
JavaScript


ES6

UD 08 ES06

Introducción

En 1995, Netscape quería implementar su propia máquina virtual de Java para el navegador, usando un lenguaje más amigable, orientado a programadores principiantes. Este lenguaje era muy permisivo, por ejemplo, el punto y coma que se pone al final de una instrucción era opcional, las variables no eran tipadas, etc. El nombre de este proyecto se conoció como **Mocha** (mencionar que Java también es una forma de referirse al café por su cultivo en dicha isla).

En 1995, Microsoft se quejó porque el API de JavaScript cambiaba frecuentemente y querían implementar su propia versión compatible. Netscape pensó que Microsoft quería usar JavaScript para crear su propia versión incompatible (de hecho, lo intentaron) y convertirse en el nuevo estándar. Con el fin de protegerlo, Netscape creó un estándar independiente llamado **ECMAScript**.

Para entender la diferencia, ECMAScript es la especificación (características) que el lenguaje debería tener, y JavaScript es la implementación de esa especificación (otra implementación, por ejemplo, sería ActionScript de Adobe). Este estándar fue adoptado por muchas compañías incluyendo IBM, Microsoft, Netscape, Sun, etc. Funcionó bien hasta la versión 3.

Desde 1999 hasta 2006, no hubo nuevas versiones de ECMAScript principalmente porque Microsoft llego a tener el monopolio con Internet Explorer y no modificó nada en el estándar. Afortunadamente, cuando Netscape murió, se convirtió en un proyecto de código abierto y Mozilla Firefox nació de él (otros navegadores como Opera ya existían pero su cuota de mercado era baja). En 2006, hubo discusiones sobre si realizar ligeras modificaciones en el estándar y lanzar una versión ES3.1 (Microsoft, Yahoo, etc.) o hacer cambios mayores y sacar la versión ES4 (Mozilla, Opera, Adobe).

Al final salió la versión ES3.1 y la versión ES4 fue pospuesta. Por motivos de marketing ES3.1 fue renombrado como ES5 (ES4 nunca existió por tanto), es por tanto, la versión que usa actualmente, y la versión ES6 tendría más o menos lo que se incluiría en la versión original ES4. ES5 fue la versión lanzada en el 2009 y ES5.1 en el 2011. De 1999 hasta el 2015 se puede decir que JavaScript ha estado bastante estancado como lenguaje.

ES6 fue lanzada en Junio del 2015 y renombrada a **ES2015** (cuestiones de marketing). Desde entonces, las nuevas versiones se lanzan anualmente con el número de año (añadiendo pequeñas características poco a poco). En junio del 2016 se lanzó ES2016 (pocos cambios). La versión en desarrollo de ECMAScript se conoce como **ES.Next.**

Hoy en día la mayoría de los navegadores implementan el estándar ES2015 (no el 100% del mismo), por tanto, si queremos hacer una aplicación web compatible con todos los navegadores en un rango de varios años, necesitamos tener una herramienta llamada **transpilador** que nos permite convertir nuestro código en ES2015 a ES5. Podemos ver la compatibilidad de los navegadores con el estándar ES aquí: http://kangax.github.io/compat-table/es6/

Nueva sintaxis en ES2015

let y const

let pasa a ser la forma recomendada para declarar variables. La principal diferencia con var es que evita **hoisting**. Si se accede a una variable antes de ser declarada, no devolverá undefined (la declaración de la variable se movía al principio por el intérprete), en lugar de eso, nos mostrará un error (comportamiento lógico).

```
'use strict';
console.log(number); // Imprime undefined (hoisting)
var number = 14;
'use strict';
console.log(number);//→Uncaught ReferenceError: number is not defined(lo correcto)
let number = 14;
```

