



ECMAScript: Versiones de JavaScript

⌚ Created January 27, 2022 4:22 PM

⌚ Last edited time August 25, 2022 4:39 PM

≡ Teacher Oscar Barajas Tavares

≡ Labels Programacion Internet

► Examen

ECMA-262 - Ecma International

Kindly note that the normative copy is the HTML version; the PDF version has been produced to generate a printable document. This 13 th

 [https://www.ecma-international.org/publications-and-standards/stand...](https://www.ecma-international.org/publications-and-standards/standards/ecma-262/)

GitHub - tc39/ecma262: Status, proce...

This repository contains the source for the current draft of ECMA-262, the

 <https://github.com/tc39/ecma262#ecmascript-2022>

tc39/ecma262

Status, process, and documents for ECMA-262



157 Contributors 290 Issues 14k Stars 1k Forks



¿Qué se implementó en ES6?

▼ Parámetros por defecto

Poder establecer ciertos valores que le pasamos a una función de forma por defecto. Esto hace que siempre que se llame la función ya tenga estos parámetros consigo, si queremos cambiarlos solamente debemos modificarlos en el momento en que la llamamos cambiando los parámetros.

```
// Antes de ES6 function newFunction (name, age, country) { var  
name = name || 'Juan'; var age = age || 20; var country = countr  
y || 'Col'; console.log(name, age, country); } //ES6 function ne  
wFunction2 (name = 'Juan', age = 20, country = 'Col') { console.  
log(name, age, country); } newFunction2(); // 'Juan', 20, 'Col n  
ewFunction2('Maria', 23, 'Mx');
```

▼ Template Literals

Nos facilita la vida si queremos concatenar varios elementos en un mensaje.

```
// Antes de ES6 var hello = 'hello'; var world = 'world'; var ph  
rase = hello + ' ' + world; // ES6 var phrase2 = `${hello} ${wor  
ld}`;
```

▼ Multilínea en los Scripts

Anteriormente para poder declarar una variable multilinea se requería agregar `\n` al final del string, así como `+` al inicio del siguiente string, por ejemplo:

```
let lorem = "Proin sapien tortor, posuere ac dolor ut \n" + ", s  
emper sollicitudin orci.";
```

Con ES6 lo podemos realizar de la siguiente forma:

```
let lorem2 = `Suspendisse consequat justo enim, ut sollicitudin  
diam posuere eget`;
```

Como vemos en el extracto anterior, ya no es requerido utilizar el `\n` al final del string, así como `+` al inicio del siguiente string.

Recuerda usar la comilla francesa ````` en caso del ejemplo con ES6

▼ Desestructuración de Elementos

Mediante la Desestructuración de Elementos, podemos extraer valores de un objeto y asignarlos a variables independientes, primero veamos un ejemplo de como se realizaba anteriormente:

```
let person = { 'name': 'Kevin', 'age': 27, 'country': 'VE' }  
console.log(person.name, person.age);
```

Ahora mediante la desestructuración lo haríamos de la siguiente forma:

```
let {name, age, country} = person; console.log(name, age, country);
```

De esta manera se hace mas amigable, extraer la información del objeto.

▼ Spread Operator

El operador de propagación nos permite que los elementos de un array se expandan y, de esta manera, podemos añadir un array dentro de otro sin que el resultado sean arrays anidados, si no un único array al que se han añadido nuevos valores, mediante el operador `...`.

```
let team1 = ['Oscar', 'Julian', 'Ricardo']; let team2 = ['Valeria', 'Yesica', 'Camila']; let education = ['David', ...team1, ...team2]; console.log(education);
```

▼ Let y Const

A partir de ES6 hay 3 maneras de declarar una variable:

- **CONST:** Es una constante la cual NO cambiara su valor en ningún momento en el futuro.
- **VAR:** Es una variable que SI puede cambiar su valor y su scope es local.
- **LET:** Es una variable que también podrá cambiar su valor, pero solo vivirá(*funcionara*) en el bloque donde fue declarada.

