

# Curso de ECMAScript: Historia y Versiones de JavaScript

**ECMAScript** es una especificación de lenguaje de programación con la que trabaja JavaScript. Ecma International está a cargo de estandarizar este lenguaje de programación, a través de una serie de versiones que añaden funcionalidades nuevas.

La historia del **primer navegador web** empieza desde la necesidad de comunicar varias computadoras, a través de los siguientes acontecimientos:

- 1950: Las computadoras surgen para analizar temas de la Segunda Guerra Mundial.
- 1969: Surge la Red Arpanet, capaz de conectarse dos computadoras para compartir información.
- 1990: Tim Berners-lee creó las bases de la web, la World Wide Web.
- 1993: Se crea **Mosaic**, el primer navegador web.
- 1994: Marc Andreessen crea la empresa Netscape, y a su vez crea el primer navegador comercial con el mismo nombre, con enlaces e imágenes muy primitivas.

La **guerra de los navegadores** surge por la necesidad de las empresas de acaparar con el mercado de la web. En la primera guerra de navegadores, entre 1995 y 2001, se enfrentaron **Netscape** y **Microsoft** para posicionar comercialmente su propio navegador.

Los acontecimientos más importantes fueron:

- 1995: Microsoft crea su propio navegador web, Internet Explorer.
- 1996: Microsoft crea su propuesta de estilos para la web, CSS.
- 1995: Netscape crea su propuesta de lenguaje de programación para la web, Mocha. Después sería nombrado LiveScript, y finalmente JavaScript. **JavaScript** es un nombre elegido por marketing, ya que Java (otro lenguaje de programación) era muy popular en aquella época.
- 1995: Microsoft crea su propuesta de lenguaje de programación para la web, Jscript.
- 1997: Se crea ECMA, European Computer Manufacturer Association, para estandarizar los múltiples lenguajes de programación que estaban surgiendo por parte de Netscape, Microsoft, y otras empresas más. Este estándar se denomina **ECMAScript** o ES.

A partir de 1997, ECMA empezó a lanzar versiones para estandarizar el lenguaje. Algunas abandonadas, como la ES4.



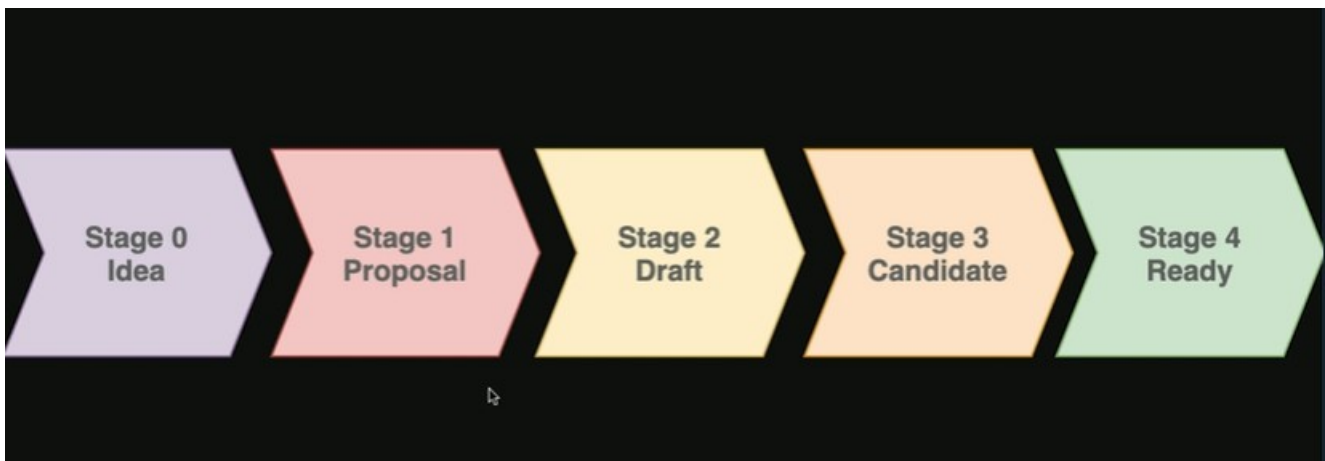
A partir de 2015, con ECMAScript 6, fue un antes y después para el lenguaje. Se incluyen varias funcionalidades que situaron a JavaScript como uno de los mejores lenguajes de programación.

**Cada navegador web tarda un tiempo en aplicar las nuevas características de ECMAScript.** Por lo que si utilizas una funcionalidad nueva, el navegador puede que no las procese y colapse tu programa.

**TC39** es un grupo de desarrolladores, académicos y hackers que están a cargo de revisar cada nueva propuesta o funcionalidad que cumpla con el estándar. El estándar es una serie de pasos que la nueva propuesta sigue para publicarla en la alguna versión de ECMAScript a futuro.

Las etapas de una nueva propuesta para ECMAScript son:

1. **Idea:** Una inquietud del desarrollador.
2. **Propuesta:** Cómo y por qué la idea soluciona un problema.
3. **Borrador:** Todo lo que implica la nueva funcionalidad detalladamente.
4. **Candidato:** La funcionalidad es probada y desarrollada por el comité.
5. **Preparada:** La funcionalidad está lista para ser publicada.



## VERSIONES DE JAVASCRIPT

En **ECMAScript 6** (ES6 o ES2015) fueron publicadas varias características nuevas que dotaron de gran poder al lenguaje, dos de estas son una nueva forma de declaración de variables con **let** y **const**, y **funciones flechas**.

En variables globales, **let** y **const** no guardan sus variables en el objeto global (**window**, **global** o **globalThis**), mientras que **var** sí los guarda. Esto es importante para que no exista re-declaración de variables.