Otra ventaja de definir una variable con let es que pasa a ser local en cualquier bloque. Por ejemplo, una variable que ha sido declarada con let dentro de un bloque if, while, etc. sólo existe dentro de ese bloque.

```
'use strict';
for (var i = 0; i < 10; i++) {
   console.log(i);
}
console.log(i); // Imprime 10 (si existe fuera del bucle)
'use strict';
for (let i = 0; i < 10; i++) {
   console.log(i);
}
console.log(i); //→ Uncaught ReferenceError: i is not defined
```

Podemos crear un bloque simple para cambiar el ámbito de una variable.

```
'use strict';
let number = 10; {
   let number = 200; // aquí number es local a este bloque
}
console.log(number); // Imprime 10 (valor de number fuera del bloque anterior)
```

Vamos a ver otro ejemplo que normalmente es problemático cuando creamos una función dentro de un bucle y usamos un contador dentro de estas funciones:

```
'use strict';
let functions = [];
for (var i = 0; i < 10; i++) {
   functions.push(function () {
      console.log(i);
   });
}
functions[0](); // Imprime 10, i valor actual (no es local en cada iteración)
```

```
functions[1](); // Sigue imprimiendo 10
```

Funciones lambda (o arrow functions)

Una de las funcionalidades más importantes que se añadió fue la posibilidad de usar las funciones o expresiones lambda (o flecha). Otros lenguajes como C#, Java, etc. también las soportan. Estas expresiones ofrecen la posibilidad de crear funciones anónimas pero con algunas ventajas.

Vamos a ver las diferencias que tiene por ejemplo entre dos funciones equivalentes (una anónima y otra lambda que hacen lo mismo):

```
let sum = function (num1, num2) {
   return num1 + num2;
}
console.log(sum(12, 5)); // Imprime 17
let sum = (num1, num2) => num1 + num2;
console.log(sum(12, 5)); // Imprime 17
```

Cuando declaramos una función lambda, la palabra reservada **function** no se usa. Si sólo se recibe un parámetro, los paréntesis de la función pueden ser omitidos. Después de los paréntesis debe ir una flecha (=>), y el contenido de la función.

```
let square = num => num * num;
console.log(square(3)); // Imprime 9
```

Si sólo hay una instrucción dentro de la función lambda, podemos omitir las llaves '{}', y debemos omitir la palabra reservada **return** ya que lo hace de forma implícita (devuelve el resultado de esa instrucción). Si hay más de una instrucción, usamos las llaves y se comporta la función como una función normal y por tanto, si devuelve algo, debemos usar la palabra reservada return.

```
let sumInterest = (price, percentage) => {
   let interest = price * percentage / 100;
   return price + interest;
}
console.log(sumInterest(200, 15)); // Imprime 230
```

La diferencia más importante entre ambos tipos de funciones es el comportamiento de la palabra reservada **this**. Si usamos una función anónima, normalmente tendrá su propio prototipo (excepto para los métodos de una clase), y si la función anónima está dentro de un constructor o un método, **this** no apuntará al objeto principal, sino a la función anónima... Cuando usamos una lambda, no se crea un prototipo, por tanto la palabra reservada **this** todavía apunta al objeto externo.

```
function Product(price) {
   this.price = price;
   setTimeout(function () {
      console.log(this.price); // Imprime undefined
   }, 1000);
```

```
}
let p = new Product(200);

function Product(price) {
   this.price = price;
   setTimeout(() => {
      console.log(this.price); // Imprime 200
   }, 1000);
}
let p = new Product(200);
```

Esto pasa mucho con las llamadas AJAX, donde usamos funciones anónimas para definir la acción que realizará cuando reciba la respuesta. Una solución que se ha utilizado de forma clásica con estas funciones anónimas es declarar una variable (por ejemplo llamada **self**) y asignarle este objeto a la variable.

```
'use strict';

function Product(price) {
   this.price = price;
   let self = this;
   setTimeout(function () {
      console.log(self.price); // Imprime 200
   }, 1000);
}
let p = new Product(200);
```