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

Se aconseja utilizar let en lugar de var como buena práctica.

▼ Arrow Functions

Las **Arrow functions**, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de **ECMAScript 6**. Básicamente, se trata de reemplazar eliminar la palabra `function` y añadir `=>` antes de abrir las llaves:

```
const func = function () { return "Función tradicional."}; const  
func = () => { return "Función flecha."};
```

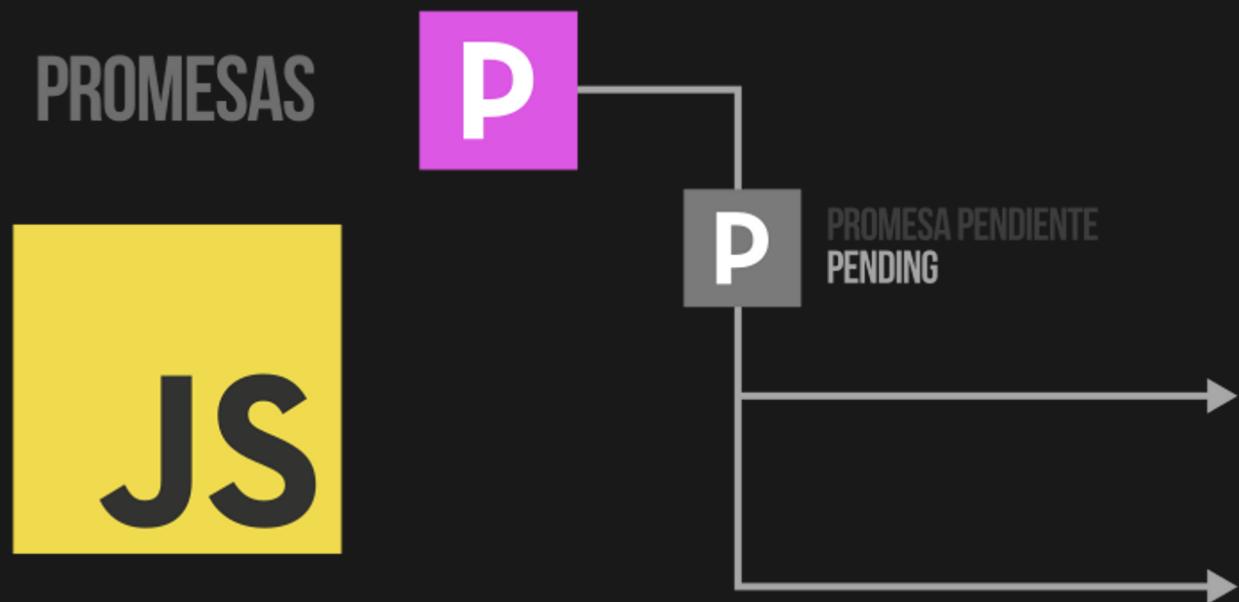
Sin embargo, las **funciones flechas** tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves `{}`.
- Además, en ese caso, automáticamente se hace un `return` de esa única línea, por lo que podemos omitir también el `return`.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: `() =>`.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: `e =>`.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: `(a, b) =>`.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: `({name: 'Manz'})`.

▼ Promesas y Parámetros en objetos

Las **promesas** son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica que, por ejemplo, utilizando funciones callbacks directamente.

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en Javascript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: **como consumirlas** (*utilizar promesas*) y **como crearlas** (*preparar una función para que use promesas y se puedan consumir*).

▼ Clases

Una **clase** es una forma de organizar código de forma entendible con el objetivo de simplificar el funcionamiento de nuestro programa. Además, hay que tener en cuenta que las clases son «conceptos abstractos» de los que se pueden crear objetos de programación, cada uno con sus características concretas.

Esto puede ser complicado de entender con palabras, pero se ve muy claro con ejemplos:



This image couldn't be found.

