JAVASCRIPT CONTEMPORÁNEO

Charly Cimino

JavaScript contemporáneo

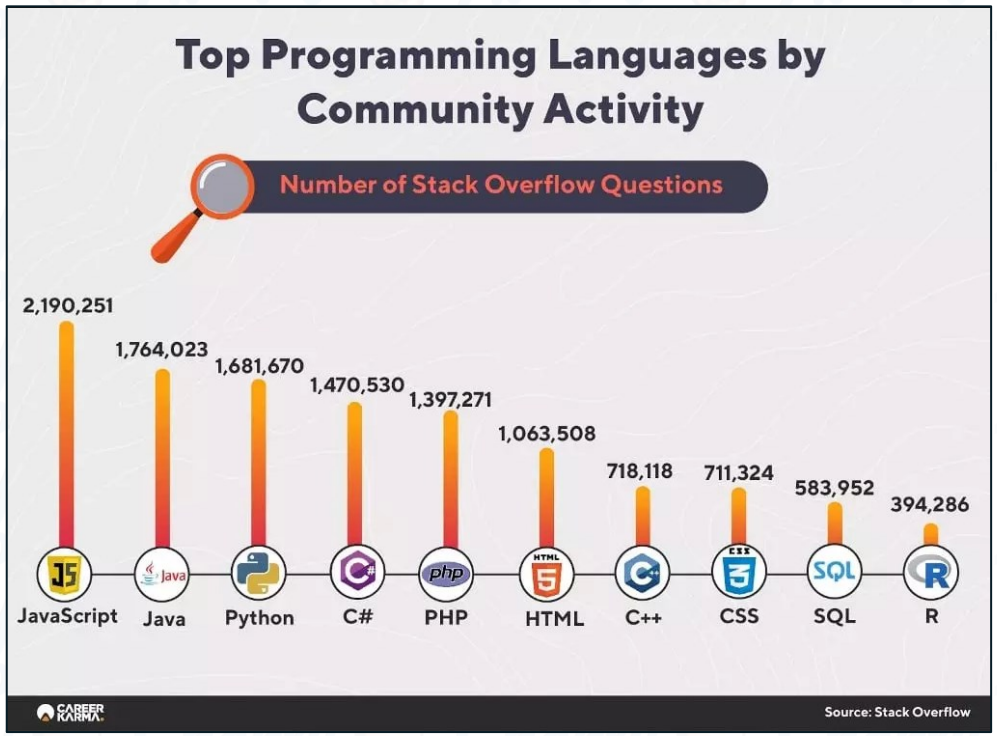
Charly Cimino

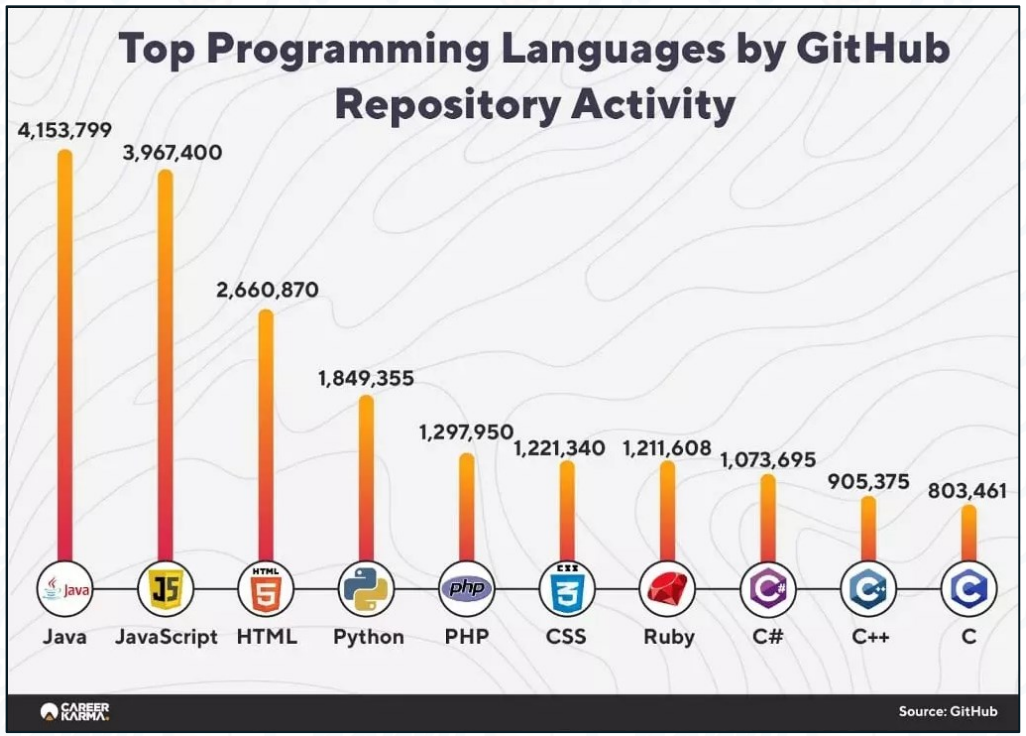
Versión 2023.08.06

JavaScript (JS)

Lenguaje de programación interpretado, multiparadigma y de tipado dinámico.

Tutorial de JS: https://www.w3schools.com/js/default.asp





Versiones relevantes de JS

Versión	Nombre Oficial	Descripción breve
ES1	ECMAScript 1 (1997)	Primera edición.
ES3	ECMAScript 3 (1999)	Añadidas expresiones regulares, try-catch, switch y do-while.
ES5	ECMAScript 5 (2009)	Añadido “strict mode”, soporte para JSON, Método **trim()** para String y métodos de orden superior para iterar arrays (**map()**, **reduce()**, **filter()**, entre otras).
ES6	ECMAScript 2015	Añade palabras **let** y **const**, funciones flecha, plantillas literales, valores por defecto en los parámetros, sintaxis rest / spread, desestructuración, entre otras novedades.
ES6	ECMAScript 2016	Añadido operador de exponenciación (**) y método **includes** para Arrays
ES6	ECMAScript 2017	Añade String padding, Object.entries, Object.values, funciones asíncronas y memoria compartida.
ES6	ECMAScript 2018	Añade sintaxis rest / spread para objetos literales, iteración asíncrona, método **finally()** para promesas y adiciones a las expresiones regulares.
ES6	ECMAScript 2019	Cambios en el método **sort()** y añade método **flat()** para Arrays