Las **funciones flecha** (**arrow functions**) consiste en una función anónima con la siguiente estructura:

```
//Función tradicional
function nombre (parámetros) {
    return valorRetornado
}

//Función flecha
const nombre = (parámetros) => {
    return valorRetornado
}
```

Se denominan función flecha por el elemento => en su sintaxis.

Si existe un solo parámetro, puedes omitir los paréntesis:

```
const porDos = num => {
    return num * 2
}
```

Las funciones flecha tienen un retorno implícito, es decir, se puede omitir la palabra reservada return, para que el **código sea escrito en una sola línea**.

```
//Función tradicional
function suma (num1, num2) {
    return num1 + num2
}

//Función flecha
const suma = (num1, num2) => num1 + num2
```

Si el retorno requiere de más líneas y aún deseas utilizarlo de manera implícita, deberás envolver el cuerpo de la función entre paréntesis.

```
const suma = (num1, num2) => (
    num1 + num2
)
```

**NOTA:** Las arrow functions:

- No tienen sus propios enlaces a this o super y no se debe usar como métodos.
- No se puede utilizar como constructor.
- No se puede utilizar yield dentro de su cuerpo.
- No apta para los métodos call, apply y bind, que generalmente se basan en establecer un ámbito o alcance.

Las **plantillas literales (template literals)** consisten en crear cadenas de caracteres que puedan contener variables sin utilizar la concatenación. Esto mejora la legibilidad y la mantenibilidad del código.

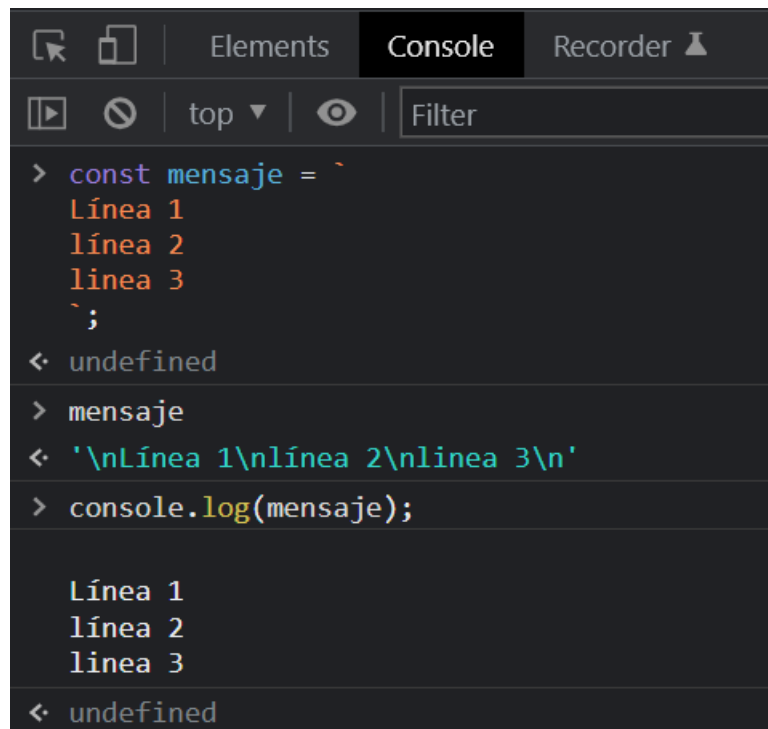
Las plantillas literales añadidas en ES6, se emplea el caracter acento grave ( ` ), para envolver el mensaje. Para incluir las variables se utiliza la sintaxis `${variable}`.

```
var nombre = "Andres"
var edad = 23

var mensaje = `Mi nombre es ${nombre} y tengo ${edad} años.`

console.log(mensaje)
// 'Mi nombre es Andres y tengo 23 años.'
```

La **plantilla multilinea** consiste en crear mensajes que contengan varias líneas separadas entre sí, utilizando las plantillas literales. Antes de ES6, la forma de crear una plantilla multilinea era agregar `\n` al string. Con ES6 solamente necesitas utilizar las plantillas literales.

The image shows a browser's developer console with the 'Console' tab selected. The console displays the execution of several JavaScript commands. First, a multiline string is assigned to a variable 'mensaje' using backticks and line breaks. Then, the variable is logged to the console, showing the string with its line breaks. Finally, the string is logged using console.log, and the output is displayed as three separate lines.

```
> const mensaje = `
  Línea 1
  línea 2
  linea 3
`;
< undefined

> mensaje
< '\nLínea 1\nlínea 2\nlinea 3\n'

> console.log(mensaje);

  Línea 1
  línea 2
  linea 3
< undefined
```

Los **parámetros por defecto (default params)** consisten en establecer un valor por defecto a los parámetros de una función, para asegurar que el código se ejecute correctamente en el caso de que no se establezcan los argumentos correspondientes en la invocación de la función.

```
function sumar(number1 = 0, number2 = 0){
  return number1 + number2
}

sumar(3,4) // 7
sumar(3)    // 3
sumar()     // 0
```

**Posición de los parámetros por defecto:** Si obligatoriamente necesitas el valor como argumento, ten presente que los parámetros por defecto siempre deben estar en las **posiciones finales**.

```
// ❌ Mal
function sumar(number1 = 0, number2) { ... }
sumar(3)    // number1 = 3 y number2 = undefined

// ✅ Bien
function sumar(number1, number2 = 0) { ... }
sumar(3)    // number1 = 3 y number2 = 0
```

**Function Rest Parameter:** The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

La **desestructuración (destructuring)** consiste en extraer los valores de arrays o propiedades de objetos en distintas variables.

