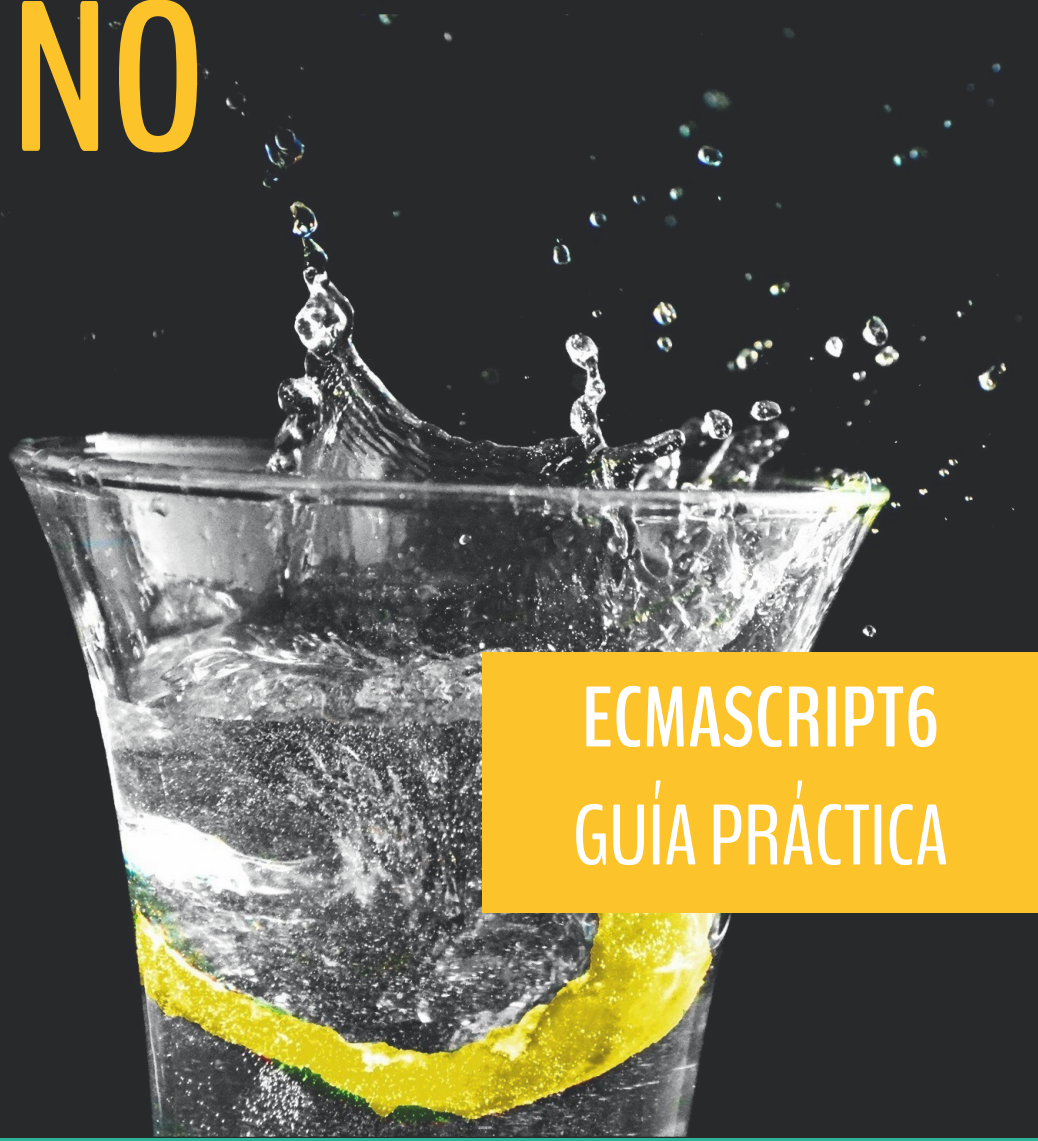
APRENDE ES6, JAVASCRIPT MODERNO



ECMASCRIPT6
GUÍA PRÁCTICA

ENRIQUE ORIOL

Aprende ES6, Javascript moderno - ECMAScript6
Guía práctica
por Enrique Oriol

Acerca del autor

Enrique Oriol es Ingeniero Superior de Telecomunicaciones y trabaja como director técnico en una startup de Barcelona. Con varios años de experiencia desarrollando software en entornos *mobile*, *backend* y *frontend*, dedica su tiempo libre contribuyendo a la comunidad con publicaciones educativas y a experimentar con las últimas tendencias de software.

Publicado en 2016, por Enrique Oriol
http://www.blog.enriqueoriol.com

¿Qué es ECMAScript 6?

ECMAScript 6, también conocido como **ECMAScript 2015** o **ES6**, es la nueva versión de Javascript, aprobada en Junio 2015, y en la que se lleva trabajando desde 2011.

Se podría considerar que es una auténtica **revolución en la sintaxis de Javascript**. Su buque insignia es, probablemente, una clara **orientación a clases y herencia**, pero la verdad es que hay muchas otras novedades interesantes, como el uso de **módulos**, los **parámetros por defecto**, las variables **let** y **const**, o la novedosa sintaxis de las **funciones arrow**, entre otros cambios.

¿Se soporta ES6 actualmente?

Los **principales navegadores** ya implementan la **mayoría de funcionalidades** (ver compatibilidad actual con ES6), si bien aún están en proceso de adaptación, por lo que es recomendable utilizar un transpilador como Babel para convertir nuestro elegante ES6 en ES5 (el JS antiguo), y asegurar que nuestro codigo se podrá ejecutar en todos los navegadores.

Dicho esto, ya **NO tienes excusa para retrasar tu aprendizaje de *ES6***: Vas a disponer de herramientas más modernas para desarrollar tu código, y gracias a los transpiladores funcionará en todos los navegadores.

Principales novedades de ES6

Vamos a ver las novedades más revolucionarias de esta nueva versión de Javascript.

Variable **const**

De forma análoga a otros lenguajes, se ha definido un tipo de varaible que **solo puede asignarse en su declaración**, y **no puede ser modificada**.

```
const URL = 'www.mydomain.com';
URL = 'whatever'; // ERROR!!
```

> Como en el resto de javascript, si se define dentro de un scope, solo pertenecerá a ese scope.

Variable **let**

La variable tipo **let**, a diferencia de **var** no puede ser accesible más allá de su scope.

```
(function() {
    console.log(global); // undefined
    console.log(local); // undefined
    if(true) {
        var global = "I'm global";
        let local = "I'm only local";
    }
    console.log(global); // I'm global
    console.log(local); //undefined
})();
```

Podemos observar como fuera del scope *if(true){...}*, la variable definida con **let** no existe, mientras que la definida con **var**, ha sido asignada al objeto raíz **window** por lo que podemos usarla como una variable global fuera de su scope.

Función arrow

Las funciones arrow proporcionan una sintaxis más compacta para la definición de funciones. Pongamos algunos ejemplos.