ES6	ECMAScript 2020	Añade tipo de dato BigInt, operador de encadenamiento opcional (**?**) y operador de coalescencia de nulls (**??**)
ES6	ECMAScript 2021	Añade método replaceAll() para String, método any() para promesas, operadores de asignación lógica, entre otras novedades.

https://en.wikipedia.org/wiki/ECMAScript#Versions

ES5 / ES6

El nuevo estándar EcmaScript 6 surgido en 2015 y posteriores revisiones introdujeron mejoras importantes en la sintaxis del lenguaje.

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Merece la pena conocer algunas de ellas en esta PPT.

Todas las características de ES6: http://es6-features.org/

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Declaración de variables

Se introduce la palabra **`let`** que permite declarar variables con un comportamiento más parecido a los lenguajes tradicionales, a diferencia de la palabra **`var`** y su hoisting.

```
var x = 5;
console.log(x); // 5

var x = 10;
console.log(x); // 10
```

ES5

```
let x = 5;
console.log(x); // 5

let x = 10;❌ Identifier 'x' has already been declared
console.log(x); // No se interpreta
```

ES6

Si bien la palabra **`var`** continúa presente dentro del nuevo estándar, se recomienda usar **`let`**.

https://www.w3schools.com/js/js_let.asp

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Declaración de constantes

Se introduce la palabra **const** que permite declarar constantes.

```
const PI = 3.14159265359;
```

