**

 Friday 06 Feb 2015 -  [22 Comentarios](https://carlosazaustre.es/blog/ecmascript-6-el-nuevo-estandar-de-javascript/#disqus_thread)



25566

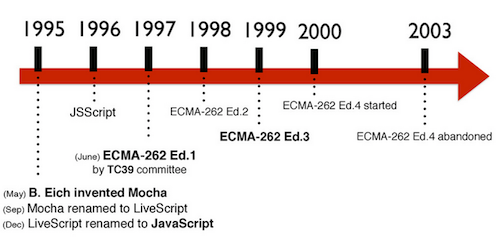
**ECMAScript v6** (Abreviado como ES6 o ES2015) es el estándar que seguirá JavaScript a partir de Junio de este año (2015). Hasta el momento la versión de JS que estamos usando en nuestros navegadores y Node.js, es la v5.

**La evolución de JavaScript**

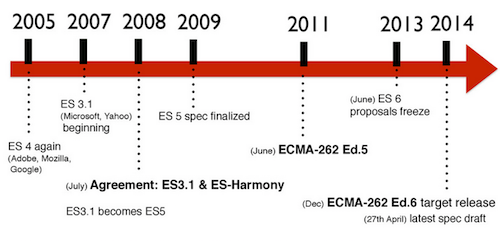
Primero un poco de historia. En 1995 (hace 20 años!) **[Brendan Eich](http://es.wikipedia.org/wiki/Brendan_Eich)** crea un lengujea llamado **Mocha**cuando trabajaba en **Netscape**. En Septiembre de ese año lo renombra a *LiveScript* hasta que le cambiaron el nombre a *JavaScript* debido a una estrategia de marketing, ya que Netscape fue adquirida por *Sun Microsystems*, propietaria del lenguaje **Java**, muy popular por aquel entonces.

**¡Advertencia!**: aclarar lo siguiente Java != JavaScript.

En 1997 se crea un comité (TC39) para estadarizar JavaScript por la *European Computer Manufacturers' Association*, ECMA. Se diseña el estándar del DOM (*Document Object Model*) para evitar incompatibilidades entre navegadores. A partir de entonces los estándares de JavaScript se rigen por ECMAScript.



En 1999 aparece la 3a versión del estándar ECMAScript, que se mantendría vigente hasta hace pocos años. Hubo pequeños intentos de escribir la versión 4, pero hasta 2011 no se aprobó y se estandarizó la versíon 5 (ES5) que es la que usamos hoy en día.



En junio de 2013 quedó parado el borrador de la versión 6, pero en Diciembre de 2014 se aprobó al fin y se espera su estandarización a partir de Junio de 2015.

**Principales novedades de ES6**

ES6 trae muchos cambios significativos al lenguaje. Veamos algunos de ellos:

**Función Arrow**

¿Cuántas veces has programado un código con una estructura similar a la siguiente?

// ES5

// Imaginemos una variable data que incluye un array de objectos

var data = [{...}, {...}, {...}, ...];

data.forEach(function(elem){

// Tratamos el elemento

console.log(elem)

});

Con la función *arrow* => de ES6, el código anterior se sustituiría por:

//ES6

var data = [{...}, {...}, {...}, ...];

data.forEach(elem => {

console.log(elem);

});

Mucho más limpio y claro. CoffeeScript (un metalenguaje que compila a JavaScript) usa algo parecido.

Incluso la podemos utilizar así:

// ES5

var miFuncion = function(num) {

return num + num;

}

// ES6

var miFuncion = (num) => num + num;

**Clases**

Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objectos que realizabamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia.

class LibroTecnico extends Libro {

constructor(tematica, paginas) {

super(tematica, paginas);

this.capitulos = [];

this.precio = "";

// ...

}

metodo() {

// ...

}

}

**This**

La variable this muchas veces se vuelve un dolor de cabeza. antíguamente teníamos que cachearlo en otra variable ya que solo hace referencia al contexto en el que nos encontremos. Por ejemplo, en el siguiente código si no hacemos var that = this dentro de la función document.addEventListener, *this*haría referencia a la función que pasamos por *Callback* y no podríamos llamar a foo()

//ES3

var obj = {

foo : function() {...},

bar : function() {

var that = this;

document.addEventListener("click", function(e) {

that.foo();

});

}

}

Con ECMAScript5 la cosa cambió un poco, y gracias al método bind podíamos indicarle que this hace referencia a un contexto y no a otro.