La desestructuración de objetos implica extraer las propiedades de un objeto en variables. Mediante el mismo nombre de la propiedad del objeto con la siguiente sintaxis:

```
const objeto = {
  prop1: "valor1",
  prop2: "valor2",
  ...
}

// Desestructuración
const { prop1, prop2 } = objeto
```

**Cambiar el nombre de las variables con desestructuración:** Si no te agrada el nombre de la propiedad del objeto, puedes cambiarlo utilizando la siguiente sintaxis:

```
const usuario = { nombre: "Andres", edad: 23, plataforma: "Platzi" }

const { nombre: name, edad: age, plataforma: platform } = usuario

console.log(name) // 'Andres'
console.log(age) // 23
console.log(platform) // 'Platzi'

console.log(nombre) // Uncaught ReferenceError: nombre is not defined
```

**Desestructuración en parámetros de una función:** Si envías un objeto como argumento en la invocación a la declaración de una función, puedes utilizar la desestructuración en los parámetros para obtener los valores directamente. Ten en cuenta que el nombre debe ser igual a la propiedad del objeto.

```
const usuario = { nombre: "Andres", edad: 23, plataforma: "Platzi" }

function mostrarDatos( { nombre, edad, plataforma } ){
  console.log(nombre, edad, plataforma)
}

mostrarDatos(usuario) // 'Andres', 23, 'Platzi'
```

La **desestructuración de arrays** consiste en extraer los valores de un array en variables, utilizando la misma posición del array con una sintaxis similar a la desestructuración de objetos.

```
const array = [ 1, 2, 3, 4, 5 ]

// Desestructuración
const [uno, dos, tres] = array

console.log(unos) // 1
console.log(dos) // 2
console.log(tres) // 3
```

**Desestructuración para valores retornados de una función:** Cuando una función retorna un array, puedes guardarlo en una variable. Por ende, puedes utilizar la desestructuración para utilizar esos valores por separado de manera legible.

En el siguiente ejemplo, la función **useState** retorna un array con dos elementos: un valor y otra función actualizadora.

```
function useState(value){
  return [value, updateValue()]
}

//Sin desestructuración
const estado = useState(3)
const valor = estado[0]
const actualizador = estado[1]

//Con desestructuración
const [valor, actualizador] = useState(3)
```

**Lo que puedes hacer con desestructuración, pero no es recomendable**

Si necesitas un elemento en cierta posición, puedes utilizar la separación por comas para identificar la variable que necesitas.

```
const array = [ 1, 2, 3, 4, 5 ]

const [,,, cinco] = array

console.log(cinco) // 5
```

Como los arrays son un tipo de objeto, puedes utilizar la desestructuración de objetos mediante el **índice** y **utilizando un nombre para la variable**.

**IMAGEN ABAJO**

```
const array = [ 1, 2, 3, 4, 5 ]

const {4: cinco} = array

console.log(cinco) // 5
```

El **operador de propagación** (*spread operator*), como su nombre lo dice, consiste en propagar los elementos de un iterable, ya sea un array o string utilizando tres puntos (...) dentro de un array.

```
// Para strings
const array = [ ..."Hola" ] // [ 'H', 'o', 'l', 'a' ]

// En arrays
const otherArray = [ ...array ] // [ 'H', 'o', 'l', 'a' ]
```

Para realizar una copia de un array, deberás tener cuidado de la **referencia en memoria**. Los arrays se guardan en una referencia en la memoria del computador, al crear una copia, este tendrá la misma referencia que el original. Debido a esto, **si cambias algo en la copia, también lo harás en el original**.

```
const originalArray = [1,2,3,4,5]
const copyArray = originalArray
copyArray[0] = 0

originalArray // [0,2,3,4,5]
originalArray === copyArray // true
```

Para evitar esto, utiliza el operador de propagación para crear una copia del array que utilice una **referencia en memoria diferente al original**.

```
const array1 = [1,2,3]
const number = 4
const array2 = [5,6,7]

const otherArray = [ ...array1, number, ...array2 ]

otherArray // [1,2,3,4,5,6,7]
```



**Cuidado con la copia en diferentes niveles de profundidad:** El operador de propagación sirve para producir una copia en **un solo nivel de profundidad**, esto quiere decir que si existen objetos o arrays dentro del array a copiar. Entonces los sub-elementos en cada nivel, tendrán la **misma referencia de memoria en la copia y en el original**.

Sin embargo, recientemente salió una forma de producir una copia profunda con **StructuredClone**, aunque es una característica muy reciente, así que revisa que navegadores tienen soporte. Este comportamiento también sucede para objetos dentro de otros objetos, u objetos dentro de arrays.

```
const originalArray = [1, [2,3] ,4,5]
const copyArray = structuredClone(originalArray)

originalArray === copyArray // false
originalArray[1] === copyArray[1] // false
```

El **parámetro rest** consiste en agrupar el residuo de elementos mediante la sintaxis de tres puntos (...) seguido de una variable que contendrá los elementos en un array.

Esta característica sirve para crear funciones que acepten cualquier número de argumentos para agruparlos en un array. El parámetro rest siempre deberá estar en la **última posición** de los parámetros de la función, caso contrario existirá un error de sintaxis.

```
function hola (primero, segundo, ...resto) {
  console.log(primero, segundo) // 1 2
  console.log(resto) // [3,4,5,6]
}

hola(1,2,3,4,5)
```

**Diferencias entre el parámetro rest y el operador de propagación:** Aunque el parámetro rest y el operador de propagación utilicen la misma sintaxis, son diferentes.