✓

```
const PI = 3.14159265359;
PI = 4; ✗ Assignment to constant variable
```

✗

```
const PI; ✗ Missing initializer in const declaration
PI = 3.14159265359;
```

✗

```
const PI = 3.14159265359;
const PI = 4; ✗ Identifier 'PI' has already been declared
```

✗

Se recomienda el uso de **const** en lugar de **let** para alojar referencias a arrays, objetos, funciones y expresiones regulares.

https://www.w3schools.com/js/js_const.asp

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

El scope de let/const

A diferencia de las **var** que tienen scope a nivel global o de función, las **let** y **const** tienen scope de bloque.

```var x = 5;
console.log(x); // 5

if (true) {
    var x = 10;
    console.log(x); // 10
}

console.log(x); // 10
prueba();
console.log(x); // 10

function prueba() {
    var x = 30;
    console.log(x); // 30
}
```

var

```let x = 5;
console.log(x); // 5

if (true) {
    let x = 10;
    console.log(x); // 10
}

console.log(x); // 5
prueba();
console.log(x); // 5

function prueba() {
    let x = 30;
    console.log(x); // 30
}
```

let

```const x = 5;
console.log(x); // 5

if (true) {
    const x = 10;
    console.log(x); // 10
}

console.log(x); // 5
prueba();
console.log(x); // 5

function prueba() {
    const x = 30;
    console.log(x); // 30
}
```

const

https://www.w3schools.com/js/js_scope.asp

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Arrow functions

Las funciones flecha son la manera más concisa de escribir funciones, símil a las expresiones lambda.

```
const laFuncion = (param1, param2, …) => { … };
```

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

var restar = function(a, b) {
    return a - b;
}

console.log( sumar(8, 6) ); // 14
console.log( restar(8, 6) ); // 2
```

ES5

```
const sumar = (a, b) => { return a + b };
const restar = (a, b) => { return a - b };

console.log( sumar(8, 6) ); // 14
console.log( restar(8, 6) ); // 2
```

ES6

Se logra prescindir de la palabra **`function`**.

https://www.w3schools.com/js/js_arrow_function.asp

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Generalidades de las arrow functions

```
const sumar = (a, b) => { return a + b };
```

⟷

```
const sumar = (a, b) => a + b;
```

Si la función solo tiene una sentencia **`return`**, se puede omitir tanto ella como las llaves.

```
const doble = (x) => x * 2;
```

⟷

```
const doble = x => x * 2;
```

Si la función tiene un solo parámetro, pueden omitirse los paréntesis.

```
const constante5 = () => 5;
```

Si la función no recibe parámetros, de todas formas los paréntesis son obligatorios.

```
const saludar = () => { console.log("Hola") };
```

Si la función tiene más de una sentencia o no retorna valor, las llaves son obligatorias.

https://www.w3schools.com/js/js_arrow_function.asp

El scope de las arrow functions

El comportamiento del `this` en las funciones es diferente según cómo se implementen.

```
window.nombre = "Pepe";

function Persona (nom) {
    this.nombre = nom;
    setTimeout(function() { // Se define en Persona
        console.log(this); // Apunta al objeto window
        console.log(nombre); // "Pepe"
        console.log(this.nombre); // "Pepe"
    }, 100);
}

const p = new Persona("María"); // Se invoca desde window
```

Función con function: this apunta al objeto que invocó la función

```
window.nombre = "Pepe";

function Persona (nom) {
    this.nombre = nom;
    setTimeout(() => {// Se define en Persona
        console.log(this); // Apunta al objeto actual
        console.log(nombre); // "Pepe"
        console.log(this.nombre); // "María"
    }, 100);
}

const p = new Persona("María"); // Se invoca desde window
```

Función flecha: this apunta al objeto que definió la función

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions#this_y_funciones_flecha

El scope de las arrow functions

El comportamiento del `this` en los métodos es diferente según cómo se implementen.