Parámetros por defecto

Si un parámetro se declara en una función y no se pasa cuando la llamamos, se establece su valor como **undefined**.

```
function Persona(nombre) {
   this.nombre = nombre;
   this.diHola = function () {
      console.log("Hola! Soy " + this.nombre);
   }
}
let p = new Persona();
p.diHola(); // Imprime "Hola! Soy undefined"
```

Una solución usada para establecer un valor por defecto era usar el operador ‘ ||’ (or), de forma que si se evalúa como undefined (false), se le asigna otro valor.

```
function Persona(nombre) {
   this.nombre = nombre || "Anónimo";
   ...
}
```

Sin embargo, en **ES2015** tenemos la opción de establecerle un valor por defecto directamente.

```
function Persona(nombre = "Anónimo") {
   this.nombre = nombre;
   ...
}
```

Podemos asignarle un valor basado en una expresión (y podemos usar otros parámetros en dicha expresión).

```
function getPrecioTotal(precio, impuesto = precio * 0.07) {
   return precio + impuesto;
}
console.log(getPrecioTotal(100)); // Imprime 107
```

Rest y spread

Rest es la acción de transformar un grupo de parámetros en un array, y **spread** es justo lo opuesto, extraer los elementos de un array (o de un string) a variables.

Para usar **rest** en los parámetros de una función, se declara siempre como último parámetro (**1 máximo)** y se le ponen tres puntos‘...’ delante del mismo. Este parámetro se transformará automáticamente en un array conteniendo todos los parámetros restantes que se le pasan a la función. Si por ejemplo, el parámetro **rest** está en la tercera posición, contendrá todos los parámetros que se le pasen a excepción del primero y del segundo (a partir del tercero).

```
function getMedia(...notas) {
   console.log(notas); // Imprime [5, 7, 8.5, 6.75, 9] (está en un array)
   let total = notas.reduce((total, notas) => total + notas, 0);
   return total / notas.length;
}
console.log(getMedia(5, 7, 8.5, 6.75, 9)); // Imprime 7.25
function imprimirUsuario(nombre, ...lenguajes) {
   console.log(nombre + " sabe " + lenguajes.length + " lenguajes: " + lenguajes.join(" - ")
);
}
// Imprime "Pedro sabe 3 lenguajes: Java - C# - Python"
printUserInfo("Pedro", "Java", "C#", "Python");
// Imprime "María sabe 5 lenguajes: JavaScript - Angular - PHP - HTML - CSS"
printUserInfo("María", "JavaScript", "Angular", "PHP", "HTML", "CSS");
```

Spread es lo “opuesto” de rest. Si tenemos una variable que contiene un array, y ponemos los tres puntos ‘...’ delante de este, extraerá todos sus valores. Podemos usar la propiedad por ejemplo con el método **Math.max,** el cual recibe un número indeterminado de parámetros y devuelve el mayor de todos.

```
let nums = [12, 32, 6, 8, 23];
console.log(Math.max(nums)); // Imprime NaN (array no es válido)
console.log(Math.max(...nums)); // Imprime 32 -> equivalente a Math.max(12, 32, 6 ,8 ,23)
```

Podemos usar también esta propiedad si necesitamos clonar un array (sin iterar sobre él).

```
let a = [1, 2, 3, 4];
let b = a; // Referencia el mismo array que 'a' (las modificaciones afectan a ambos).
let c = [...a]; // Nuevo array -> contiene [1, 2, 3, 4]
```

Desestructuración

Desestructuración es la acción de extraer elementos individuales de un array (o propiedades de un objeto) directamente en variables individuales. Podemos también desestructurar un string en caracteres.