El parámetro rest agrupa el residuo de elementos y siempre debe estar en la última posición en los parametros al momento de definir una función, mientras que el operador de propagación expande los elementos de un iterable en un array y no importa en que lugar esté situado.

```
const array = [1,2,3,4,5]

function hola (primero, segundo, ...resto) { // <- Parámetro Rest
  console.log(primero, segundo) // 1 2
  console.log(resto) // [3,4,5, "final"]
}

hola(...array, "final") //<- Operador de propagación
//Lo mismo que hacer -> hola(1,2,3,4,5, "final")
```

Los **objetos literales** consiste en crear objetos a partir de variables sin repetir el nombre.

```
const nombre = "Andres"
const edad = 23
const esteEsUnID = 1

const objeto = {
  nombre,
  edad,
  id: esteEsUnID
}

objeto // { nombre: 'Andres', edad: 23, id: 1 }
```

Una **promesa** es un objeto de javascript que representa el eventual resultado (o error) de una operación asíncrona.

Los tres posibles estados de una promesa son: Pendiente, Cumplida, Rechazada.

El objeto de la promesa se asocia a una función callback que se ejecuta al cumplirse la operación asíncrona.

Una función callback es una función que se pasa a otra función como argumento y luego se ejecuta dentro de la función externa.

Las promesas tienen un método **.then()**, con el cual podemos decidir qué ocurre cuando se completa la promesa (éxito o error).

```
//Primer ejemplo
const promesaCumplida = false;
const miPromesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (promesaCumplida) {
      resolve('¡Promesa cumplida!');
    } else {
      reject('Promesa rechazada...');
    }
  }, 3000);
});

function manejarResolve(valor) {
  console.log(valor);
}

function manejarReject(razonRechazo) {
  console.log(razonRechazo);
}

miPromesa.then(manejarResolve, manejarReject);
```

En el ejemplo de arriba, los parametros `resolve` y `reject` son funciones que representaran el resultado de la promesa.

En el metodo `.then()`, el parametro (valor) de la funciones `manejarResolve` y `manejarReject` representa el argumento que se le pasa a `resolve/reject`.

-----  
Para que el código de JavaScript sea más ordenado, legible y mantenible; ES6 introduce una forma de manejar código en **archivos de manera modular**. Esto involucra **exportar** funciones o variables de un archivo, e **importarlas** en otros archivos donde se necesite.

**Cómo utilizar los módulos de ECMAScript:** Debes tener mínimo dos archivos, uno para exportar las funcionalidades y otro que las importe para ejecutarlas.

Además, si iniciaste un proyecto con **NPM** (Node Package Manager) con Node.js, necesitas especificar que el código es modular en el archivo `package.json` de la siguiente manera:

```
// package.json
{
  ...
  "type": "module"
}
```

**Qué son las exportaciones de código:** Las exportaciones de código consisten en crear funciones o variables para utilizarlas en otros archivos mediante la palabra reservada **export**. Existen dos formas de exportar: 1) Antes de declarar la funcionalidad, 2) o entre llaves `{}`.

Por ejemplo en un archivo, declaramos una función para sumar dos valores, el cual lo exportaremos de dos maneras.

```
//math_function.js
export const add = (x,y) => {
  return x + y
}
```

```
//math_function.js
const add = (x,y) => {
  return x + y
}

export { add, otherFunction, ... }
```

**Qué son las importaciones de código:** Las importaciones de código consiste en usar funciones o variables de otros archivos mediante la palabra reservada **import**, que deberán estar siempre lo más arriba del archivo y utilizando el mismo nombre que el archivo original.

Por ejemplo, importamos la función `add` de otro archivo para utilizarla en el archivo actual.

```
// main.js
import { add, otherFunction } from './math_functions.js'

add(2,2) //4
```

Para **importar todas las funcionalidades de un archivo** se utiliza un asterisco (\*) y se puede cambiar el nombre para evitar la repetición de variables o funciones a través de la palabra reservada **as**.

```
// main.js
import * as myMathModule from './math_functions.js';

myMathModule.add(2,2) //4
myMathModule.otherFunction()
```

**Exportaciones por defecto:** Si solo UN valor será exportado, entonces se puede utilizar **export default**. De esta manera no es necesario las llaves {} al exportar e importar.

```
//math_function.js
export default function add (x,y){
    return x + y;
}
```

Adicionalmente, no se puede usar export default antes de declaraciones const, let o var, pero puedes exportarlas al final.

```
// ❌ Erróneo
export default const add = (x,y) => {
    return x + y;
}

// ✅ Correcto
const add = (x,y) => {
    return x + y;
}

export default add
```

**Importaciones por defecto:** Si únicamente un valor será importado, entonces se puede utilizar cualquier nombre en la importación. De esta manera no es necesario las llaves {}. Sin embargo, **es recomendable utilizar siempre el nombre de la función**, para evitar confusiones.

```
//Las siguientes importaciones son válidas
import add from './math_functions.js'
import suma from './math_functions.js'
import cualquierNombre from './math_functions.js'
```

**Combinar ambos tipos de exportaciones e importaciones:** Teniendo las consideraciones de importaciones y exportaciones, nombradas y por defecto, entonces podemos combinarlas en un mismo archivo.

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

export default myFunction

// main.js
import myFunction, { myExport } from "/module.js"
```

Los **generadores** son funciones especiales que pueden pausar su ejecución, luego volver al punto donde se quedaron, recordando su scope y seguir retornando valores. Estos se utilizan para guardar la totalidad de datos infinitos, a través de una función matemática a valores futuros. De esta manera ocupan poca memoria, con respecto a si creamos un array u objeto.

La sintaxis de los generadores comprende lo siguiente:

- La palabra reservada **function\*** (con el asterisco al final).
- La palabra reservada **yield** que hace referencia al valor retornado cada vez que se invoque, recordando el valor anterior.
- Crear una variable a partir de la función generadora.
- El método next devuelve un objeto que contiene una propiedad **value** con cada valor de yield; y otra propiedad **done** con el valor true o false si el generador ha terminado.

Por ejemplo, creemos un generador para retornar tres valores.

IMAGEN ABAJO

```
function* generator(){
  yield 1
  yield 2
  yield 3
}

const generador = generator()

generador.next().value //1
generador.next().value //2
generador.next().value //3
generador.next() // {value: undefined, done: true}
```

**Set** es una nueva estructura de datos para almacenar elementos **únicos**, es decir, sin elementos repetidos. Para iniciar un Set, se debe crear una instancia de su clase a partir de un iterable. Generalmente, un iterable es un array.