```
const p = {
    nombre: "Pepe",
    saludar: function() { // Se define en p
        console.log(this); // Apunta al objeto actual
        console.log("Hola, soy "
                    + this.nombre); // "Hola, soy Pepe"
    }
}

p.saludar(); // Se invoca desde window
```

Método con function: this apunta al objeto que definió la función

```
const p = {
    nombre: "Pepe",
    saludar: () => { // Se define en p
        console.log(this); // Apunta al objeto window
        console.log("Hola, soy "
                    + this.nombre); // "Hola, soy undefined"
    }
}

p.saludar(); // Se invoca desde window
```

Método flecha: this apunta al objeto que invocó la función

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions#funciones_flecha_utilizadas_como_m%C3%A9todos

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Funciones con `function`

Usar la sintaxis más adecuada según el comportamiento deseado.

Funciones flecha

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Plantillas literales

Una nueva manera de construir cadenas literales utilizando back-sticks (``)

```
console.log("Esta es una cadena usando comillas");
```

Cadena con comillas

```
console.log(`Esta es una cadena usando back-sticks`);
```

Cadena con back-sticks

```
let nombre = "Pepe";
let anioNac = 1990;
let anioAct = new Date().getFullYear();
console.log("Hola, soy " + nombre);
console.log("Tengo " + (anioAct - anioNac) + " años");
```

Concatenación de expresiones dentro de cadenas con comillas

```
let nombre = "Pepe";
let anioNac = 1990;
let anioAct = new Date().getFullYear();
console.log(`Hola, soy ${nombre}`);
console.log(`Tengo ${anioAct - anioNac} años`);
```

Concatenación de expresiones dentro de cadenas con back-sticks

```
console.log("Esto va en una línea\nY esto va en otra");
```

```
console.log(`Esto va en una línea
Y esto va en otra`);
```

```
console.log("Título 1\tTítulo 2");
```

Uso de caracteres de escape para manipular la salida.

```
console.log(`Título 1   Título 2`);
```

La salida se interpreta literalmente como se escribe

https://www.w3schools.com/JS/js_string_templates.asp

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Parámetros predeterminados

Es posible tener funciones con valores por defecto en sus parámetros.

```
const laFuncion = (param1 = valorDefault, param2 = valorDefault, …) => { // Sentencias };
```

(Opc.) (Opc.)

```
function saludar(nombre = "extraño") {
    console.log(`Hola ${nombre}`);
}

saludar("Pepe"); // "Hola Pepe"
saludar(); // "Hola extraño"
```

```
const cociente = (a = 1, b) => a / b;

console.log( cociente(10, 4) ); // 2.5
console.log( cociente(undefined, 4) ); // 0.25
console.log( cociente(5) ); // NaN
console.log( cociente() ); // NaN
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Default_parameters

Parámetros Rest en funciones

El operador `...` delante del último parámetro permite obtener un número indefinido de valores en un array.

```
const laFuncion = (param1, param2, ...argsRestantes) => { // Sentencias };
```

```
function idiomas (i1, ...restantes) {
    console.log(i1);
    console.log(restantes);
}

idiomas("Español"); // "Español" []
idiomas("Español", "Inglés", "Alemán"); // "Español" ["Inglés", "Alemán"]
```

Sólo el último parámetro puede ser un "parámetro rest"

```
const sumatoria = (...numeros) => {
    return numeros.reduce((acu, act) => acu + act);
}

console.log( sumatoria(8, 4) ); // 12
console.log( sumatoria(2, 2, 3, 2, 15, 4) ); // 28
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/rest_parameters

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Sintaxis Spread en iterables

El operador `...` delante de un `String` o un array permite expandirlo en lugares donde se esperen cero o más argumentos (en invocaciones a funciones) o elementos (en arrays literales).