Si no hay argumentos, empezamos con 2 paréntesis y la flecha:

```
// ES5
setInterval(function(){
    console.log('hi world');
}, 100);
```

```
// ES6
setInterval(() => {
    console.log('hi world');
}, 100);
```

Con un argumento, no usamos paréntesis:

```
// ES5
var vowels = ['a', 'e', 'i', 'o', 'u'];
vowels.forEach(function(value){
    console.log('vowel :' + value);
});
```

```
// ES6
var vowels = ['a', 'e', 'i', 'o', 'u'];
vowels.forEach(value => {
    console.log('vowel :' + value);
});
```

Con más de un argumentos, volvemos a los paréntesis:

```
// ES5
var sum = function(a, b){
    return a+b;
}
```

```
// ES6
var sum = (a, b) => a + b;
```

Parámetros por defecto

Por fin **podemos incluir valores por defecto en nuestros parámetros**, como en otros lenguajes de programación. Podemos incluso referenciar otros parámetros:

```
// ES6
function greet(name, gender = 'Mr.', greeting = 'Hello ' +
gender){
    console.log(greeting + ' ' +  name);
};

greet('Peter'); // Hello Mr. Peter
greet('Alex', undefined, 'Whats up'); //Whats up Alex
```

Parámetros Rest

Hasta ahora, cuando pasábamos argumentos a una función, se **añadía** una variable **arguments** que incluía todos los parámetros (definidos o no) que había recibido nuestra función.

```
// ES5
function printName(name){
  var length = arguments.length;
  var fullName = name;
    if(length > 1){
        for(var i=1; i< length; i++){
            fullName += ' ' + arguments[i];
```

```
        }
    }
  console.log(fullName);
};

printName('Felipe'); // Felipe
printName('Felipe', 'Juan', 'Froilan'); //Felipe Juan Froilan
```

Los **parámetros Rest** nos proporcionan una manera de **pasar un conjunto indeterminado de parámetros** que la función agrupa en forma de Array. Como detalle (de lógica), solo puede ser parámetro rest el último argumento de la función. Veamos mejor a qué me refiero.

```
// ES6
function printName(name, ...fancyNames){
  var fullName = name;
  fancyNames.forEach(fancyN => fullName += ' ' + fancyN);

  console.log(fullName);
};

printName('Felipe'); // Felipe
printName('Felipe', 'Juan', 'Froilan'); //Felipe Juan Froilan
```

Clases y herencia

¿Cuantas veces has deseado utilizar clases como dios manda en JS, en lugar de ensuciar el código con **prototype** y demás historias? **Pués estás de suerte**.

Donde en ES5 creamos clases y herencia del siguiente modo:

```
// ES5
//Class creation
function Document(title, author, isPublished) {
  this.title = title;
```

```
  this.author = author;
  this.isPublished = isPublished;
}
Document.prototype.publish = function publish() {
  this.isPublished = true;
};

//Class inherittance
function Book(title, author, topic) {
  Document.call(this, title, author, true);
  this.topic = topic;
}
Book.prototype = Object.create(Document.prototype);
```

En **ES6** lo podemos hacer con una **sintaxis más clara y explícita**:

```
// ES6
//Class creation
class Document {
  constructor(title, author, isPublished) {
    this.title = title;
    this.author = author;
    this.isPublished = isPublished;
    }
  publish(){
        this.isPublished = true;
    }
}

//Class inherittance
class Book extends Document{
  constructor(title, author, topic){
    super(title, author, true);
    this.topic = topic;
  }
}
```

Módulos

Hasta ahora, los desarrolladores de Javascript habían cubierto su anhelo de modularizar el código gracias a librerías como *RequireJS* o *browserify*. **ES6 incluye la funcionalidad de módulos**, que nos permite exportar/importar objetos, funciones y clases desde código, sin tener que importarlos desde HTML.

Veamos como exportar un módulo, en el archivo *lib/greetings.js*:

```
// ES6
// lib/utils.js
module "utils" {
    export function greeting(name){
        console.log("Hi! " + name);
    }
}
```

Y como importarlo desde otro archivo JS:

```
// ES6
// app.js
import { greeting } from "utils";
var app = {
    welcome: function(){
        greeting("Mike");
    }
}
export app;
```

También podemos importar los módulos por su **path**, sin necesidad de definir explícitamente el módulo, así como definir un método **default** que se asigna a la variable que definimos a continuación del import:

```
// ES6
// lib/math.js
export function mult(a, b){
    return a*b;
```

```
}

export const PI = 3.141593;

export default function(a, b){
    return a + b;
}
```

Podemos encontrar documentación muy detallada al respecto. A continuación dejo las posibles formas de importar módulos:

```
// ES6
import defaultMember from "module-name";
import * as name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import defaultMember, { member [ , [...] ] } from "module-name";
import defaultMember, * as name from "module-name";
import "module-name";
```

Operador de propagación (Spread operator)

El **spread operator** lo que nos permite es pasar un array de elementos a una función, convirtiendo cada uno de los elementos en un argumento. Se podría pensar en el spread operator como la versión inversa de los parámetros rest. Lo vemos mejor con un ejemplo.

Antes (**en ES5**), para pasar un array de elementos a una función como parámetros, usaríamos el método **apply** del siguiente modo:

```
//ES5
function f(x, y, z) { }
var args = [0, 1, 2];
f.apply(null, args);
```

Ahora en cambio lo podemos hacer poniendo 3 puntos delante del array, es decir, usando el **spread operator**:

```
//ES6
function f(x, y, z) { }
var args = [0, 1, 2];
f(...args);
```

Además, cualquier argumento puede aprovecharse de esta característica, con lo que podríamos sacar ventaja para cosas como las siguientes:

```
//ES6
//example1
function f(v, w, x, y, z) { }
var args = [0, 1];
f(-1, ...args, 2, ...[3]);

//example2
var parts1 = ['shoulder', 'knees'];
var parts2 = ['chest', 'waist'];
var lyrics = ['head', ...parts1, ...parts2, 'and', 'toes'];
//lyrics = ['head', 'shoulder', 'knees', 'chest', 'waist', 'and',
'toes']
```

> En **ES5** no es posible combinar los métodos *apply* y *new*, mientras que **ES6 permite combinar *new* con el *spread operator***:

```
//ES6
var d = new Date(...dateFields);
```

Destructuring

El destructuring nos permite descomponer un array u objeto para asignarlo a un conjunto

de variables.

```
//ES6
//destructuring array
var a, b, rest;
[a, b] = [1, 2];

//destructuring array
var foo = function() {
    return [175, 75];
};
var [height, weight] = foo();
```

En el caso de **descomponer** las propiedades de un **objeto**, es importante que las **variables** a las que van a parar tengan el **mismo nombre que las propiedades** que queremos asignar

```
//ES6
//destructuring object
({a, b} = {a:1, b:2})

//destructuring user
var user = {
    name: 'Peter',
    surname: 'Griffin'
};
var { name, surname } = user;
```

Además, podemos combinar el *destructuring* con el *spread operator* para hacer una asignación como la siguiente:

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

Template Literals

Los **template literals** (también denominados *template strings* en las primeras revisiones de ES6) son literales de texto que nos permiten embeber expresiones, utilizar varias lineas e

interpolar expresiones.

Veamos un ejemplo de interpolación:

```
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and not ${2 * a + b}.`); //
'Fifteen is 15 and not 20.'
```

Y un ejemplo de string multilinea:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Tagged template literals

Esto vendría a ser una forma más compleja de **template literal**, que nos permite modificar lo que devuelve un template literal a través de una función. La función de un **tagged template literal** recibe en primer lugar un argumento que contiene un array con los elementos del *template literal* que son literales, mientras que los siguientes argumentos se corresponden a los valores interpolados del *template literal*. Vamos a ver un ejemplo:

```
var name = 'Peter';
var last_name = 'Griffin';

function sayHello(strings, ...values) {
  console.log(strings[0]); // "Name "
  console.log(strings[1]); // ", surname "
  console.log(values[0]);  // Peter
  console.log(values[1]);  // Griffin

  return `Hello ${values[0]} ${values[1]}`;
}

var greeting = sayHello`Name ${name}, surname ${last_name}`;
```

```
console.log(greeting);// Hello Peter Griffin
```

En la web de Mozilla encontrarás más información sobre los *template literals*.