//ES5

var obj = {

foo : function() {...},

bar : function() {

document.addEventListener("click", function(e) {

this.foo();

}.bind(this));

}

}

Ahora con ES6 y la función *Arrow* => la cosa es todavía más visual y sencilla.

//ES6

var obj = {

foo : function() {...},

bar : function() {

document.addEventListener("click", (e) => this.foo());

}

}

**let y const**

Ahora podemos declarar variables con let en lugar de var si no queremos que sean accesibles más allá de un ámbito. Por ejemplo:

//ES5

(function() {

console.log(x); // x no está definida aún.

if(true) {

var x = "hola mundo";

}

console.log(x);

// Imprime "hola mundo", porque "var" hace que sea global

// a la función;

})();

//ES6

(function() {

if(true) {

let x = "hola mundo";

}

console.log(x);

//Da error, porque "x" ha sido definida dentro del "if"

})();

Ahora con const podemos crear constantes que sólo se puedan leer y no modificar a lo largo del código. Veamos un ejemplo

(function() {

const PI;

PI = 3.15;

// ERROR, porque ha de asignarse un valor en la

// declaración

})();

(function() {

const PI = 3.15;

PI = 3.14159;

// ERROR de nuevo, porque es sólo-lectura

})();

**Template Strings**

Con ES6 podemos interpolar *Strings* de una forma más sencilla que como estábamos haciendo hasta ahora. Fíjate en este ejemplo:

//ES6

let nombre1 = "JavaScript";

let nombre2 = "awesome";

console.log(`Sólo quiero decir que ${nombre1} is ${nombre2`);

// Solo quiero decir que JavaScript is awesome

También podemos tener String multilínea sin necesidad de concatenarlos con +.

//ES5

var saludo = "ola " +

"que " +

"ase ";

//ES6

var saludo = "ola

que

ase";

console.log("hola

que

ase");

**Destructuring**

Tenemos nuevas formas de asignar valores a Arrays y a Objetos. Veamos unos ejemplos

var [a, b] = ["hola", "mundo"];

console.log(a); // "hola"

console.log(b); // "mundo"

var obj = { nombre: "Carlos", apellido: "Azaustre" };

var { nombre, apellido } = obj;

console.log(nombre); // "Carlos"

¿No te ha estallado el cerebro todavía? Pues mira esto:

var foo = function() {

return ["175", "75"];

};

var [estatura, peso] = foo();

console.log(estatura); //175

console.log(peso); //75

**Valores por defecto**

Otra novedad es asignar valores por defecto a las variables que se pasan por parámatros en las funciones. Antes teníamos que comprobar si la variable ya tenía un valor. Ahora con ES6 se la podemos asignar según creemos la función.

//ES5

function(valor) {

valor = valor || "foo";

}

//ES6

function(valor = "foo") {...};

**Módulos**

A esto lo llamo un browserify nativo. Ahora JavaScript se empieza a parecer a lenguajes como *Python* o *Ruby*. Llamamos a las funciones desde los propios Scripts, sin tener que importarlos en el HTML, si usamos JavaScript en el navegador.

//File: lib/person.js

module "person" {

export function hello(nombre) {

return nombre;

}

}

También se puede exportar así, según una aclaración de [Sergio Daniel Xalambrí](https://twitter.com/sergiodxa)

export function hello(nombre) {...}; o también, si solo es una función la que tiene el módulo: export default function(nombre) {...};

Y para importar en otro fichero:

//File: app.js

import { hello } from "person";

var app = {

foo: function() {

hello("Carlos");

}

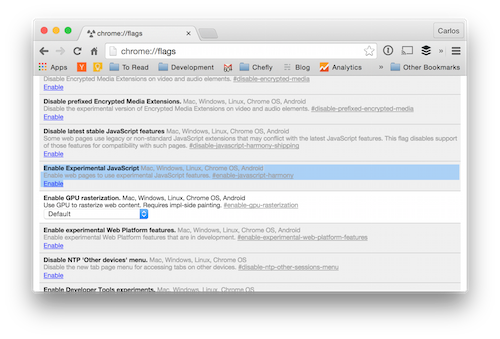
}

export app;

**Empieza a usar ES6 hoy mismo**

Todo esto es muy bonito, pero ¿Cómo podemos empezar a utilizarlo hoy en día?. Lo primero más recomendable es que te instales la última versión de Chrome, que es uno de los navegadores que está implementando las nuevas *features* de ES6 más rápidamente. Te aconsejo incluso que instales **[Chrome Canary](https://www.google.com/chrome/browser/canary.html)**, que es la versión de Chrome que prueba funcionalidades antes de lanzarlas en el Chrome original.