```
const sumatoria = (a, b, c) => a + b + c;
const numeros = [1, 2, 3];

console.log( sumatoria(numeros) ); // "1,2,3undefinedundefined"
console.log( sumatoria(...numeros) ); // 6
```

```
let colores = ["Rojo", "Azul", "Lila"];
colores = [...colores, "Verde"];

console.log(colores); // ["Rojo", "Azul", "Lila", "Verde"]
```

```
const nombre = "Juan";
const nombreArr = [...nombre];

console.log(nombreArr); // ["J", "u", "a", "n"]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Desestructuración de arrays

Es una sintaxis que nos permite asignar los valores de un array en variables por separado, en una sola expresión.

```
const colores = ["Rojo", "Azul", "Lila"];

let [a, b, c] = colores;

console.log(a); // "Rojo"
console.log(b); // "Azul"
console.log(c); // "Lila"
```

Asignación de variables declaradas en el momento

```
let a = "Rojo";
let b = "Azul";

[a , b] = [b , a];

console.log(a); // "Azul"
console.log(b); // "Rojo"
```

Intercambio de variables

```
const colores = ["Rojo", "Azul", "Lila"];
let a, b;

[a , , b] = colores;

console.log(a); // "Rojo"
console.log(b); // "Lila"
```

Ignorar algunos valores

```
const colores = ["Rojo", "Azul", "Lila"];
let a, b, c;

[a, b, c] = colores;

console.log(a); // "Rojo"
console.log(b); // "Azul"
console.log(c); // "Lila"
```

Asignación de variables declaradas previamente

```
const colores = ["Rojo", "Azul"];
let a, b, c;

[a, b, c = "Negro"] = colores;

console.log(a); // "Rojo"
console.log(b); // "Azul"
console.log(c); // "Negro"
```

Valores por defecto si alguno es `undefined`

```
const colores = ["Rojo", "Azul", "Lila"];
let a, resto;

[a, ...resto] = colores;

console.log(a); // "Rojo"
console.log(resto); // ["Azul", "Lila"]
```

Valores restantes a una variable usando spread

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#desestructuraci%C3%B3n_de_arreglos

Desestructuración de objetos

Es una sintaxis que nos permite asignar las propiedades de un objeto en variables por separado, en una sola expresión.

```
const p = { nombre: "Pepe", edad: 30 };

let {nombre, edad} = p;

console.log(nombre); // "Pepe"
console.log(edad); // 30
```

Asignación de variables declaradas en el momento

```
const p = { nombre: "Pepe", edad: 30 };

const {nombre: fname, edad: age} = p;

console.log(fname); // "Pepe"
console.log(age); // 30
```

Asignación de variables con otros nombres

```
const p = { nombre: "Pepe", edad: 30 };
let nombre, edad;

({nombre, edad} = p); // Paréntesis obligatorios

console.log(nombre); // "Pepe"
console.log(edad); // 30
```

Asignación de variables declaradas previamente

```
const p = { nombre: "Pepe" };
let nombre, edad;

({nombre, edad = 18} = p);

console.log(nombre); // "Pepe"
console.log(edad); // 18
```

Valores por defecto si alguno es `undefined`

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#desestructuraci%C3%B3n_de_objetos

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Desestructuración de objetos en parámetros de funciones

```
const p = { nombre: "Pepe", edad: 30 };

const presentar = ({nombre, edad}) => {
    console.log(`Es ${nombre} y tiene ${edad}`);
}

presentar(p); // "Es Pepe y tiene 30"
```

Desestructuración de las propiedades del objeto argumento en parámetros

```
const p = { edad: 30 };

const presentar = ({nombre = "desconocido", edad}) => {
    console.log(`Es ${nombre} y tiene ${edad}`);
}

presentar(p); // "Es desconocido y tiene 30"
```