```
const set = new Set(iterable)
```

Para manipular estas estructuras de datos, existen los siguientes métodos:

- **add(value)**: añade un nuevo valor.
- **delete(value)**: elimina un elemento que contiene el Set, retorna un booleano si existía o no el valor.
- **has(value)**: retorna un booleano si existe o no el valor dentro del Set.
- **clear(value)**: elimina todos los valores del Set.
- **size**: retorna la cantidad de elementos del Set.

# ES7

El método **includes** determina si un array o string incluye un determinado elemento. Devuelve true o false, si existe o no respectivamente.

Este método recibe dos argumentos:

1. El elemento a comparar.
2. El índice inicial (**opcional**) desde donde comparar hasta el último elemento.

Los índices positivos comienzan desde 0 hasta la **longitud total menos uno**, de izquierda a derecha del array.

```
[0,1,2,3, ..., lenght-1]
```

Los índices negativos comienzan desde -1 hasta el negativo de la longitud total del array, de derecha a izquierda.

```
[-lenght, ..., -3, -2, -1]
```

El método includes se utiliza para arrays y strings. El método es sensible a mayúsculas, minúsculas y espacios.

```
//Utilizando strings
const saludo = "Hola mundo"

saludo.includes("Hola") // true
saludo.includes("Mundo") // false
saludo.includes(" ") // true
saludo.includes("Hola", 1) // false
saludo.includes("mundo", -5) // true
```

```
// Utilizando arrays
const frutas = ["manzana", "pera", "piña", "uva"]

frutas.includes("manzana") // true
frutas.includes("Pera") // false
frutas.includes("sandía") // false
frutas.includes("manzana", 1) // false
frutas.includes("piña", -1) // false
frutas[0].includes("man") // true
```

Los métodos de transformación de objetos a arrays sirven para obtener la información de las propiedades, sus valores o ambas.

**Object.entries()** devuelve un array con las entries en forma [propiedad, valor] del objeto enviado como argumento.

```
const usuario = {
  name: "Andres",
  email: "andres@correo.com",
  age: 23
}

Object.entries(usuario)
/*
[
  [ 'name', 'Andres' ],
  [ 'email', 'andres@correo.com' ],
  [ 'age', 23 ]
]
*/
```

**Object.keys()** devuelve un array con las propiedades (keys) del objeto enviado como argumento.

```
const usuario = {
  name: "Andres",
  email: "andres@correo.com",
  age: 23
}

Object.keys(usuario)
// [ 'name', 'email', 'age' ]
```

**Object.values()** devuelve un array con los valores de cada propiedad del objeto enviado como argumento.

```
const usuario = {
  name: 'Andres',
  email: "andres@correo.com",
  age: 23
}

Object.values(usuario)
// [ 'Andres', 'andres@correo.com', 23 ]
```



# ES8

El **padding** consiste en rellenar un string por el principio o por el final, con el carácter especificado, repetido hasta que complete la longitud máxima.

Este método recibe dos argumentos:

- La longitud máxima a rellenar, incluyendo el string inicial.
- El string para rellenar, por defecto, es un espacio.

Si la longitud a rellenar es menor que la longitud del string actual, entonces no agregará nada.

El método **padStart** completa un string con otro string en el inicio hasta tener un total de caracteres especificado.

```
'abc'.padStart(10) // "      abc"
'abc'.padStart(10, "foo") // "foofoofabc"
'abc'.padStart(6, "123465") // "123abc"
'abc'.padStart(8, "0") // "00000abc"
'abc'.padStart(1) // "abc"
```

El método **padEnd** completa un string con otro string en el final hasta tener un total de caracteres especificado.

```
'abc'.padEnd(10) // "abc      "
'abc'.padEnd(10, "foo") // "abcfoofoof"
'abc'.padEnd(6, "123456") // "abc123"
'abc'.padEnd(1) // "abc"
```

Las **trailing commas** consisten en comas al final de objetos o arrays que faciliten añadir nuevos elementos y evitar errores de sintaxis.

```
const usuario = {
  name: 'Andres',
  email: "andres@correo.com",
  age: 23, //<-- Trailing comma
}

const nombres = [
  "Andres",
  "Valeria",
  "Jhesly", //<-- Trailing comma
]
```

La función **asíncrona** se crea mediante la palabra reservada `async` y retorna una **promesa**.

```
async function asyncFunction () {...}  
  
const asyncFunction = async () => { ... }
```

La palabra reservada **await** significa que espera hasta que una promesa sea resuelta y solo funciona dentro de una función asíncrona. Los bloques `try / catch` sirven para manejar si la promesa ha sido resuelta o rechazada.

```
async function asyncFunction () {  
  try {  
    const response = await promesa()  
    return response  
  } catch (error) {  
    return error  
  }  
}
```

## ES9

Las **expresiones regulares** o RegEx (regular expressions) son patrones de búsqueda y manipulación de cadenas de caracteres increíblemente potente y están presentes en todos los lenguajes de programación.

En JavaScript se crea este patrón entre barras inclinadas (`/patrón/`) y se utiliza métodos para hacer coincidir la búsqueda.