[Learn more](#)

lenguajejs.com (Error 404)

En primer lugar tenemos la **clase**. La clase es el **concepto abstracto** de un objeto, mientras que el **objeto** es el elemento final que se basa en la clase. En la imagen anterior tenemos varios ejemplos:

- En el **primer ejemplo** tenemos dos variables: `pato` y `lucas`. Ambos son animales, por lo que son objetos que están basados en la clase `Animal`. Tanto `pato` como `lucas` tienen las características que estarán definidas en la clase `Animal`: color, sonido que emiten, nombre, etc...
- En el **segundo ejemplo** tenemos dos variables `seat` y `opel`. Se trata de dos coches, que son vehículos, puesto que están basados en la clase `Vehículo`. Cada uno tendrá las características de su clase: color del vehículo, número de ruedas, marca, modelo, etc...
- En el **tercer ejemplo** tenemos dos variables `cuadrado` y `c2`. Se trata de dos formas geométricas, que al igual que los ejemplos anteriores tendrán sus propias características, como por ejemplo el tamaño de sus lados. El elemento `cuadrado` puede tener un lado de `3` cm y el elemento `c2` puede tener un lado de `6` cm.

En Javascript se utiliza una sintaxis muy similar a otros lenguajes como, por ejemplo, Java. Declarar una clase es tan sencillo como escribir lo siguiente:

```
// Declaración de una clase class Animal {} // Crear o instanciar  
un objeto const pato = new Animal();
```

El nombre elegido debería hacer referencia a la información que va a contener dicha clase. Piensa que el objetivo de las clases es almacenar en ella todo lo que tenga relación (*en este ejemplo, con los animales*). Si te fijas, es lo que venimos haciendo hasta ahora con objetos como `,`, `u` otros.

Observa que luego creamos una variable donde hacemos un `new Animal()`. Estamos creando una variable `pato` (*un objeto*) que es de tipo `Animal`, y que contendrá todas las características definidas dentro de la clase `Animal` (*de momento, vacía*).

Una norma de estilo en el mundo de la programación es que las clases deben siempre empezar en mayúsculas. Esto nos ayudará a diferenciarlas sólo con leerlas. Si te interesa este tema, puedes echar un vistazo al tema de las convenciones de nombres en programación.

▼ Módulos y Generadores

Importaciones nombradas

Puedes importar uno o más objetos o valores utilizando el nombre que se le definió en el módulo y que se haya declarado con la palabra clave `export`. Ejemplo:

```
// module.js export const myExport = "hola" // index.js import { myExport } from "module.js"
```

Importación predeterminada (*default*)

Cuando el módulo tiene una exportación predeterminada (*default*) omitimos el uso de llaves al momento de importar. Ejemplo:

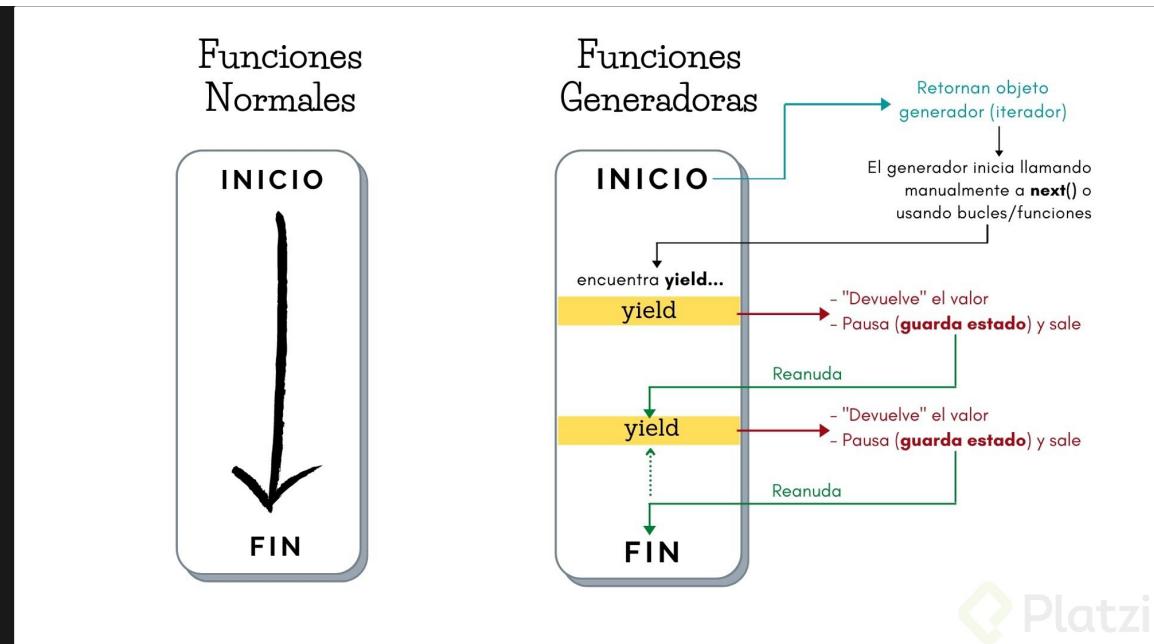
```
// module.js function myFunction() {...} export default myFunction  
// index.js import myFunction from "module.js"
```