For … of loop

El **for...of loop** nos permite crear un bucle de iteración a través de **colecciones** (Array, string, Map, Set, …). Este tipo de bucle es equivalente al que podríamos hacer con un **forEach**, pero la sintaxis es más similar a bucles for en otros lenguajes, como por ejemplo *Python*.

```
//ES5
var numbers = [1,2,3,4,5];
numbers.forEach(function(value) {
    console.log(value);
});
//1, 2, 3, 4, 5
```

```
//ES6
var numbers = [1,2,3,4,5];
for(let item of numbers){
    //remember let is useful to define local vars
    console.log(item);
});
//1, 2, 3, 4, 5
```

```
//ES6
var word = "foo";
for(let item of word){
    console.log(item);
});
//"f", "o", "o"
```

> La principal diferencia entre **for...in** y **for...of** es que el primero itera entre **todas** las propiedades enumerables de

> un objeto, mientras que el segundo no funciona con todos los objetos, sino que es especifico de **colecciones**, es decir, solo itera sobre los elementos de cualquier colección que contenga la propiedad **Symbol.iterator**

Comparemos la diferencia entre **for..in** y **for..of**:

```
//ES6
let iterable = [3, 5, 7];
iterable.foo = "hello";

for (let i in iterable) {
  console.log(i); // logs 0, 1, 2, "foo"
}

for (let i of iterable) {
  console.log(i); // logs 3, 5, 7
}
```

Block level function declarations

Con ES6 podemos declarar funciones a nivel de bloque de forma segura (ES5 lo desaconsejaba, por que sus scopes está diseñados a nivel de función).

Si miramos el siguiente ejemplo, veremos que en **ES5** el log de **f()** siempre es 2:

```
//ES5
function f() { return 1; }
{
    console.log(f());  // 2
    function f() { return 2; }
    console.log(f());  // 2
}
console.log(f()); // 2
```

Lo que está pasando es que las llaves no crean un scope nuevo, y por tanto la función f() se

redefine para el scope global.

En cambio, si hacemos lo mismo en ES6, vemos que **f()** devuelve el valor 2 dentro de las llaves, pero el valor 1 fuera.

```
//ES6

function f() { return 1; }
{
    console.log(f());  // 2
    function f() { return 2; }
    console.log(f());  // 2
}
console.log(f()); // 1
```

Esto es por que **en ES6**, como en muchos otros lenguajes de programación, **el bloque que se define entre llaves genera un nuevo scope** (block scope).

Generadores

Los **generadores** son funciones de las que se puede salir y volver a entrar y que conservan su contexto entre las reentradas. Así de primeras parece extraño, pero no lo es tanto, si vemos a los generadores como una **herramienta para construir iteradores**.

Un generador se declara con **function*** (la palabra clave function seguida de una asterisco). También se pueden definir funciones generadoras usando el constructor **GeneratorFunction** y una function* expression.

La **llamada a una función generadora no se ejecuta inmediatamente, sino que devuelve un objeto iterador**. Cuando llamamos al metodo **next()** del iterador, se ejecuta el cuerpo de la función hasta la primera expresión **yield**, que determina el valor a devolver (o se delega con **yield*** a otro generador).
El método **next()** devuelve un objeto con 2 propiedades:

- **value**: El valor que devuelve la expresión *yield*
- **done**: Indica si es el último *yield* del generador.

Vamos a clarificarlo con un ejemplo, donde aprovechamos para recordar también algo de *destructuring*:

```
function* idMaker(){
    var index = 0;
    while(index < 3)
        yield index++;
    yield "end";
}

var gen = idMaker();

while(true){
    let {value, done} = gen.next();

    if(done)
        break;

    console.log(value);
}
// 0, 1, 2, end
```

> **DETALLE:** NO puedes utilizar **new** con los generadores, no son constructores. *var obj = new idMarker;* lanzaría un error.

Para completar el tema de los generadores, vamos a ver un ejemplo de un generador que referencia a otro con **yield***. Además, aprovecho para recordar que al devolver un iterador, puede aprovecharse del bucle **for...of**:

```
var anotherGenerator = function*(i) {
    yield i + 1;
    yield i + 2;
    yield i + 3;
}

function* generator(i){
    yield i;
    yield* anotherGenerator(i);
    yield i + 10;
}

for(let val of generator(10)){
    console.log(val);
```

```
}
//10, 11, 12, 13, 20
```

Mapas y Sets

ES6 incorpora 4 nuevas estructuras de datos, que son **Map**, **WeakMap**, **Set** y **WeakSet**. Si has trabajado con lenguajes como *Java* o *Python* ya te harás una idea de para que sirven. Vamos a repasarlos.

Map

El objecto **Map** nos permite relacionar (*mapear*) unos valores con otros como si fuera un diccionario, en formato **clave/valor**. Cualquier valor (tanto objetos como valores primitivos) puede ser usados como clave o valor.

Los *Maps* nos permiten, por ejemplo, saber de inmediato si existe una clave o borrar un par clave/valor concreto:

```
//ES6
let map = new Map();
map.set('foo', 123);
let user = {userId: 1};
map.set(user, 'Alex');

map.get('foo'); //123
map.get(user); //Alex

map.size; //2

map.has('foo'); //true

map.delete('foo'); //true
map.has('foo'); //false
```

```
map.clear();
map.size; //0
```

Además, podemos crear *Maps* a partir de un array de pares:

```
map = new Map([['user1','Alex'], ['user2', 'Vicky'], ['user3',
'Enrique']]);

for(let [key, value] of map){
  console.log(key, value);
}

//"user1" "Alex"
//"user2" "Vicky"
//"user3" "Enrique"

map.keys(); //iterator with keys
map.values(); //iterator with values
map.entries(); //iterator with pair [key, value]
```

En la documentación de Mozilla sobre Map puedes ver todas las propiedades del objeto Map.

WeakMap

Los **WeakMaps** son similares a los **Maps**, pero con algunas diferencias:

- Un **WeakMap solo acepta objetos como claves**
- La **referencia a las claves es débil**, lo que significa que **si no hay otras referencias al objeto** que actúa como **clave**, el **garbage collector podrá liberarlo**.

> **Debido a** que usa **referencias débiles**, un *WeakMap* **NO** dispone del método **.keys()** para recuperar las claves, **NI** de propiedades o métodos relacionados con más de un

> elemento a la vez, como **.values(), .entries(), .clear() o .size**. Tampoco podemos iterar un **WeakMap** con el bucle **for of**.

Veamos un ejemplo práctico:

```
let key = {userId:1};
let key2 = {userId:2};
let weakmap = new WeakMap();

weakmap.set(key,"Alex");
weakmap.has(key); //true
weakmap.get(key); //Alex
weakmap.delete(key); // true
weakmap.get(key); //undefined

weakmap.set(key2,"Vicky");
weakmap.size; //undefined
key2=undefined;
weakmap.get(key2); //undefined
```

Set

Los **sets son conjuntos de elementos no repetidos**, que pueden ser tanto objetos, como valores primitivos.

Tiene métodos equivalentes a un **Map**, con la diferencia que utilizamos **add** para añadir elementos, y de que en un set las **keys** y los **values** son lo mismo, el valor del objeto. Del mismo modo, **.entries()** devuelve una pareja *[value, value]*

```
let set = new Set();
set.add('foo');
set.add('bar');
set.size  //2

for(let item of set){
  console.log(item);
```

```
}
//"foo"
//"bar"

for(let item of set.entries()){
  console.log(item);
}
//["foo", "foo"]
//["bar", "bar"]

set.has('foo');  //true
set.delete('foo');  //true
set.has('foo');  //false
set.size  //1
set.clear();
set.size  //0
```

Otras formas de definir un set es a través de un array o bien concatenando el método add.

```
let set1 = new Set(['foo', 'bar']);
for(let i of set1){
  console.log(i)
}
//foo
//bar

//remember set elements are unique
let set2 = new Set().add('foo').add('bar').add('bar');
for(let i of set2){
  console.log(i)
}
//foo
//bar
```

WeakSet

Nos encontramos con una situación análoga al *WeakMap*, pero con los *Sets*. Las dos principales diferencias de un *WeakSet* respecto a un *Set* son:

- Los *WeakSets* **únicamente** pueden contener **colecciones de objetos**.
- La **referencia** a los objetos es **débil**, por lo que si no hay otra referencia a uno de los objetos contenidos en el *WeakSet*, el *garbage collector* lo podrá liberar. Esto implica que:
    - No hay una lista de objetos almacenados en la colección
    - Los *WeakSet* no son enumerables.