```
const regexData = /([0-9]{4})-([0-9]{2})-([0-9]{2})/  
const match = regexData.exec('2018-04-20')
```

Las **propiedades de propagación** consisten en expandir las propiedades de un objeto utilizando el spread operator. Sirve para crear nuevos objetos a partir de otros.

IMAGEN ABAJO

```
const objeto = {
  nombre: "Andres",
  age: 23,
}

const usuario = {
  ...objeto,
  plataforma: "Platzi"
}
```

**Crear copias de objetos utilizando las propiedades de propagación:** Semejante a crear copias de arrays utilizando el operador de propagación, se puede realizar copias de objetos **en un solo nivel** mediante las propiedades de propagación.

```
const objetoOriginal = {a: 1, b: 2}
const objetoReferencia = objetoOriginal
const objetoCopia = {...objetoOriginal}

objetoReferencia === objetoOriginal // true
objetoOriginal === objetoCopia // false
```

**Cuidado con la copia en diferentes niveles de profundidad:** El operador de propagación sirve para crear una copia **en un solo nivel de profundidad**, esto quiere decir que si existen objetos o arrays dentro de un objeto a copiar. Entonces los sub-elementos en cada nivel, tendrán la misma referencia en la copia y en el original.

```
const original = { datos: [1, [2, 3], 4, 5] }
const copia = { ...original }

original === copia // false
original["datos"] === copia["datos"] // true
```

Recientemente salió una forma de crear una copia profunda con **StructuredClone**.

```
const original = { datos: [1, [2, 3], 4, 5] }
const copia = structuredClone(original)

original === copia // false
original["datos"] === copia["datos"] // false
```

El método **finally** para promesas consiste en ejecutar código después que una promesa haya sido ejecutada como resuelta o rechazada.

```
promesa()
  .then(response => console.log(response)) // Promesa resuelta
  .catch(error => console.log(error)) // Promesa rechazada
  .finally(() => console.log("Independientemente del estado final de la promesa, esto se tiene que hacer...")); // Código final
```

**Generadores asíncronos:** Los generados asíncronos son semejantes a los generadores que ya conoces, pero combinando sintaxis de promesas.

```
async function* anotherGenerator() {
  yield await Promise.resolve(1)
  yield await Promise.resolve(2)
  yield await Promise.resolve(3)
}

const generador = anotherGenerator()
generador.next().then(respuesta => console.log(respuesta.value))
generador.next().then(respuesta => console.log(respuesta.value))
generador.next().then(respuesta => console.log(respuesta.value))
```

**Cómo utilizar for await:** De la misma manera, for await es un ciclo repetitivo que se maneja asíncronamente. El ciclo siempre debe estar dentro de una función con async.

```
async function* asyncGenerator() {
  let i = 0;
  while (i < 3) {
    yield i++;
  }
}

(async () => {
  for await (const num of asyncGenerator()) {
    console.log(num);
  }
})();
// 0
// 1
// 2
```

# ES10

El aplanamiento consiste en transformar un array de arrays a una sola dimensión. Los métodos **flat** y **flatMap** permitirán realizar el aplanamiento.

El método **flat** devuelve un array donde los sub-arrays han sido propagados hasta una profundidad especificada. Este método es **immutable**, es decir, retorna un nuevo array con los cambios y no cambia el array original.

Este método recibe un argumento: La **profundidad del aplanamiento**, por defecto, tiene un valor de 1.

Si se desea aplanar todos los sub-arrays en una sola dimensión, **utiliza el valor de Infinity**.

```
const array = [1,2,[3,4],5,6]
const result = array.flat()
result// [1,2,3,4,5,6]

const array2 = [1, 2, [3, 4, [5, 6]]];
const result2 = array2.flat()
result2// [1, 2, 3, 4, [5, 6]]

const array3 = [1, 2, [3, 4, [5, 6]]]
const result3 = array3.flat(2)
result3// [1, 2, 3, 4, 5, 6]

const array4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]]
const result4 = array4.flat(Infinity)
result4// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

El método **flatMap** es una combinación de los métodos map y flat. Primero realiza la iteración de los elementos del array (como si fuera map), y después los aplanan en **una sola profundidad** (como si fuera flat). Esto sería equivalente a esto: **array.map(...args).flat()**

Este método es **immutable**, es decir, retorna un nuevo array con los cambios y no cambia el array original.

```
const arr1 = [1, 2, [3], [4, 5], 6, []];

const flattened = arr1.flatMap(num => num * 2);

console.log(flattened);
// Expected output: Array [2, 4, 6, NaN, 12, 0]
```

Existen tres métodos para eliminar espacios en blanco de un string:

- El método **trim** elimina los espacios en blanco al inicio y al final.
- El método **trimStart** o **trimLeft** elimina los espacios al inicio.
- El método **trimEnd** o **trimRight** elimina los espacios al final.

**Cómo transformar un array de arrays en un objeto:** El método **Object.fromEntries** devuelve un objeto a partir de un array donde sus elementos son las entries en forma [propiedad, valor].

```
const arrayEntries = [
  [ 'name', 'Andres' ],
  [ 'email', 'andres@correo.com' ],
  [ 'age', 23 ]
]

const usuario = Object.fromEntries(arrayEntries)

console.log(usuario)
/* {
  name: 'Andres',
  email: 'andres@correo.com',
  age: 23
}
*/
```

## ES11

Cuando intentas acceder a propiedades de un objeto que no existen, JavaScript te retornará **undefined**. Al acceder a una propiedad más profunda de un objeto, que previamente fue evaluada como **undefined**, el programa se detendrá y mostrará un error de tipo.

```
const usuario = {}
console.log(usuario.redes.facebook)
// TypeError: Cannot read properties of undefined (reading 'facebook')
```

Es como intentar ejecutar **undefined.facebook**, lo cual es un error de tipo, **debido a que undefined es un primitivo**, no es un objeto.