Para importar los dos tipos de exportaciones podemos separarlos por comas. Ejemplo:

```
// module.js export const myExport = "hola" function myFunction()  
{} export default myFunction // index.js import myFunction, { myExport }
```

Funcion Generador

función que se puede pausar o detener para ser usada mas tarde, manteniendo el mismo contexto donde se le dejo anteriormente.



¿Qué se implementó en ES7?

En ES7 que salió en junio de 2016 se añadieron nuevas funcionalidades a JS, dentro de estas está **Includes** y **Potency**.

- **Includes:** Permite identificar un valor dentro de un arreglo o una variable.
- **Potencia:** Es simplemente una notación para realizar una potencia (**).

En el siguiente código se muestra el ejemplo:

```
// Includes: Trabaja sobre un arreglo o string, nos permite saber si hay un elemento dentro de este valor. Es una forma muy útil de validar elementos dentro de. let number =[1,2,3,4,6]; if(number.includes(6)){ console.log("Se encuentra el valor"); } else { console.log("No se encuentra el valor"); } //Elevar a la potencia: let base = 4; let exponent = 3; let resultado = base ** exponent; console.log(resultado);
```

¿Qué se implementó en ES8?

- **Object.entries()**: Devuelve una matriz de pares propios de una propiedad enumerable [key, value] de un objeto dado.
- **Object.values()**: Devuelve un array con los valores correspondientes a las propiedades enumerables de un objeto.

Object → Array

The diagram illustrates how ECMAScript's built-in `Object` methods can be used to convert an object into an array. It features a central yellow box containing code examples for `Object.keys()`, `Object.values()`, and `Object.entries()`. To the left, a white box shows the creation of an object `zoo` with keys `'lion'` and `'panda'` and their corresponding emoji values. To the right, three boxes show the resulting arrays produced by each method: `Object.keys(zoo)` returns `['lion', 'panda']`; `Object.values(zoo)` returns `[lion, panda]`; and `Object.entries(zoo)` returns `[['lion', 'lion'], ['panda', 'panda']]`.

```

object
const zoo = {
  lion: '🦁',
  panda: '🐼',
}

KEYS
VALUES
KEYS & VALUES

Object.keys(zoo)
// ['lion', 'panda']

Object.values(zoo)
// [ '🦁', '🐼' ]

Object.entries(zoo)
// [ ['lion', '🦁'], ['panda', '🐼'] ]

```

- **padStart()**: Rellena la cadena actual con una cadena dada, el relleno es aplicado desde el inicio (*izquierda*).
- **padEnd()**: Rellena la cadena actual con una cadena dada, el relleno se aplica desde el final

Padding

```
const string = 'hello';
console.log(string.padStart(7,'hi'));
```

A la constante que se llama "string" le agregamos un padding al inicio (en total tendríamos un máximo de 7 caracteres, "Lo que queremos agregar").....Agrega caracteres al inicio de los que tenemos.

```
console.log(string.padEnd(12, ' -----'));
```

A la constante que se llama "string" le agregamos un padding al final (en total tendríamos un máximo de 12 caracteres, "Lo que queremos agregar").....Agrega caracteres al final de los que tenemos.