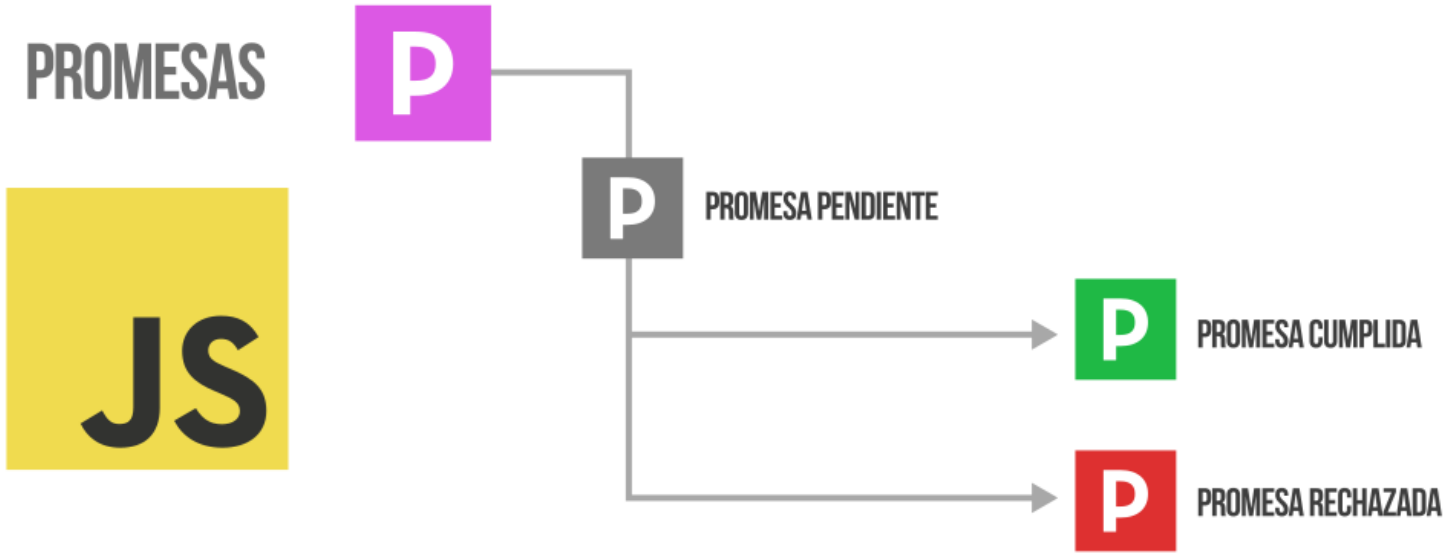
Parámetro por defecto si la respectiva propiedad es `undefined`

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#desestructuraci%C3%B3n_de_objetos

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Promesas

Cubiertas en la PPT [Asincronismo en JavaScript](#).



https://lenguajejs.com/javascript/asincronia/promesas/
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise

Operador de exponenciación

El operador ** permite elevar el primero operando al segundo operando.

```
let x = Math.pow(2, 5);
console.log(x); // 32
```

```
let x = 2 ** 5;
console.log(x); // 32
```

El operador **= permite elevar el valor de una variable al valor de la derecha y actualizarla.

```
let x = 4;
x = x * x;
console.log(16); // 16
```

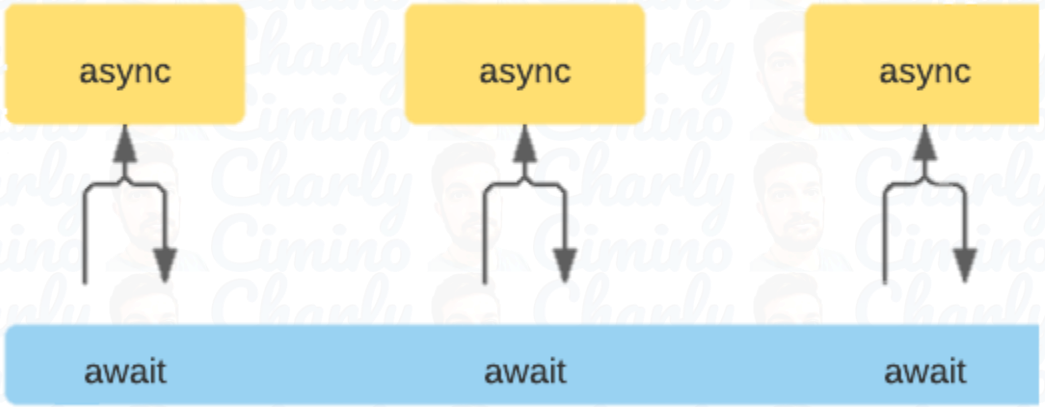
```
let x = 4;
x **= 2;
console.log(x); // 16
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Exponentiation

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Async / await

Cubiertas en la PPT [Asincronismo en JavaScript](#).



https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/await

Sintaxis Spread en objetos literales

El operador `...` delante de un objeto literal dentro de otro permite copiar sus propiedades.