Vamos a ver un ejemplo donde asignamos los tres primeros elementos de un array, en tres variables diferentes, usando una única asignación.

```
let array = [150, 400, 780, 1500, 200];
let [first, second, third] = array; // Asigna los tres primeros elementos del array
console.log(third); // Imprime 780
```

¿Qué pasa si queremos saltarnos algún valor? Se deja vacío (sin nombre) dentro de los corchetes y no será asignado:

```
let array = [150, 400, 780, 1500, 200];
let [first, , third] = array; // Asigna el primer y tercer elemento
console.log(third); // Imprime 780
```

Podemos asignar el resto del array a la última variable que pongamos entre corchetes usando **rest** (como en el punto anterior del tema):

```
let array = [150, 400, 780, 1500, 200];
let [first, second, ...rest] = array; // rest -> array
console.log(rest); // Imprime [780, 1500 ,200]
```

Si queremos asignar más valores de los que contiene el array y no queremos obtener undefined, podemos usar valores por defecto:

```
let array = ["Peter", "John"];
let [first, second = "Mary", third = "Ann"] = array; // rest -> array
console.log(second); // Imprime "John"
console.log(third); // Imprime "Ann" -> valor por defecto
```

También podemos desestructurar **arrays anidados:**

```
let sueldos = [["Pedro", "Maria"], [24000, 35400]];
let [[nombre1, nombre2], [sueldo1, sueldo2]] = sueldos;
console.log(nombre1 + " gana " + sueldo1 + "€"); // Imprime "Pedro gana 24000€"
```

También se puede desestructurar un array enviado como parámetro a una función en valores individuales:

```
function imprimirUsuario([id, nombre, email], otraInfo = "Nada") {
   console.log("ID: " + id);
   console.log("Nombre: " + nombre);
   console.log("Email: " + email);
   console.log("Otra info: " + otraInfo);
}
   let infoUsu = [3, "Pedro", "peter@gmail.com"];
   imprimirUsuario(infoUsu, "No es muy listo");
```

Desestructurando objetos

También es posible desestructurar propiedades de objetos. El funcionamiento es similar a desestructurar un array, pero usamos llaves '{}' en lugar de corchetes.

```
let usuario = {
   id: 3,
   nombre: "Pedro",
   email: "peter@gmail.com"
}
let {id, nombre, email} = usuario;
console.log(nombre); // Imprime "Pedro"
// En lugar de un array, esta función recibirá un objeto como primer parámetro
function imprimirUsuario({id,nombre,email}, otraInfo = "Nada") {
   ...
}
otraInfo(usuario, "No es muy listo");
```

Existe la posibilidad de asignar diferentes nombres desde las propiedades en las variables desestructuradas → (**nombrePropiedad: nombreVar**):

```
let {id: idUsuario, nombre: nombreUsuario, email: emailUsuario} = usuario;
console.log(nombreUsuario); // Imprime "Pedro"
```

Desestructurando strings

Por último, podemos tratar un string como un array de caracteres al desestructurar. La primera letra será asignada a la primera variable y así sucesivamente. Veamos un ejemplo:

```
let str = "abcdefg";
let [first, second] = str;
console.log(second); // Imprime "b"
```

Extensiones a los objetos JSON

Nueva sintaxis de los métodos

Hasta ahora, cuando definíamos un método en un objeto JSON utilizábamos la sintaxis **propiedad: function() {...}**. Ahora podemos crear un método con una sintaxis más simple:

```
let usuario = {
   id: 3,
   nombre: "Pedro",
   email: "peter@gmail.com",
   saludar() {
      console.log("Hola, mi nombre es " + this.nombre);
   }
}
usuario.saludar();
```

Se pueden crear propiedades en un objeto con el mismo nombre y valor que variables existentes (usando el nombre de esas variables sin poner valor).

```
let id = 1,
   descripcion = "Buen producto!",
   precio = 3.95;
let producto= {
   id,
   descripcion,
   precio,
   precioTotal(cantidad) {
      return this.precio * cantidad;
   }
}
console.log(producto); // Objeto {id: 1, descripcion: "Buen producto!", precio: 3.95}
console.log(producto.precioTotal(5)); // Imprime 19.75
```