Básicamente, los métodos de los que dispone un *WeakSet* son:

- add()
- delete()
- has()

```
let obj = ['foo', 'bar'];
let ws = new WeakSet();
ws.add(obj);

ws.has(obj); //true

obj = undefined;
ws.has(obj); //false
ws.delete(obj); //false
```

Typed Arrays

Otra de las novedades que incorpora **ECMAScript 2015** es la de vectores tipados (**Typed arrays** en inglés).

Estos objetos son del tipo **Array** y nos proporcionan un mecanismo para acceder a datos binarios puros, lo que nos puede ayudar a la hora de manipular directamente datos crudos de audio o vídeo, trabajar con WebSockets, etc.

> Los vectores tipados no deben confundirse con vectores normales: la llamada **Array.isArray()** en un ***typed array*** devuelve **false**. Del mismo modo, algunos métodos de arrays

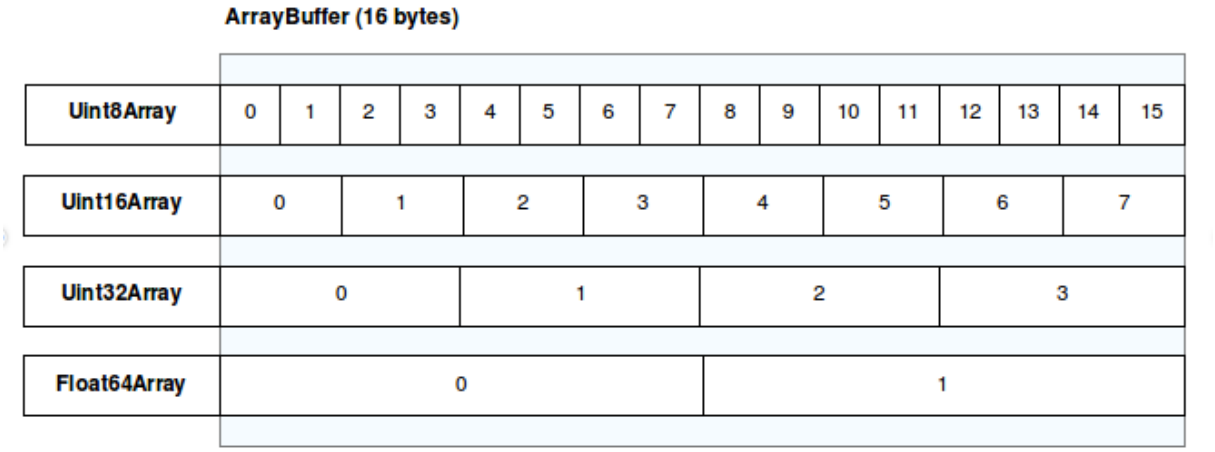
> normales, no están disponibles en los *typed arrays*, como *push* o *pop*.

Arquitectura de los *typed arrays*

Los *typed arrays* separan su implementación en 2 elementos:

- **buffer**: (objeto **ArrayBuffer**) Es el objeto que representa un segmento de datos de tamaño fijo, sin un formato de referencia ni mecanismo de acceso a su contenido.
- **vistas**: La vista proporciona un contexto (tipo de dato, desplazamiento y número de elementos) para obtener los datos en un vector tipado real.

La siguiente imagen de la documentación de *Mozilla Foundation* nos permite entender mejor la **relación entre buffers y vistas**



Como vemos, el **ArrayBuffer es un buffer de dimensión fija** donde se almacenan los datos binarios, mientras que **Uint8Array**, **Uint16Array**, etc. **son vistas** de vectores tipados para diferentes tipos numéricos (o contextos), es decir, **definen de qué modo deben agruparse los bytes de su buffer a la hora de acceder a ellos**.

Tipos de vistas

Para leer y escribir el contenido de un ArrayBuffer, creas una vista de vector tipado o bien una *DataView* que represente el buffer en un formato específico.

Vistas de vector tipado

Las vistas de vector tipado tienen nombres autodescriptivos, y proporcionan contextos para los tipos numéricos más habituales.
Cabe destacar el **Uint8ClampedArray**, que restringe los valores entre **0 y 255** (util para procesar imágenes, como datos de Canvas, por ejemplo).

Type	Size in bytes	Description	Web IDL type
Int8Array	1	8-bit two's complement signed integer	byte
Uint8Array	1	8-bit unsigned integer	octet
Uint8ClampedArray	1	8-bit unsigned integer (clamped)	octet
Int16Array	2	16-bit two's complement signed integer	short
Uint16Array	2	16-bit unsigned integer	unsigned short
Int32Array	4	32-bit two's complement signed integer	long
Uint32Array	4	32-bit unsigned integer	unsigned long
Float32Array	4	32-bit IEEE floating point number	unrestricted float
Float64Array	8	64-bit IEEE floating point number	unrestricted double

DataView

Un **DataView** es una interface de bajo nivel que **proporciona una API getter/setter para leer y escribir datos en el buffer** de forma arbitraria.
Una de las ventajas que nos ofrece un **DataView**, es que puedes controlar el orden de los

bytes, lo que permite modificar los getters/setters para pasar de big-endian (por defecto) a little-endian, por ejemplo.

> Las vistas de vector tipado están en la **Endianness** de tu plataforma, mientras que el en **DataView**, se trabaja por defecto en **big-endian**.

Trabajando con Typed Arrays

A continuación, un ejemplo de como trabajar con *typed Arrays* mediante vistas de vector tipado, a partir de un *ArrayBuffer*:

```
let buffer = new ArrayBuffer(16);
console.log(buffer.byteLength); //16

//necesitamos una vista para poder leer/escribir del buffer.
Creamos una vista de 4 bytes por elemento
var int32View = new Int32Array(buffer);

//Inicializamos cada elemento de la vista
for (var i = 0; i < int32View.length; i++) {
  int32View[i] = 3-i;
}

//el buffer contiene los valores [3, 2, 1, 0]
for(let item of int32View){
  console.log(item);
}//3, 2, 1, 0
```

No obstante, podemos inicializar directamente una vista de vector tipado a partir de un array tradicional:

```
let typedArray = new Uint8Array([0,1,2]);
console.log(typedArray.length); // 3
typedArray[0] = 5;
```

```
for(let item of typedArray){
  console.log(item);
}//5, 1, 2

let normalArray = [...typedArray]; // [5,1,2]
```

Y así es como trabajaríamos con un DataView:

```
let typedArray = new Uint8Array([0,1,2]);
//Obtenemos un buffer, ya sea a través de ArrayBuffer, o mediante
una vista de vector tipado
let dataView = new DataView(typedArray.buffer);

//definimos cómo queremos acceder al contenido
console.log(dataView.getUint8(0)); // 5
```

Conclusiones

Los *typed arrays* nos facilitan el acceso a datos binarios, lo cual puede ser muy util de cara a trabajar con imágenes, o manipulando ficheros en el dispositivo, ya sea una aplicación en local o estemos trabajando en la parte de servidor con Node.js.

En la documentación de Mozilla Foundation para vectores tipados puedes encontrar más detalles.

Proxies

En nuestro repaso por las novedades de ECMAScript 2015 nos encontramos con el objeto *Proxy*. **El objeto *Proxy* se utiliza para personalizar los comportamientos de operaciones básicas** como:

- observación de propiedades
- asignaciones
- enumeraciones

- funciones invocación
- etc.

Esto lo hace interceptando las llamadas a dichas operaciones y manipulando la respuesta.