```
const color = { r: 43, g: 35, b: 21 };
const colorAlpha = { ...color, a: 0.5};

console.log(colorAlpha); // {r: 43, g: 35, b: 21, a: 0.5}
```

```
const color = { r: 43, g: 35, b: 21 };

console.log({ ...color, r: 0 }); // {r: 0, g: 35, b: 21}
console.log({ r: 0, ...color }); // {r: 43, g: 35, b: 21}
```

```
const punto2D = { x: 2, y: 3 };
const prof = { z: 4 }

const punto3D = {...punto2D, ...prof}

console.log(punto3D); // {x: 2, y: 3, z: 4}
```

```
const color = { r: 43, g: 35, b: 21, a: 0.5 };

let {a, ...resto} = color;

console.log(a); // 0.5
console.log(resto); // { r: 43, g: 35, b: 21 }
```

```
const p = { nombre: "Pepe", edad: 30 };
const clonP = {...p};
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Método flat en Arrays

Cubierto en la PPT Arrays en JavaScript.

```
[1,2,[3,4,[5,6]]]
```

```
[1, 2, 3, 4, 5, 6]
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/flat

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Operador de encadenamiento opcional

El operador `?` funciona de manera similar al . a la hora de acceder a las propiedades de un objeto, pero sin causar errores en caso de que alguna referencia sea `null` o `undefined`.

```
const auto = {
    marca: 'Acme',
    motor: {
        numero: '654116'
    }
};

console.log( auto.motor.numero ); // 654116
console.log( auto.motor.cilindrada ); // undefined

console.log( auto.chasis.numero ); //  ✖ Cannot read property 'numero' of undefined
console.log( auto.chasis?.numero ); // undefined

console.log( auto.acelerar() ); //  ✖ auto.acelerar is not a function
console.log( auto.acelerar?.() ); // undefined
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Optional_chaining

ES6 2015 → ES6 2016 → ES6 2017 → ES6 2018 → ES6 2019 → ES6 2020 → ES6 2021

Operador de coalescencia de nulls

El operador `??` regresa el valor del primer operando si no es `null` o `undefined`, de lo contrario, retorna el segundo.

```
console.log( "string1" ?? "string2" ) // "string1"
console.log( undefined ?? "string" ) // "string"
console.log( null ?? "string" ) // "string"
console.log( false ?? "string" ) // false
console.log( NaN ?? "string" ) // NaN
console.log( "" ?? "string" ) // ""
```

Anteriormente se utilizaba el operador lógico `||`, que a diferencia de `??`, no devuelve valores *falsy*.

```
console.log( "string1" || "string2" ) // "string1"
console.log( undefined || "string" ) // "string"
console.log( null || "string" ) // "string"
console.log( false || "string" ) // "string"
console.log( NaN || "string" ) // "string"
console.log( "" || "string" ) // "string"
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_Coalescing_Operator

Método replaceAll para String

El método `replaceAll` permite reemplazar todas las ocurrencias de una cadena o expresión regular por otra.

```
const cad = 'Mi mamá me mima';

console.log(cad.replaceAll("M", "L")); // "Li mamá me mima"
console.log(cad.replaceAll("m", "l")); // "Mi lalá le lila"

const regex = /M/ig;
console.log(cad.replaceAll(regex, "l")); // "li lalá le lila"
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replaceAll

FIN DE LA PRESENTACIÓN

Encontrá más como estas en mi sitio web.