También, podemos crear propiedades usando como nombre un string (valor de otra variable, etc.), usando los corchetes:

```
let prop1 = "id";
let prop2 = "precio";
let meth1 = "getTotal";
let prod = {
   [prop1]: 1,
   [prop2]: 5.75,
   [prop2 + "Rebaja"]: 4.25,
   [meth1](cantidad, rebaja = false) {
      return (rebaja ? this[prop2 + "Rebaja"] : this[prop2]) * cantidad;
   }
}
console.log(prod); // Objeto {id: 1, precio: 5.75, precioRebajar: 4.25}
console.log(prod.getTotal(10)); // Imprime 57.5
```

```
console.log(prod.getTotal(10, true)); // Imprime 42.5
```

Bucle for...of

Podemos iterar a través de los elementos de un array o incluso a través de los caracteres de un string sin usar un índice. El bucle for...of (al contrario que el bucle for...in) es un tipo de bucle al estido de for each en otros lenguajes.

```
var a = ["Item1", "Item2", "Item3", "Item4"];
for (let index in a) {
   console.log(a[index]);
}
for (let item of a) { // Hace lo mismo arriba
   console.log(item);
}
var str = "abcdefg";
for (let letter of str) {
   if (letter.match(/^[aeiou]$/)) {
      console.log(letter + " es una vocal");
   } else {
      console.log(letter + " es una consonante");
   }
}
```

literales binarios y octales

Vamos a ver cómo podemos usar valores binarios y octales poniéndole el prefijo 0o (número cero y la letra o) para los octales y 0b para los binarios (internamente son almacenados como números).

```
console.log(0o10); // Imprime 8
console.log(0b10); // Imprime 2
```

literales Template

Ahora JavaScript soporta string **multilínea** con **sustitución de variables.**
Ponemos el string entre caracteres ` (backquote) en lugar de entre comillas simples o dobles. Las variables van dentro de **${}** si se quieren sustituir por su valor.

```
let num = 13;
console.log(`Example of multi-line string
the value of num is ${num}`);
```

```
Example of multi-line string
the value of num is 13
```

También se pueden utilizar expresiones:

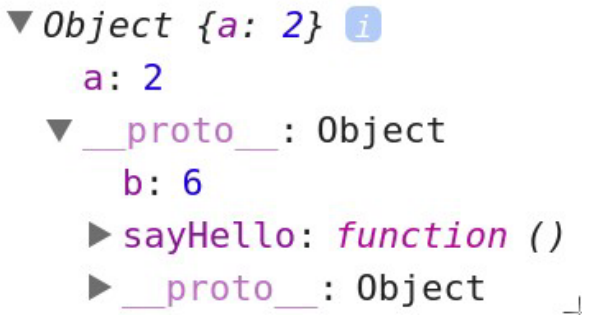
```
var a = 5;
var b = 10;
console.log(`Quince es ${a + b} y\nno ${2 * a + b}.`);
// "Quince es 15 y
// no 20
```

Extensiones de clases

Object

Object.setPrototypeOf(objA, objB) → **objB** ahora es el prototipo de objA (objA hereda todas las propiedades y métodos de objB).

```
let objA = {
  a: 2
}
let objB = {
  b: 6,
  sayHello() {
    console.log("Hello from B");
  }
}
Object.setPrototypeOf(objA, objB);
console.log(objA); // El prototipo objA es ahora objB
objA.sayHello(); // Imprime "Hello from B"
```



Object.assign(target, obj1, obj2) → Copia las propiedades de obj1 y obj2 en target. Si un nombre de una propiedad está en ambos objetos (obj1 y obj2) la propiedad de obj2 la sobreescribe.

```
let obj1 = {
  a: 2,
  b: 1
}
let obj2 = {
  b: 6,
  c: 4
}
let obj3 = {};
Object.assign(obj3, obj1, obj2);
console.log(obj3); // Objeto {a: 2, b: 6, c: 4}
```