**Cómo utilizar el encadenamiento opcional:** El encadenamiento opcional u **optional chaining** (?.) detiene la evaluación del objeto cuando el valor es undefined o null antes del (?.), retornando **undefined** sin detener el programa por un error.

```
const usuario = {}
console.log(usuario.redes?.facebook)
// undefined
```

Pero, ¿por qué usaría propiedades de un objeto vacío? Cuando realizas **peticiones**, el objeto no contiene la información solicitada en seguida, por ende, necesitas que el **programa no colapse** hasta que lleguen los datos y puedas utilizarlos.

El **encadenamiento opcional** se debe utilizar únicamente cuando probablemente un valor no exista.

Si abusas del encadenamiento opcional y existe un error en un objeto, el programa podría “**ocultarlo**” por un **undefined**, provocando que el debugging sea más complicado.

**Big Int, enteros muy grandes:** El nuevo dato primitivo **bigint** permite manejar números enteros muy grandes. Existen dos formas de crear un bigint: el número entero seguido de n o mediante la función **BigInt**.

JavaScript tiene límites numéricos, un máximo `Number.MAX_SAFE_INTEGER` y un mínimo `Number.MIN_SAFE_INTEGER`.

```
const number1 = 45n
const number2 = BigInt(45)

typeof 45n // 'bigint'
```

Después de los límites, los cálculos muestran resultados erróneos. Los **bigint** ayudan a manejar operaciones de enteros fuera de los límites mencionados.

```
const increment = 2
const number = Number.MAX_SAFE_INTEGER + increment
const bigint = BigInt(Number.MAX_SAFE_INTEGER) + BigInt(increment)

console.log(number) // 9007199254740992
console.log(bigInt) // 9007199254740993n
```

**Operador Nullish Coalescing:** El operador nullish coalescing (??) consiste en evaluar una variable si es **undefined** o **null** para asignarle un valor.

El siguiente ejemplo se lee como: ¿usuario.name es **undefined** o **null**? Si es así, asígnale un valor por defecto "Andres", caso contrario asigna el valor de usuario.name.

```
const usuario1 = {}
const nombre1 = usuario1.name ?? "Andres"

const usuario2 = {name: "Juan"}
const nombre2 = usuario2.name ?? "Andres"

console.log(nombre1) // 'Andres'
console.log(nombre2) // 'Juan'
```

**Diferencia entre el operador OR y el Nullish coalescing:** El operador OR (||) **evalúa un valor falsey**. Un valor falsey es aquel que es falso en un contexto booleano, estos son: 0, "" (string vacío), false, NaN, undefined o null.

Puede que recibas una variable con un valor falsey que necesites asignarle a otra variable, que no sea null o undefined. Si evalúas con el operador OR, este lo cambiará, provocando un resultado erróneo.

```
const id = 0

const orId = id || "Sin id"
const nullishId = id ?? "Sin id"

console.log( orId ) // 'Sin id'
console.log( nullishId ) // 0
```

**Promise.all:** El método Promise.all sirve para manejar varias promesas al mismo tiempo. Recibe como argumento un array de promesas. El problema es que Promise.all() se resolverá, si y solo si **todas las promesas fueron resueltas**. Si al menos una promesa es rechazada, Promise.all será rechazada.

```
Promise.all([promesa1, promesa2, promesa3])
  .then(respuesta => console.log(respuesta))
  .catch(error => console.log(error))
```

**Promise.allSettled:** Promise.allSettled() permite manejar varias promesas, que devolverá un array de objetos con el **estado y el valor de cada promesa, haya sido resuelta o rechazada**.

```
const promesa1 = Promise.reject("Ups promesa 1 falló");
const promesa2 = Promise.resolve("Promesa 2");
const promesa3 = Promise.reject("Ups promesa 3 falló");

Promise.allSettled([promesa1, promesa2, promesa3])
  .then(respuesta => console.log(respuesta));

//Resultado
[
  { status: 'rejected', reason: 'Ups promesa 1 falló' },
  { status: 'fulfilled', value: 'Promesa 2' },
  { status: 'rejected', reason: 'Ups promesa 3 falló' }
]
```



**Objeto global para cualquier plataforma:** El motor de JavaScript, aquel que compila tu archivo y lo convierte en código que entiende el computador, al iniciar la compilación crea un **objeto global**.

El objeto global proporciona funciones y variables propias e integradas en el lenguaje o el entorno. Dependiendo la plataforma, este objeto global tendrá un nombre diferente.

En el navegador el objeto global es **window**, para Node.js es **global**, y así para cada entorno. Sin embargo, en Node.js no podrás acceder a window, ni en el navegador podrás acceder a global.

Para estandarizar el objeto global se creó **globalThis**, un objeto compatible para cualquier plataforma.

```
// Ejecuta el siguiente código y observa que muestra
console.log(window)
console.log(globalThis)

// En el navegador
window === globalThis // true

// En Node.js
global === globalThis // true
```

**Método matchAll para expresiones regulares:** El método **matchAll** de los strings devuelve un iterable con todas las coincidencias del string específico a partir de una expresión regular, colocada como argumento: *string.matchAll(regex)*

En el iterable, existe una propiedad denominada index con el índice del string donde la búsqueda coincide.

```
{
  const regex = /\b(Apple)+\b/g;
  const fruit = "Apple, Banana, Kiwi, Apple, Orange, etc. etc. etc."
  // Transformación del iterable retornado a array
  const array = [...fruit.matchAll(regex)];
  console.log(array);
}
//RESULTADO
[
  [
    'Apple',
    'Apple',
    index: 0,
    input: 'Apple, Banana, Kiwi, Apple, Orange, etc. etc. etc.',
    groups: undefined
  ],
  [
    'Apple',
    'Apple',
    index: 21,
    input: 'Apple, Banana, Kiwi, Apple, Orange, etc. etc. etc.',
    groups: undefined
  ]
]
```

La **importación dinámica** consiste en cargar los módulos cuando el usuario los vaya a utilizar, y no al iniciar la aplicación. Esto permite que la página web sea más rápida, porque descarga menos recursos.