> Los Proxies de ES6 no tienen forma de ser definidos mediante ES5, por lo que es una **característica NO SOPORTADA por** los **transpiladores** como Babel, ni mediante polyfills. La única forma de utilizar proxies, de momento, es que lo soporte directamente el navegador.

Definición

Un proxy se define mediante un **handler** (objeto que gestiona las intercepciones a propiedades del *proxy*) y un **target** (objeto sobre el que queremos construir el *proxy*)

La sintaxis de declaración de un proxy sería la siguiente:

```
let proxy = new Proxy(target, handler);
```

También podríamos crear un proxy revocable, es decir, que quedará inservible después de utilizar el método *revoke()*, devolviéndonos un *TypeError* cuando se intenten utilizar sus interceptores. Veamos como usarlo:

```
var revocable = Proxy.revocable({}, {
  get: function(target, prop) {
    return "property: [" + prop + "]";
  }
});
var proxy = revocable.proxy;
console.log(proxy.foo); // "property [foo]"

revocable.revoke();
```

```
console.log(proxy.foo); // lanza TypeError
proxy.foo = 1           // lanza TypeError
delete proxy.foo;       // lanza TypeError
typeof proxy            // en este caso no se utiliza el
interceptor del proxy, por lo que no hay error
```

Veamos ahora un ejemplo básico donde interceptamos el método de acceso a las propiedades del objeto, de modo que si la propiedad no existe, nos devuelva el valor "-1":

```
var handler = {
    get: function(target, propertyKey){
        return name in propertyKey?
            propertyKey[name] : -1;
    }
};

let proxy = new Proxy({}, handler);

p.a = 1;
p.b = undefined;

console.log(p.a, p.b, p.c);
//1, undefined, -1
```

Interceptores

Los métodos del *handler* con los que podemos interceptar al *proxy* son los siguientes:

- **handler.getPrototypeOf(target)**: Intercepta Object.getPrototypeOf.
- **handler.setPrototypeOf(target, prototype)**: Intercepta Object.setPrototypeOf.
- **handler.isExtensible(target)**: Intercepta Object.isExtensible.
- **handler.preventExtensions(target)**: Intercepta Object.preventExtensions.
- **handler.getOwnPropertyDescriptor(target, property)**: Intercepta Object.getOwnPropertyDescriptor.

- **handler.defineProperty(target, property, descriptor)**: Intercepta Object.defineProperty.
- **handler.has(target, property)**: Intercepta el operador *in*.
- **handler.get(target, property, receiver)**: Intercepta el *getter* de las propiedades.
- **handler.set(target, property, value, receiver)**: Intercepta el *setter* de las propiedades
- **handler.deleteProperty(target, property)**: Intercepta el operador *delete*.
- **handler.ownKeys(target)**: Intercepta Object.getOwnPropertyNames.
- **handler.apply(target, thisArg, argumentsList)**: Intercepta la función *call*.
- **handler.construct(target, argumentsList, newTarget)**: Intercepta el operador *new*.

Ejemplos de uso de proxies

Veamos un par de ejemplos.

En primer lugar, un Proxy de redirección: Lo único que hace es redirigir todas las operaciones que se hacen sobre el mismo, hacia el target al que apunta:

```
var target = {};
var p = new Proxy(target, {});

p.a = 37; // operation forwarded to the target

console.log(target.a); // 37. The operation has been properly
forwarded
```

Y otro ejemplo sería un Proxy de validación

```
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('La edad tiene que ser un valor
```

```
entero');
      }
      if (value < 0) {
        throw new RangeError('Edad inválida');
      }
    }

    // comportamiento por defecto del setter
    obj[prop] = value;
  }
};

let person = new Proxy({}, validator);

person.age = 100;
console.log(person.age); // 100
person.age = 'young'; // Lanza excepción
person.age = -5; // Lanza excepción
```

Puedes encontrar más ejemplos en la documentación de *Mozilla Foundation* para el objeto Proxy.

Conclusiones

Los proxies nos permiten alterar el comportamiento de algunas operaciones básicas de los objetos, **pero ten cuidado** si los quieres utilizar en webs hoy en día, ya que aún no está soportado por todos los navegadores, y esta característica no se puede emular en ES5 con transpiladores o polyfills.

Promises

Probablemente ya sabes lo que es -en Javascript, se entiende- una promesa (o **promise** en inglés), pero hasta la llegada de ES6, javascript no incorporaba *promises* por defecto, y para usarlas tenías que cargar librerías como *JQuery* o frameworks como *AngularJS*.

ES6 incorpora su propio mecanismo de Promises, y en este POST vas a ver como

funcionan.

Promises, qué son

Las promesas se utilizan para la realización de tareas asíncronas, como por ejemplo la obtención de respuesta a una petición HTTP.

Una promesa puede tener 4 estados:

- **Pendiente**: Es su estado inicial, no se ha cumplido ni rechazado.
- **Cumplida**: La promesa se ha resuelto satisfactoriamente.
- **Rechazada**: La promesa se ha completado con un error.
- **Arreglada**: La promesa ya no está pendiente. O bien se ha cumplido, o bien se ha rechazado.

Sintaxis

```
new Promise(function(resolve, reject) { ... });
```

Cuando creamos una promise, le pasamos una función en cuyo interior deberían producirse las operaciones asíncronas, que recibe 2 argumentos:

- **Resolve**: Es la función que llamaremos si queremos resolver satisfactoriamente la promesa.
- **Reject**: Es la función que llamaremos si queremos rechazar la promesa.

Vamos a ver a grandes rasgos el contenido de una promise:

```
return new Promise(function(resolve, reject){

    //get some object that can do async tasks
    var asyncObj = new MyAsyncObj();

    //when async task ends successfully, resolve promise
    asyncObj.onSuccess = function(result){
        resolve(result);
```

```
    };
    //when async task ends with error, reject promise
    asyncObj.onError = function(error){
        reject(error);
    }

    //perform async operation
    asyncObj.doSomething();
  })
```

Añadiendo callbacks a una *promise*

En el apartado anterior hemos visto como crear nuestra propia promesa, pero igual de importante es saber **cómo decirle a una promesa qué tiene que ejecutar cuando se resuelve o rechaza**.

Para eso tenemos 2 métodos:

- **Promise.prototype.catch(onRejected)**: Añade un *callback* que se ejecutará si la *promise* es rechazada, y devuelve la *promise* actualizada.
- **Promise.prototype.then(onFulfilled, onRejected)**: Añade un *callback* para caso de éxito, y otro para caso de error y devuelve la *promise* actualizada.

Vamos a ver un ejemplo de como añadiríamos callbacks a una promise:

```
let promise = new Promise(function(resolve, reject){
//...some stuff...
});

promise.then(
  function(value){
    console.log("success");
  },
  function(reason){
    console.log("error ", reason);
  }
);

promise.then(function(success){console.log("another callback")});
```

```
//if promise is resolved successfully, output will be:
//    success
//    another callback
//otherwise, output will be:
//    error "some reason"
```

Utilizando promises

Ahora que ya hemos visto como crear promesas y vincular callbacks para su resolución, vamos a ver un ejemplo completo de cómo crearíamos una promise para realizar una petición HTTP GET:

```
//Here we create the promise
function httpGET(url){
  return new Promise(function(resolve, reject){
    //create request object
    var request = new XMLHttpRequest();

    //callback to call when readyState changes
    request.onreadystatechange = function(){
      //check status when operation is completed
      if(request.readyState == 4){
        //if GET request is resolved with code 200, resolve promise
        if(request.status === 200){
          resolve(request.response);
        }
        //otherwise, reject promise
        else{
          reject(new Error(request.statusText));
        }
      }
    };

    //callback to call when error is received: reject promise
    request.onerror = function(){
      reject(new Error(this.statusText));
    };

    //initialize as GET request and set url
    request.open('GET', url);
```

```
    //send http get request
    request.send();
  });
};


//Here, we add callbacks
httpGET('http://codepen.io/chriscoyier/pen/difoC.html').then(functi
on(value){
  console.log("success");
}, function(reason){
  console.log("error ", reason);
})
```

Conclusiones

Hace tiempo que las *promises* se han convertido en la vía más habitual de gestionar operaciones asíncronas. **ES6 refuerza esta apuesta incorporando este mecanismo en la propia librería de Javascript**, evitando dependencias con librerías externas. Si hasta ahora no utilizabas *promises*, ¡este es el momento de empezar!