Este método es bastante útil cuando recibimos un objeto en una función parametrizada y si alguna propiedad que necesitamos dentro de ese objeto no está definida, se crea un objeto "predeterminado" con sus propiedades predeterminadas en lugar de coger las propiedades que hay.

```
function getConfiguration(timeout, options = {}) {
  let defaults = {
    container: '#products',
    classItem: 'product',
    title: 'Example',
    description: 'Example description'
  };
  // Propiedades definidas en las opciones sobreescribirán las de por defecto
  let settings = Object.assign({}, defaults, options);
  ...}
```

String

startsWith(substring) → Comprueba si el string actual comienza con el substring pasado por parámetro.

```
console.log(str.startsWith("This")); // Imprime true
```

endsWith(substring) → Comprueba si el string actual acaba con el substring pasado por parámetro.

```
console.log(str.endsWith("string")); // Imprime true
```

includes(substring) → Comprueba si el string actual contiene el substring pasado por parámetro.

```
console.log(str.includes("ex")); // Imprime true
```

repeat(times) → Devuelve el string repetido un número determinado de veces

```
console.log("la".repeat(6)); // Imprime "lalalalalala"
```

También, ahora los string soportan caracteres con más de dos bytes (4 caracteres hexadecimales), a veces llamados caracteres de plano astral en unicode (astral plane), se deben escribir entre llaves \ u{}. Ejemplo:

```
let uString = "Unicode astral plane: \u{1f3c4}";
console.log(uString); // Imprime "Unicode astral plane: 🏄 " (icono del surfista)
```

Estos caracteres especiales devuelven un valor de dos caracteres cuando se mide la longitud de un string:

```
let surfer = "\u{1f3c4}"; // Un carácter: 🏄
console.log(surfer.length); // Imprime 2
```

Sin embargo, si transformamos un string en un array (de caracteres), este será dividido correctamente, cada carácter en una posición del array:

```
let surfer2 = "\u{1f30a}\u{1f3c4}\u{1f40b}"; // TRES caracteres: 🌊🏄🐋
console.log(surfer2.length); // Imprime 6
console.log(Array.from(surfer2).length); // Imprime 3 (convertido en array correctamente)
```

Number

En ES2015, algunas funciones globales (que operan con números) como parseInt, se han incluido dentro del objeto global Number. Veamos cómo usarlas:

- **Number.parseInt(n)**
- **Number.parseFloat(n)**
- **Number.isNaN(n)**
- **Number.isFinite(n)**
- **Number.isInteger(n)**

Math

El objeto Math ha sido ampliado con muchos más métodos:
• **Métodos hiperbólicos** → cosh(), sinh(), acosh(), asinh(), tanh(), atanh(), hypot().
• **Métodos aritméticos** → cbrt(): raíz al cubo ($\sqrt[3]{n}$), expm1(): igual a una expresión (x) – 1, log2(): logaritmo en base 2, log10(): logaritmo en base 10, log1p(): igual a logaritmo (x+1),...
• **Otros → sign()**: el signo de un número (0, -0, 1, -1, NaN), trunc(): quita los decimales de la parte entera (no redondea), etc.

Colecciones

Extensiones de Array

Vamos a ver los nuevos métodos que tiene el objeto global Array:

- **Array.of(value)** → Si queremos instanciar un array con un sólo valor, y éste es un número, usando el constructor new Array() no podemos hacerlo. Esto se debe a que un sólo número significa que se crea un array vacío con ese número de posiciones. Con Array.of(num),se crea un array con una posición que contiene ese número (u otro valor).

```
let array = new Array(10); // Array vacío (longitud 10)
let array = Array(10); // Mismo que arriba: array vacío ( longitud 10)
let array = Array.of(10); // Array con longitud 1 -> [10]
let array = [10]; // // Array con longitud 1 -> [10]
```