Nota extra:

Cuando declaramos un objeto y al final de la línea dejamos una "," aun cuando no hay más objetos que lo siguen en la declaración, Estamos diciendo que puede agregarse más objetos en el futuro o puede que no.

► **Async Await**

Actualidad y próximos pasos de ECMAScript

▼ ¿Qué se implementó en ES9?

Operador de reposo

permite separar elementos de un objeto según se necesite. se utiliza `...all` para especificar que son los elementos que no se escogieron en la desestructuración del objeto.

```
/* Operador de reposo */ const obj = { name: 'santiago', age: 19, country: 'CO' } let { name, ...all} = obj console.log(name, all) // santiago { age: 19, country: 'CO' }
```

Propagación en objetos

Funciona de la misma forma que en Arrays, sirve para concatenar objetos según las propiedades que se necesite de un objeto específico.

```
// Propagación en objetos const obj1 = { name: 'santiago', age: 19 } const obj2 = { ...obj1, country: 'CO' } console.log(obj2) // { name: 'santiago', age: 19, country: 'CO' }
```

Promise Finally

Sirve para hacer una acción cuando se termina de obtener los datos de una promesa. En este ejemplo ejecuta `'finalizo'` cuando termina la ejecución de la promesa.

```
const helloWorld = () => { return new Promise((resolve, reject) => { (true) ? setTimeout(() => resolve('Hello World'), 3000) : reject(new Error('Test Error!')) }) } helloWorld() .then(response => console.log(response)) // Hello World .catch(error => console.log(error)) // Test Error .finally(() => console.log('Finalizo')) // Finalizo
```

Regex

Acceder a cada elemento de un Regular Expression para entender como se compone

```
/* Acceder a cada elemento de un Regular Expression para entender como se compone */ const regexData = /([0-9]{4})-([0-9]{2})-([0-9]{2})/ const match = regexData.exec('2018-04-20') const year = match[1] const month = match[2] const day = match[3] console.log(year, month, day)
```

▼ ¿Qué se implementó en ES10?

- **Array.prototype.flat(nivel_de_profundidad)**: un nuevo método que nos permite aplanar arreglos.
- **Array.prototype.flatMap()** lo mismo que flat con el beneficio de que primero manipular la data para luego poder aplanar.
- **String.prototype.trimStart() | String.prototype.trimEnd()** permite quitar los espacios al inicio o al final dependiendo de la función.
- **try/catch**: ahora puedes utilizarlo sin necesidad de especificarlo como catch(error) sino directamente usarlo en el scope del catch.
- **Object.fromEntries()** lo inverso a Object.entries(), es decir podemos convertir un objeto en una matriz clave/valor con Object.entries(), y hace lo inverso es decir de una matriz clave/valor a un objeto con Object.fromEntries().
- **Symbol.prototype.description**: permite regresar el descripción opcional del Symbol.

```
//-----method flat----- //devolver una
matriz con una submatriz aplanada, //recibe como argumento la
profundidad let array = [1,2,3, [1,2,3, [1,2,3]]];
console.log(array.flat(2)) //-----flatMap-----
----- //mapear cada elemento, luego pasarle una
funcion y aplanar let array = [1,2,3,4,5];
console.log(array.flatMap(value => [value, value * 2])); //-----
trim----- // let hello = '
hello world'; console.log(hello);
console.log(hello.trimStart()); // inicio let hello = 'hello
world ' ; console.log(hello); console.log(hello.trimEnd()); // final //-----optional catch biding----- try {
} catch /*(error) ya no es necesario colocarlo*/ { error } //-----
fromEntries----- // array to
object let entries = [["name", "oscar"], ["age", 32]];
console.log(Object.fromEntries(entries)) //-----symbol
object----- // let mySymbol = 'My Symbol'; let
symbol = Symbol(mySymbol); console.log(symbol.description);
```