Puedes mirar la documentación sobre promises para casos menos habituales de los que no he hablado, como iteradores de promesas.

Métodos de Object

Con la entrada de **ES6**, el objeto **Object** ha sufrido una actualización, incorporando nuevos métodos estáticos que lo complementan. Vamos a explorarlos.

Object.assign()

Con este método, podemos **copiar todas las propiedades enumerables *propias*** de uno o varios objetos (que denominaremos *source*) a otro objeto (que llamaremos *target*).

```
Object.assign(target, source_1, ..., source_n)
```

Veamos como funciona:

```
var targetObj = {};
var sourceObj1 = {};
var sourceObj2 = {};
target.a = 1;
sourceObj1.b = 2;
sourceObj2.c = 3;
Object.assign(targetObj, sourceObj1, sourceObj2);
console.log(targetObj.b, targetObj.c) //2, 3

//example with returned object
var copy = Object.assign({}, sourceObj1);
console.log(copy); // { b: 2 }
```

Object.is()

El método **Object.is()** determina si dos objetos son iguales.

```
Object.assign(target, source_1, ..., source_n)
```

El método **Object.is()** es similar al operador === con la diferencia qe que éste último trata los valores -0 y +0 como iguales, y además, trata Number.NaN como no igual a NaN.

Veamos unos ejemplos:

```
Object.is('foo', 'foo');     // true
Object.is(window, window);   // true

Object.is('foo', 'bar');     // false
Object.is([], []);           // false
```

```
var test = { a: 1 };
Object.is(test, test);       // true

Object.is(null, null);       // true
Object.is(undefined, undefined);       // true

// Special Cases
Object.is(0, -0);            // false
Object.is(-0, -0);           // true
Object.is(NaN, 0/0);         // true
```

Object.setPrototypeOf()

El método Object.setPrototypeOf() establece el *prototype* de un objeto especificado (lo llamaremos *prototype*) a otro objeto (lo llamaremos *target*) o sino establece null.

```
Object.setPrototypeOf(target, prototype)
```

Veamos un par de ejemplos:

```
//example 1, prototype null
var dict = Object.setPrototypeOf({}, null);

//example 2
var bicycle = {
 //ES6 way to define method
 speed() { console.log('15km/h')}
};

var mountainBike = {
  trial() { return true; }
};

Object.setPrototypeOf(mountainBike, bicycle);
mountainBike.speed(); //15km/h
```

Puedes mirar la documentación de Mozilla para casos más complejos como agregar una cadena entera de prototipos a el prototipo nuevo del objeto.

Object.getOwnPropertySymbols()

ES6 incorpora un nuevo tipo de datos llamado **Symbol**, que son únicos (en la mayoría de casos) e inmutables.

Veamos unos pocos ejemplos de de como crear símbolos:

```
var sym1 = Symbol();
var sym2 = Symbol("foo");
var sym3 = Symbol("foo");
Symbol("foo") === Symbol("foo"); // false
```

Desde mi punto de vista, el uso de símbolos en Javascript le dota de mucha potencia de cara a la **metaprogramación**, pero eso es otro tema que da para varios artículos. De momento, podemos pensar en **Symbol** como una forma de crear etiquetas realmente únicas. Me explicaré mejor con un ejemplo:

```
//ES5
log.levels = {
    DEBUG: 'debug',
    INFO: 'info'
};
log(log.levels.DEBUG, 'debug message');
log(log.levels.INFO, 'info message');
log('info', 'info message'); //this is logged like log.levels.INFO
```

```
log.levels = {
    DEBUG: Symbol('debug'),
    INFO: Symbol('info')
};
log(log.levels.DEBUG, 'debug message');
log(log.levels.INFO, 'info message');
log(Symbol('info'), 'info message'); //this log level is not
recognized
```

> Los **Symbols** no se listan con la llamada **Object.getOwnPropertyNames()**, sino que tienen su propio

> método: **Object.getOwnPropertySymbols(obj)**

Veamos como funciona **Object.getOwnPropertySymbols(obj)**:

```
var obj = {};
var a = Symbol('a');
var b = Symbol('b');

obj[a] = 'Symbol1';
obj[b] = 'Symbol2';

var objectSymbols = Object.getOwnPropertySymbols(obj);

console.log(objectSymbols.length); // 2
console.log(objectSymbols);        // [Symbol(a), Symbol(b)]
console.log(objectSymbols[0]);     // Symbol(a)
```

Abreviación de Object.prototype

No es propiamente un método, pero ya que hablamos de las novedades que incorpora ES6 a la clase Object, aprovecho para comentarlo: podemos usar **{}** como abreviación de **Object.prototype**..

> Con ES6, puedes acceder a los métodos de **Object.prototype** a través de un **objeto literal vacío**, es decir, **{}**.

Es decir, las dos lineas a continuación son equivalentes:

```
Object.prototype.hasOwnProperty.call(obj, 'propKey')
{}.hasOwnProperty.call(obj, 'propKey')
```

Métodos de String

Seguimos fusilando las novedades de ES6, y esta vez es el turno de la clase String. ECMASCript 6 añade nuevos métodos a los *strings* de Javascript. Vamos a verlos.

Métodos de objeto string

repeat

```
myString.repeat(nTimes);
```

Gracias a la sintáxis de ES6, podemos repetir una cadena de strings de forma muy simple con el método **repeat**. Solo tenemos que indicarle cuantas veces queremos que se repita el string.

```
'Hola'.repeat(2); //HolaHola
```

startsWith

```
myString.startsWith(str);
```

Antiguamente utilizábamos el método **indexOf** para saber si un string comenzaba por una cadena de texto en concreto. Ahora, podemos hacerlo directamente con el método **startsWith**. Veamos:

```
//ES5
if('hola'.indexOf('ho') === 0) {console.log("yes!")};

//ES6
```

```
'hola'.startsWith('ho');// true
```

endsWith

```
myString.endsWith(str);
```

De forma análoga a **startsWith**, con **endsWith** ya no necesitamos usar **indexOf** para saber si un string acaba con una cadena concreta de texto. Veamos:

```
//ES5
var str = 'hola';
if(str.indexOf('a') === str.length -1) {console.log("yes!")};

//ES6
str.endsWith('a');// true
```

includes

```
myString.includes(str);
```

Acabamos con los métodos que liberan a **indexOf** de responsabilidades con el método **includes**. En este caso, el método nos permite saber si una cadena de texto determinada está incluida en nuestro string.

```
//ES5
if('hola'.indexOf('la') >= 0) {console.log("yes!")};

//ES6
'hola'.includes('la');// true
```

codePointAt

Este método nos devuelve el valor numérico de un *code point* dado un índice en el string, es decir, nos devuelve su valor unicode.

Veamos:

```
const str = 'x\uD83D\uDE80y';
console.log(str.codePointAt(0).toString(16)); // 78
console.log(str.codePointAt(1).toString(16)); // 1f680
console.log(str.codePointAt(3).toString(16)); // 79
```

Métodos estáticos

Literales *raw*

```
String.raw`template literal`;
```

Ya hemos hablado en el pasado de los *template literals* que nos permiten interpolar *strings* así como definir *strings* multilinea.