- **Array.from(array, func)** → Funciona de forma similar al método map, crea un array desde otro array. Se aplica una operación de transformación (función lambda o anónima) para cada ítem.

```
let array = [4, 5, 12, 21, 33];
let array2 = Array.from(array, n => n * 2);
console.log(array2); // [8, 10, 24, 42, 66]
let array3 = array.map(n => n * 2); // Igual que Array.from
console.log(array3); // [8, 10, 24, 42, 66]
```

- **Array.fill(value)** → Este método sobrescribe todas las posiciones de un array con un nuevo valor. Es una buena opción para inicializar un array que se ha creado con N posiciones.

```
let sums = new Array(6); // Array con 6 posiciones
sums.fill(0); // Todas las posiciones se inicializan a 0
console.log(sums); // [0, 0, 0, 0, 0, 0]
let numbers = [2, 4, 6, 9];
numbers.fill(10); // Inicializamos las posiciones al valor 10
console.log(numbers); // [10, 10, 10, 10]
```

- **Array.fill(value, start, end)** → Este método hace lo mismo que antes pero rellenando el array desde una posición inicial (incluida) hasta una final (excluida). Si no se especifica la última posición hasta la que queremos rellenar el array se rellenará hasta el final del mismo.

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.fill(10, 2, 5); // Las posiciones 2,3,4 se ponen a 10
console.log(numbers); // [2, 4, 10, 10, 10, 16]
let numbers2 = [2, 4, 6, 9, 14, 16];
numbers2.fill(10, -2); // Las dos últimas posiciones se ponen a 10
console.log(numbers2); // [2, 4, 6, 9, 10, 10]
```

- **Array.find(condition)** → Encuentra y devuelve el primer valor que encuentre que cumple la condición que se establece (recibe una función anónima o lambda que devuelve true o false). Con findIndex, devolvemos la posición que ocupa ese valor en el array.

```
let numbers = [2, 4, 6, 9, 14, 16];
```

```
console.log(numbers.find(num => num >= 10)); // Imprime 14 (primer valor encontra
do >= 10)
console.log(numbers.findIndex(num => num >= 10)); // Imprime 4 (numbers[4] -> 14)
```

- **Array.copyWithin(target, startwith)** → Copia los valores del array empezando desde la posición startWith, hasta la posición target en el resto de posiciones del array (en orden). Por ejemplo:

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.copyWithin(3, 0); // [0] -> [3], [1] -> [4], [2] -> [5]
console.log(numbers); // [2, 4, 6, 2, 4, 6]
```

- **entries(), keys(), values()** → Estos métodos devuelven un iterador para recorrer el array con el formato clave-valor. Podemos iterar usando el método next() o usando el bucle for..of. Podemos incluso usar el operador spread (...).

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.copyWithin(3, 0); // [0] -> [3], [1] -> [4], [2] -> [5]
console.log(numbers); // [2, 4, 6, 2, 4, 6]

let numbers = [2, 4, 6, 9, 14, 16];
for(let [key, val] of numbers.entries()) { // Itera sobre parejas [indice,valor]
console.log(key + " = " + val); // 0 = 2, 1 = 4, 2 = 6, 3 = 9, 4 = 14, 5 = 16
}
for(let key of numbers.keys()) { // Itera sobre índices
console.log(key); // 0, 1, 2, 3, 4, 5
}
for(let val of numbers.values()) { // Itera sobre valores
console.log(val); // 2, 4, 6, 9, 14, 16
}
console.log(...numbers.entries()); // [0, 2] [1, 4] [2, 6] [3, 9] [4, 14] [5, 16]
```

Map

Un Map es una colección que guarda parejas de [clave,valor], los valores son accedidos usando la correspondiente clave. En JavaScript, un objeto puede ser considerado como un tipo de Mapa pero con algunas limitaciones (Sólo con strings y enteros como claves).

```
let obj = {
   0: "Hello",
   1: "World",
   prop1: "This is",
   prop2: "an object"
}
console.log(obj[1]); // Imprime "World"
console.log(obj["prop1"]); // Imprime "This is"
console.log(obj.prop2); // Imprime "an object"
```

La nueva colección Map permite usar cualquier objeto como clave. Creamos la colección usando el constructor new Map(), y podemos usar los métodos set, get y delete para almacenar, obtener o eliminar un valor basado en una clave.