Para probar directamente código ES6 en la consola de tu navegador. Escribe en la barra de direcciones chrome://flags y tendrás una página como ésta:



Y activar el flag **Enable Experimental JavaScript**.

Esto te permitirá probar algunas *features* pero no todas porque aún están en desarrollo.

La otra opción, más extendida, es escribir en ES6 en tus ficheros JavaScript y después "compilarlos" a la versión ES5. Existen herramientas como [6to5](https://6to5.org/docs/using-6to5/) que te permiten lograrlo.

Esto sería un ejemplo de Gulpfile con un plugin de 6to5 que nos "compila" a ES5 y así poder usarlo en el navegador:

$ npm install --save-dev gulp-6to5

var gulp = require("gulp");

var to5 = require("gulp-6to5");

gulp.task("default", function() {

return gulp.src("scripts/app.js");

.pipe(to5())

.pipe(gulp.dest("dist"));

});

Ésto es solo el principio, recuerda que el estándar fue aprobado en diciembre y empezará a implantarse en Junio de este año. Pero no está mal prepararse para lo que viene. Por ejemplo, [AngularJS 2.0 se está reescribiendo en ES6](https://github.com/angular/angular) o [IO.js](http://iojs.org/) (el fork de Node.js) utiliza ES6 y puedes usar las *features* en el servidor

Las clases de javascript son introducidas en el ECMAScript 2015 y son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. La sintaxis de las clases **no** introduce un nuevo modelo de herencia orientada a objetos a JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

## Definiendo clases[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Definiendo_clases)

Las clases son de hecho "[funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions) especiales", tal y como el caso de las [expresiones de funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) y [declaraciones de funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function), la sintaxis de la clase tiene dos componentes:

* [expresiones de clases](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/class) y
* [declaraciones de clases](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class).

### Declaración de clases

Una manera de definir una clase es mediante una declaración de clase. Para la declaración de una clase, es necesario el uso de la palabra reservada class y un nombre para la clase ("Poligono" en esté caso).

class Poligono {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

}

#### Izado (Hoisting)

Una importante diferencia entre las **declaraciones de funciones** y las **declaraciones de clases** es que las **declaraciones de funciones** son [izadas](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) y las **declaraciones de clases** no lo son. En primer lugar necesitas declarar tu clase y luego acceder a ella, de otra modo el ejemplo de código siguiente arrojará un [ReferenceError](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/ReferenceError" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !):

var p = new Poligono(); // ReferenceError

class Poligono {}

### Expresiones de clases

Una **expresión de clase** es otra manera de definir una clase. Las expresiones de clase pueden ser nombradas o anónimas. El nombre dado a la **expresión de clase** nombrada es local dentro del cuerpo de la misma.

// Anonima

var Poligono = class {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

};

// Nombrada

var Poligono = class Poligono {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

};

## Cuerpo de la clase y definición de métodos[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Cuerpo_de_la_clase_y_definición_de_métodos)

El cuerpo de una **clase**es la parte que se encuentra entre las llaves {}. Este es el lugar donde se definen los **miembros de clase,** como los **métodos**o **constructores.**

### Modo estricto

El cuerpo de las declaraciones de clase y las expresiones de clase son ejecutadas en [modo estricto.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

### Constructor

El método [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor) es un método especial para crear e inicializar un objeto creado con una clase. Solo puede haber un método especial con el nombre "constructor" en una clase. Si esta contiene mas de una ocurrencia del método **constructor,**se arrojará un Error [SyntaxError](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/SyntaxError" \o "El objeto SyntaxError representa un error cuando se trata de interpretar codigo sintacticamente invalido.)