Pues bien, la clase **String** proporciona un método que nos permite crear *template literals* **crudos**, es decir, el **texto original NO INTERPRETADO**. Lo entenderás antes con un ejemplo:

```
var regularStr = `Hi\n${2+3}!`;
console.log(regularStr);
//"Hi
//5!"

var rawStr = String.raw`Hi\n${2+3}!`;
console.log(rawStr);
//"Hi\n5!"
```

fromCodePoint

De modo inverso al método de string *codePointAt*, String proporciona un método estático que nos devuelve el símbolo unicode a partir de su *code point*, es decir:

```
String.fromCodePoint(78); // 'x'
String.fromCodePoint(0x1f680); // 'uD83D'
String.fromCodePoint(78, 90);   // "xZ"
```

Conclusiones

Estas son las principales novedades que incorpora ES6 a la hora de trabajar con Strings y me aventuro a decir que algunas de ellas -especialmente las de búsqueda de cadenas en string- nos van a simplificar bastante la vida. Tiempo al tiempo…

Métodos de Array

Llega el momento de comprobar las novedades que aporta ECMAScript 2015 a los Arrays de Javascript.

Métodos de objeto array

Array.prototype.findIndex

```
var position = myArray.findIndex(x => x_condition)
```

ECMAScript 2015 incorpora el método **findIndex** a los arrays, mediante el cual nos devuelve el índice del array donde se encuentra el primer elemento que cumple la condición que le hemos indicado.

La función **findIndex** es parecida a **indexOf**, con un par de diferencias:

- Le pasamos como argumento una *arrow function*. Es la condición que tiene que cumplir el elemento del array que queremos detectar.
- Permite **detectar NaN**

Veamos un par de ejemplos:

```
const arr = ['a', NaN];
console.log(arr.findIndex(x => Number.isNaN(x))); // 1

[3, 1, -1, -5].findIndex(x => x < 0);//2
```

Array.prototype.find

```
var element = myArray.find(x => x_condition)
```

El método **find** es muy **similar** al método **findIndex, pero** en este caso, lo que nos **devuelve** es directamente **el valor** del elemento que cumple la condición que hemos definido.

Tomando como referencia el ejemplo de código anterior, vemos que:

```
const arr = ['a', NaN];
console.log(arr.find(x => Number.isNaN(x))); //NaN

[3, 1, -1, -5].find(x => x < 0);//-1
```

Array.prototype.fill

```
const myArray = new Array(size).fill(value, start?, end?);
```

Con el método **fill**, por fin tenemos una forma elegante de crear un Array de un tamaño determinado e inicializar su contenido. Veamos unos ejemplos:

```
const array = new Array(3).fill('a'); //['a', 'a', 'a']
['a', 'b', 'c', 'd'].fill(null, 2, 3); //['a', 'b', null, 'd']
```

Array.prototype.copyWithin

```
myArray.copyWithin(target, start, end = this.length)
```

El método ***copyWithin*** copia los elementos comprendidos **entre los indices [start,end)** a partir de la posición **target**.

Veamos:

```
const arr = [0,1,"x","y",4,5,6];
arr.copyWithin(3, 2, 4);
console.log(arr);
//[0, 1, "x", "x", "y", 5, 6]
```

Iterando sobre arrays

Igual que los *Maps* y los *Sets*, los **Arrays disponen de los métodos *keys()*, *values()*, y *entries()*** que nos facilitan su uso como **objetos iterables**.

Vamos a ver algunos ejemplos de como usarlos:

```
let array = ['x', 'y', 'z'];


for(let key of array.keys()){
  console.log(key);
}
//0
//1
//2
```

```
for(let [key, val] of array.entries()){
  console.log(val, key);
}
//"x" 0
//"y" 1
//"z" 2


var values = array.values();
while(true){
  let item = values.next();
  if(item.done)
    break;
  console.log(item.value);
}
//"x"
//"y"
//"z"
```

Abreviación de Array.prototype

De forma similar a como sucede con Object, ES6 nos permite acceder al *prototype* de Array a través de un Array vacío literal, es decir, **[]**. Vamos a verlos:

```
//ES5
Array.prototype.slice.call(arguments)

//ES6
[].slice.call(arguments)
```

Métodos estáticos

Array.from

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

Array.from permite convertir los 2 siguientes tipos de valores en arrays:

- valores ***array-like*** (tienen logintud y elementos indexados)
- valores **iterables**

Entendamos mejor el primer caso. ¿Por qué nos podría servir convertir *pseudoarrays* a objetos *Array* de verdad?

```
const arrayLike = { length: 2, 0: 'x', 1: 'y' };

// no podemos usar for-of, por que no es un iterable
for (const item of arrayLike) {
    console.log(item);
}// TypeError

const array = Array.from(arrayLike);
//ahora sí podemos, array es iterable
for (const item of array) {
    console.log(item);
}
// a
// b
```

De la segunda opción, objetos iterables, vemos que podemos crear *Arrays* a partir de Maps, Sets, strings, iterables obtenidos a partir de array.keys(), etc.

Veamos algunos ejemplos de **Array.from** con iterables:

```
Array.from(['a', 'b', 'c']);
//['a', 'b', 'c']

Array.from(['a', 'b', 'c'].keys());
//[0, 1, 2]

Array.from("foo");
// ["f", "o", "o"]

var set = new Set(["foo", "bar", 1]);
Array.from(set);
// ["foo", "bar", 1]
```

```
var map = new Map([['bar', 1], ['foo', 2]]);
Array.from(map);
//[["bar", 1], ["foo", 2]]
```

> Array.from es además una forma genérica de utilizar la función **map()**.

Veamos como usarlo de forma equivalente a *map()*:

```
//ES5
var mappedArray = Array.prototype.map.call([1, 2, 3], x => x*x);
// [1, 4, 9]

//ES6
var mappedArray = Array.from([1, 2, 3], x => x*x);
// [1, 4, 9]
```

Array.of

```
Array.of(...items)
```

El método **Array.of** crea un array con los elementos que se le pasan como argumento. Veamos como funciona:

```
let array = Array.of(1,3,5);
//[1,3,5]
```

Agujeros en Arrays

Los agujeros en arrays (**holes**) son elementos que no existen dentro de un array. Por

ejemplo:

```
const arr = ['a',,'b']
'use strict'
0 in arr; //true
1 in arr; //false
2 in arr; //true
arr[1]; //undefined
```

Algunos métodos de Array que se definieron en ES5 ignoran los agujeros en arrays, mientras que otros los eliminan, y otros los consideran elementos **undefined**.

Los métodos que añade **ES6** (los que hemos estado viendo en este artículo), tratan siempre los agujeros de array como elementos undefined.

> Los métodos que incorpora ES6 tratan los **array holes** como si fueran elementos **undefined**.

Métodos de Number

Acabando ya con el repaso a las novedades de ES6, vamos a explorar los nuevos métodos estáticos que aporta ECMAScript 2015 a la clase Number.

Nuevos literales para enteros

A partir de ahora, se pueden definir *integers* a partir de literales tanto en binario como en octal:

```
// ES5 - hexadecimal
0xFF; //255

//ES6 - binary
```

```
0b101; //5

// ES6 - octal
0o101 ; //65
```

Métodos estáticos de Number

Number.isFinite()

Nos permite saber si un numero es finito o no.

```
Number.isFinite(Infinity); //false
Number.isFinite(NaN); //false
Number.isFinite(5); //true
```

Number.isInteger()

Nos permite saber si un número es entero o no.

```
Number.isInteger(-1); //true
Number.isInteger(1); //true
Number.isInteger(1.0); //true
Number.isInteger(1.1); //false
Number.isInteger(Infinity); //false
Number.isInteger(NaN); //false
Number.isInteger('1'); //false
```

Number.isSafeInteger()

Nos permite saber si un entero es *seguro*, es decir, pertenece a un rango de 53 bits por lo que no hay pérdida de precisión.