```
let person1 = {name: "Peter", age: 21};
let person2 = {name: "Mary", age: 34};
let person3 = {name: "Louise", age: 17};
let hobbies = new Map(); // Almacenará una persona con un array de hobbies (string)
hobbies.set(person1, ["Tennis", "Computers", "Movies"]);
console.log(hobbies); // Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers",
"Movies"]}
hobbies.set(person2, ["Music", "Walking"]);
hobbies.set(person3, ["Boxing", "Eating", "Sleeping"]);
console.log(hobbies);
```

```
Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"], Object
▼{name: "Mary", age: 34} => ["Music", "Walking"], Object {name: "Louise", age: 17}
=> ["Boxing", "Eating", "Sleeping"]} i
  size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array[3]
    ▼ 0: {Object => Array[3]}
      ▼ key: Object
          age: 21
          name: "Peter"
        ▶ __proto__: Object
      ▼ value: Array[3]
          0: "Tennis"
          1: "Computers"
          2: "Movies"
          length: 3
```

Cuando usamos un objeto como clave, debemos saber que almacenamos una referencia a ese objeto (Luego veremos WeakMap). Por tanto, se debe usar la misma referencia para acceder a un valor que para eliminarlo en ese mapa.

```
console.log(hobbies.has(person1)); // true (referencia al objeto original almacenado)
console.log(hobbies.has({name: "Peter", age: 21})); // false (mismas propiedades pero obj
eto diferente!)
```

La propiedad **size** devuelve la longitud del mapa y podemos iterar a través por él usando [Symbol.iterator] o el bucle **for..of.** Para cada iteración, se devuelve un array con dos posiciones **0 → key y 1 → value.**

```
console.log(hobbies.size); // Imprime 3
hobbies.delete(person2); // Elimina person2 del Map
console.log(hobbies.size); // Imprime 2
console.log(hobbies.get(person3)[2]); // Imprime "Sleeping"
/** Imprime todo:
 * Peter: Tennis, Computers, Movies
 * Louise: Boxing, Eating, Sleeping */
for (let entry of hobbies) {
   console.log(entry[0].name + ": " + entry[1].join(", "));
}
for (let [person, hobArray] of hobbies) { // Mejor
```

```
    console.log(person.name + ": " + hobArray.join(", "));
}
hobbies.forEach((hobArray, person) => { // Mejor
    console.log(person.name + ": " + hobArray.join(", "));
});
```

Si tenemos un tenemos un array que contiene otros arrays con dos posiciones (key, value), podemos crear un mapa directamente a partir del mismo.

```
let prods = [
    ["Computer", 345],
    ["Television", 299],
    ["Table", 65]
];
let prodMap = new Map(prods);
console.log(prodMap); // Map {"Computer" => 345, "Television" => 299, "Table" => 65}
```

Set

Set es como Map, pero no almacena los valores (sólo la clave). Puede ser visto como una colección que no permite valores duplicados (en un array puede haber valores duplicados). Se usa, add, delete y has → son métodos que devuelven un booleano para almacenar, borrar y ver si existe un valor.

```
let set = new Set();
set.add("John");
set.add("Mary");
set.add("Peter");
set.delete("Peter");
console.log(set.size); // Imprime 2
set.add("Mary"); // Mary ya existe
console.log(set.size); // Imprime 2
// Itera a través de los valores
set.forEach(value => {
    console.log(value);
})
```

Podemos crear un Set desde un array (lo cual elimina los valores duplicados).

```
let names = ["Jennifer", "Alex", "Tony", "Johny", "Alex", "Tony", "Alex"];
let nameSet = new Set(names);
console.log(nameSet); // Set {"Jennifer", "Alex", "Tony", "Johny"}
```