La expresión `import()` recibe un argumento de tipo string con la ruta del módulo a importar y devuelve una **promesa**.

```
const ruta = "./modulo.js"

// Utilizando promesas
import(ruta)
  .then( modulo => {
    modulo.funcion1()
    modulo.funcion2()
  })
  .catch(error => console.log(error))

// Utilizando async/await
async function importarModulo(rutaDelModulo) {
  const modulo = await import(rutaDelModulo)
  modulo.funcion1()
  modulo.funcion2()
}

importarModulo(ruta)
```

De esta manera puedes utilizar una importación dinámica en tu aplicación para desencadenar una descarga de un módulo cuando el usuario lo vaya a utilizar. Por ejemplo, al realizar clic en un botón.

```
const boton = document.getElementById("boton")

boton.addEventListener("click", async function () {
  const modulo = await import('./modulo.js')
  modulo.funcion()
})
```

## ES12

Los **separadores numéricos** ayudan a la legibilidad de cantidades con varias cifras. Se utiliza el caracter guion bajo ( `_` ) para separar las cifras, y no afecta a la ejecución del programa.

Lo ideal es separar cada 3 cifras, para visualizar los miles, millones, billones, etc.

```
// ▼ Baja legibilidad
const numero1 = 3501548945
console.log( numero1 ) // 3501548945

// ✔ Alta legibilidad
const numero2 = 3_501_548_945
console.log( numero1 ) // 3501548945
```

El método **replaceAll** retorna un nuevo string, reemplazando todos los elementos por otro. Este método recibe dos argumentos:

- El **patrón a reemplazar**, puede ser un string o una expresión regular.
- El **nuevo elemento** que sustituye al reemplazado.

Este procedimiento fue creado para solucionar el problema que tenía el método replace, que realizaba la misma función de reemplazar elementos, pero solamente una sola vez por invocación.

```
const mensaje = "JavaScript es maravilloso, con JavaScript puedo crear el futuro de la web."

mensaje.replace("JavaScript", "Python")
// 'Python es maravilloso, con JavaScript puedo crear el futuro de la web.'

mensaje.replaceAll("JavaScript", "Python")
// 'Python es maravilloso, con Python puedo crear el futuro de la web.'

mensaje.replaceAll(/a/g, "")
// 'J*v*Script es m*r*villoso, con J*v*Script puedo cre*r el futuro de l* web.'
```

**Métodos privados de clases:** Los métodos privados consiste en limitar el acceso a propiedades y métodos agregando el caracter numeral ( #). Por defecto, las propiedades y métodos de una clase en JavaScript son públicas, es decir, se puede acceder a ellos fuera de la clase.

```
class UnaClase {
  #private(valor){
    console.log(valor);
  }

  public(valor){
    console.log(valor);
  }
}

const objClase = new UnaClase();
objClase.public("Hola"); // 'Hola'
objClase.private("Hola"); // TypeError: clase.private is not a function
```

**Promise.any:** Promise.any() es otra forma de manejar varias promesas, que retornará la **primera** promesa que sea resuelta y rebotará si todas las promesas son rechazadas.

```
const promesa1 = Promise.reject("Ups promesa 1 falló");
const promesa2 = Promise.reject("Ups promesa 2 falló");
const promesa3 = Promise.resolve("Promesa 3");
const promesa4 = Promise.resolve("Promesa 4");

Promise.any([promesa1, promesa2, promesa3, promesa4])
  .then(respuesta => console.log(respuesta)) // Promise 3
  .catch(error => console.log(error));
```

## ES13

El método **at** de arrays sirve para acceder a los elementos a partir del índice. **array.at(índice)**

Los **índices positivos** comienzan desde 0 hasta la longitud total menos uno, de izquierda a derecha del array. El índice 0 es la primera posición.

Los **índices negativos** comienzan desde -1 hasta el negativo de la longitud total del array, de derecha a izquierda. El índice -1 es la última posición.

**Cómo utilizar el método at:** Es bien sabido que puedes utilizar la notación de corchetes, pero necesitas obtener la longitud del array y restarle una unidad, generando mucho código que puede volverse difícil de leer.

La utilidad más importante de este método es para manejar índices negativos. Algo que no se puede con la notación de corchetes.

```
const nombres = ["Andres", "Valeria", "Ana", "Ramiro", "Richard"]

nombres.at(-1) // "Richard"
nombres[-1] // undefined
nombres.at(-3) // "Ana"
nombres[nombres.length -1] // "Richard"
```

**Cómo utilizar top level await:** Top level await permite utilizar la palabra reservada **await**, sin estar dentro de una función asíncrona con async. Sin embargo, únicamente se puede utilizar await en la parte superior del archivo de un módulo.

Esto puede servir para importaciones de manera dinámica o iniciar la conexión de tus bases de datos. Siempre y cuando respetes que debe estar en la parte encima del archivo de tipo módulo.

```
import fetch from "node-fetch";

const response = await fetch("https://api.escuelajs.co/api/v1/products");
const products = await response.json(); //Regresa un array de objetos con la información

export { products, response};
```