> Debido a que Javascript utiliza un formato numérico de coma flotante de doble precisión, Un ***safe integer*** en Javascript queda comprendido entre **Number.MIN_SAFE_INTEGER** (de valor $-2^{53}+1$) y **Number.MAX_SAFE_INTEGER** (de valor $2^{53}-1$). Los valores fuera de ese umbral, se truncan a dicho límite.

Así, con el método **isSafeInteger()** podemos comprobar de forma directa si el entero se encuentra dentro del rango seguro.

```
Number.isSafeInteger(1); //true
Number.isSafeInteger(1.1); //false
Number.isSafeInteger(Math.pow(2, 53)-1); //true
Number.isSafeInteger(Math.pow(2, 53)); //false
```

Number.isNaN()

Indica si el número tiene valor **NaN** (*Not A Number*), y es ligeramente distinto a la antigua función global **isNaN()**

```
Number.isNaN(NaN);        // true
Number.isNaN(Number.NaN); // true
Number.isNaN(0 / 0)       // true

// e.g. these would have been true with global isNaN()
Number.isNaN("NaN");      // false
Number.isNaN(undefined);  // false
Number.isNaN({});         // false
Number.isNaN("blabla");   // false

// These all return false
Number.isNaN(true);
Number.isNaN(null);
Number.isNaN(1);
Number.isNaN("1.1");
Number.isNaN(" ");
```

Number.parseFloat()

Es equivalente a la antigua función global **parseFloat()**.

```
var floatNum = Number.parseFloat(string);
```

Number.parseInt()

Es equivalente a la antigua función global **parseInt()**.

```
var int = Number.parseInt(string[, radix]);
```

Hasta aquí las novedades de la clase Number, pequeñas mejoras que seguramente simplificaran un poco nuestro código.

Métodos de Math

Completamos el repaso a las novedades de **ES6** con los nuevos métodos estáticos de la clase **Math**.

Math.sign(x)

Esta función nos devuelve 1 si el signo del parámetro es positivo, -1 si es negativo, y +/-0 para valores cero. Veamos un ejemplo:

```
Math.sign(5);           //  1
Math.sign('5');         // 1
Math.sign(-5);          // -1
```

```
Math.sign(0);           //  0
Math.sign(-0);          // -0
Math.sign(NaN);         //NaN
Math.sign(-Infinity)    //-1
```

Math.trunc(x)

Este método elimina la parte decimal del parámetro. Complementa a los métodos **Math.ceil()**, **Math.floor()** y **Math.round()**, con la principal diferencia de que en este caso nos devuelve la parte entera sin redondear de ningún modo.

```
Math.trunc(5.1);        //5
Math.trunc(5.9);        //5
Manth.trunc('-0.12');   //-0
Manth.trunc(-0.9);      //-0
```

Math.cbrt(x)

El nombre del método **Math.cbrt(x)** viene de *cube root*, y como es de imaginar, devuelve la **raíz cúbica** de x , es decir: $\sqrt[3]{x}$

```
Math.cbrt(27); //3
```

Métodos relacionados con exponentes y logaritmos

Debido a que los números de coma flotante en base 10, se representan internamente como *mantisa x* E^{exp}, resulta que las fracciones pequeñas se representan de forma más precisa si son un valor entre 0 y 1.

Para aclarar un poco esta idea, veamos el siguiente ejemplo:

$(1) 0.00012 = 1.2 \times 10^{-4}$ - Dígitos significativos: 12

$(2) 1.00012 = 1.00012 \times 10^{0}$ - Dígitos significativos: 100012

Como se puede apreciar, el primer ejemplo necesita almacenar menos dígitos significativos, por lo que nos ofrecerá una mayor resolución frente al segundo.

Aprovechando esta posibilidad, se han creado 2 métodos concretos: **expm1** y **log1p**.

Math.expm1(x)

Este método nos da mayor precisión con los decimales cuando el resultado de **Math.exp()** es muy cercano a 1.

A efectos prácticos, devuelve **Math.exp(x) -1**, es decir $e^x - 1$

Podemos ver la diferencia con el siguiente ejemplo:

```
//partimos de 1e-10, es decir 0.0000000001
//Math.exp(1e-10) será 1.0000000001
//pero veamos como se ve afectada la precisión de estos métodos:

//ES5
Math.exp(1e-10)-1; //1.000000082740371e-10
//ES6
Math.expm1(1e-10); //1.00000000005e-10
```

Como podemos ver del ejemplo, el nuevo método ofrece una mayor precisión.

Math.log1p(x)

Éste método devuelve la inversa de **Math.expm1()**, y del mismo módo, permite especificar parámetros cercanos a 1 con una mayor precisión.

A efectos prácticos, devuelve *Math.log(1 + x)*, es decir: $log(1 + x)$

Veamos un ejemplo:

```
//ES5
Math.log(1 + 1e-16); //0

//ES6
Math.log1p(1e-16); //1e-16, es decir 0.0000000000000001
```

Como vemos el primer caso no es tan preciso y devuelve directamente cero.

Math.log2(x)

Este método devuelve el logaritmo base 2 del argumento, es decir $log_2(x)$.

```
Math.log2(8); //3
```

Math.log10(x)

Este método devuelve el logaritmo base 10 del argumento, es decir $log_{10}(x)$.

```
Math.log10(10000); //4
```

Math.clz32(x)

Este método cuenta los bits cero por la izquierda de un entero de 32 bits. Lo entenderás mejor con un ejemplo:

```
//un entero 3, está representado en binario por '11'.
//esto, en un integer será 00...011, con 30 bits a 0 y luego dos a
1. Por tanto:
Math.clz32(3); //30

//otros ejemplos:
Math.clz32(2); //30
Math.clz32(1); //31
```

```
Math.clz32(0b01000000000000000000000000000000); //1
Math.clz32(0b00001000000000000000000000000000); //4
```

Operaciones trigonométricas

Además de los métodos anteriores, **ES6** incluye en la clase **Math** una serie de operaciones trigonométricas comunes:

Math.sinh(x)

Devuelve el **seno hiperbólico de x**

Math.cosh(x)

Devuelve el **coseno hiperbólico de x**

Math.tanh(x)

Devuelve la **tangente hiperbólica de x**

Math.asinh(x)

Devuelve la **inversa del seno hiperbólico de x**

Math.acosh(x)

Devuelve la **inversa del coseno hiperbólico de x**.

Math.atanh(x)

Devuelve la **inversa de la tangente hiperbólica de x**.

Math.hypot(...values)

El nombre se debe al teorema de Pitágoras ya que este método nos devuelve la **hipotenusa** dados los catetos. Observar, sin embargo, que le podemos pasar tantos valores como queramos. La fórmula que aplica es la siguiente :

$$\sqrt[2]{arg_1{}^2 + arg_2{}^2 + \ldots + arg_n{}^2}$$

Veamos unos ejemplos de este último:

```
Math.hypot(3, 4); //5
Math.hypot(3,4,12); //9
```

Conclusiones

ES6 potencia la librería **Math** con más funciones trigonométricas y otras operaciones básicas como obtener el *signo* o *truncar* números. Obviamente no alcanza todo el potencial de librerías matemáticas dedicadas como puedan ser **Math.js** o **numbers.js**, pero puede que nos evite tener que sobrecargar nuestra página con una librería extra, o crear uno de estos métodos tan primarios a mano.

Conclusiones finales

Estas **son las principales novedades** de ECMAScript 2015, y como puedes comprobar, **ES6** supone una **revolución** en la forma de utilizar **Javascript** y lo dota de varios mecanismos que llevan siendo reclamados por la comunidad desde hace tiempo.

Podríamos decir que nos encontramos ante una **versión más madura y potente** de Javascript, y ha llegado para quedarse, así que **NO lo dudes y empieza a utilizarlo en tu día a día**, lo agradecerás.