Un **constructor** puede usar la palabra reservada **super**para llamar al **constructor**de una superclase

### Métodos prototipo

Vea también [métodos definidos](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions).

class Poligono {

constructor(height, width) {

this.height = height;

this.width = width;

}

get area() {

return this.calcArea();

}

calcArea() {

return this.height \* this.width;

}

}

const cuadrado = new Poligono(10, 10);

console.log(cuadrado.area);

### Métodos estáticos

La palabra clave [static](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static) define un método estático para una clase. Los métodos estáticos pueden ser llamados sin instanciar la clase ni una vez instanciada. Los métodos estáticos son a menudo usados para crear funciones de utilidad para una aplicación.

class Punto {

constructor(x, y) {

this.x = x;

this.y = y;

}

static distancia(a, b) {

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.sqrt(dx\*dx + dy\*dy);

}

}

const p1 = new Punto(5, 5);

const p2 = new Punto(10, 10);

console.log(Punto.distancia(p1, p2));

### "Boxing" con prototipos y métodos estáticos

Cuando un métodos estático o del prototipo es llamado sin un objecto evaluado "this" (o con "this" como booleano, cadena, número, undefined o null), entonces el valor de "this" será **undefined** dentro de la funciona llamada. Autoboxing no ocurrirá. El comportamiento será igual inclose si se escribe el código en modo no estricto.

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

let speak = obj.speak;

speak(); // undefined

let eat = Animal.eat;

eat(); // undefined

Si se escribe el código del cuadro superior usando clases función tradicionales, entonces autoboxing ocurrirara porque tomará valor de "this" sobre la función que es llamada.

function Animal() { }

Animal.prototype.speak = function(){

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // global object

let eat = Animal.eat;

eat(); // global object

## Subclases con extends[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Subclases_con_extends)

La palabra clave [extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends) es usada en declaraciones de clase o expresiones de clase para crear una clase hija.

class Animal {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' hace un ruido.');

}

}

class Perro extends Animal {

hablar() {

console.log(this.nombre) + ' ladra.';

}

}

También se pueden extender las clases tradicionales basadas en funciones:

function Animal (nombre) {

this.nombre = nombre;

}

Animal.prototype.hablar = function () {

console.log(this.nombre + 'hace un ruido.');

}

class Perro extends Animal {

hablar() {

super.hablar();

console.log(this.nombre + ' ladra.');

}

}

var p = new Perro('Mitzie');

p.hablar();

Fijarse que las clases no pueden extender objectos regulares (literales). Si se quiere heredar de un objecto regular, se debe user [Object.setPrototypeOf()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/setPrototypeOf" \o "El método Object.setPrototypeOf() establece el prototipo (p.e., la propiedad interna [[Prototype]]) de un objeto especificado a otro objeto o sino establece null.)::

var Animal = {

hablar() {

console.log(this.nombre + 'hace ruido.');

}

};

class Perro {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' ladra.');

}

}

Object.setPrototypeOf(Perro.prototype, Animal);

var d = new Perro('Mitzie');

d.hablar();

## Especies[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Especies)

Quizás se quiera devolver objectos [Array](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array" \o "El objeto de JavaScript Array es un objeto global que es usado en la construcción de arreglos, que son objetos tipo lista de alto nivel.) derivados de la clase array MyArray. El patron species permite sobreescribir constructores por defecto.

Por ejemplo, cuendo usando metodos del tipo [map()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map" \o "El método map() crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.) que devuelve el constructor por defecto, se quiera que esos métodos devulvan un objeto padre Array, en cambio de MyArray. El símbolo [Symbol.species](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Symbol/species" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !) permite hacer:

class MyArray extends Array {

// Sobreescribe species sobre el constructor padre Array

static get [Symbol.species]() { return Array; }

}

var a = new MyArray(1,2,3);

var mapped = a.map(x => x \* x);

console.log(mapped instanceof MyArray); // false

console.log(mapped instanceof Array); // true

## Objetos de subclase built-in[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Objetos_de_subclase_built-in)

TBD

## Super class calls with super[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Super_class_calls_with_super)

La palabra clave [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) es usada para llamar funciones del objeto padre.

class Gato {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' hace ruido.');

}

}

class Leon extends Gato {

hablar() {

super.hablar();

console.log(this.nombre + ' maulla.');

}

}

## Mix-ins[EDIT](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes$edit#Mix-ins)

Subclases abstractas or mix-ins son plantillas de clases. Una clase ECMAScript solo puede tener una clase padre, con lo cual la herencia multiple no es posible. La funcionalidad debe ser proporcionada por la clase padre.

Una función con una clase padre como entrada y una subclase extendiendo la clase padre como salida puede ser usado para implementar mix-ins en EMCAScript:

var calculatorMixin = Base => class extends Base {

calc() { }

};

var randomizerMixin = Base => class extends Base {

randomize() { }

};

Una clase que use este método puede ser escrita tal que así:

class Foo { }

class Bar extends calculatorMixin(randomizerMixin(Foo)) { }