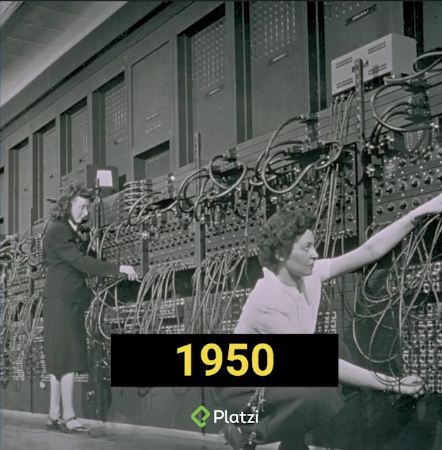
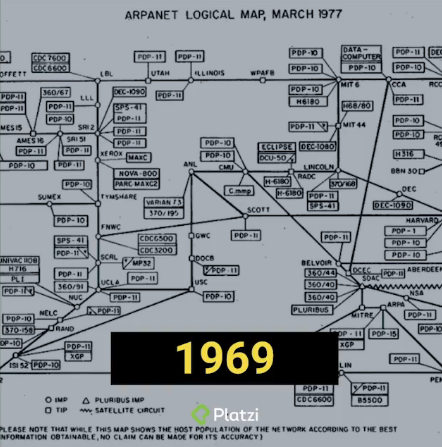
# **Historia de JavaScript: ¿qué es ECMAScript?**

## **Historia**

* 1950 inicio de las computadoras.



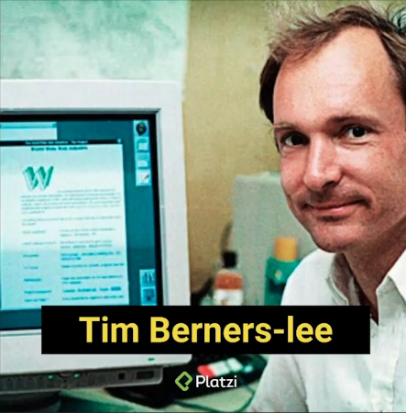
* 1969 se creó la red **ARPANET**, que era la forma en que podían comunicarse dos computadoras para compartir información.



* 1970-1990 Se explora **ARPANET**



* 1990 **Tim Berners Lee** dio pauta al internet y se conoció el termino **World Wide Web** (forma de acceder a archivos, documentos enlaces al alcance de todos)



* 1993 Nacimiento de un Navegador.

Universitarios pioneros escribieron un software simple que inicio una revolución.

* **Marc Andreessen**: Le dio vida a una gran empresa.
* **Netscape**: Tenia un navegador que podías acceder a internet (HTML simple, enlaces, algunas fotografías)

**Browser wars**: Guerra de los navegadores.

Microsoft en 1995 lanzó Internet Explorer, en la guerra nacieron tecnologías como CSS, conceptos como la empresa Mozilla y JS.

**JS**: Apareció en 4 de diciembre de 1995 Diseñado por **Netscape Communications**, Fundación Mozilla.

* Su primer nombre fue **Mocha**
* Luego **LiveScript**
* Finalmente, JavaScript.

Microsoft no se podía quedar atrás y lanzo JScript y por eso tenemos ECMAScript (1997) permitió un estándar (serie de reglas que va a tener los lenguajes de programación).

Evolución de ECMAScript:



# **¿Qué es el TC39?**

Es un grupo de hackers, académicos y/o personas afines a la tecnología que se encargan de las actualizaciones, bajo el mando de **ECMA**.

**ESNext** es un nombre que siempre indica la próxima versión de javascript y esto consta de los siguientes pasos:

* Stage0: **Strawperson** (borrador, cualquier persona puede tener una idea para implementar en el estándar)
* Stage1: **Proposal** (propuesta formal)
* Stage2: **Draft** (borrador, como va a funcionar la implementación el impacto entre otros)
* Stage3: **Candidate** (se elige el candidato, vamos a tener una propuesta que va ayudar a mejorar el lenguaje.)
* Stage4: **Finished** (va a ser desplegada en la version normalmente en JUNIO)

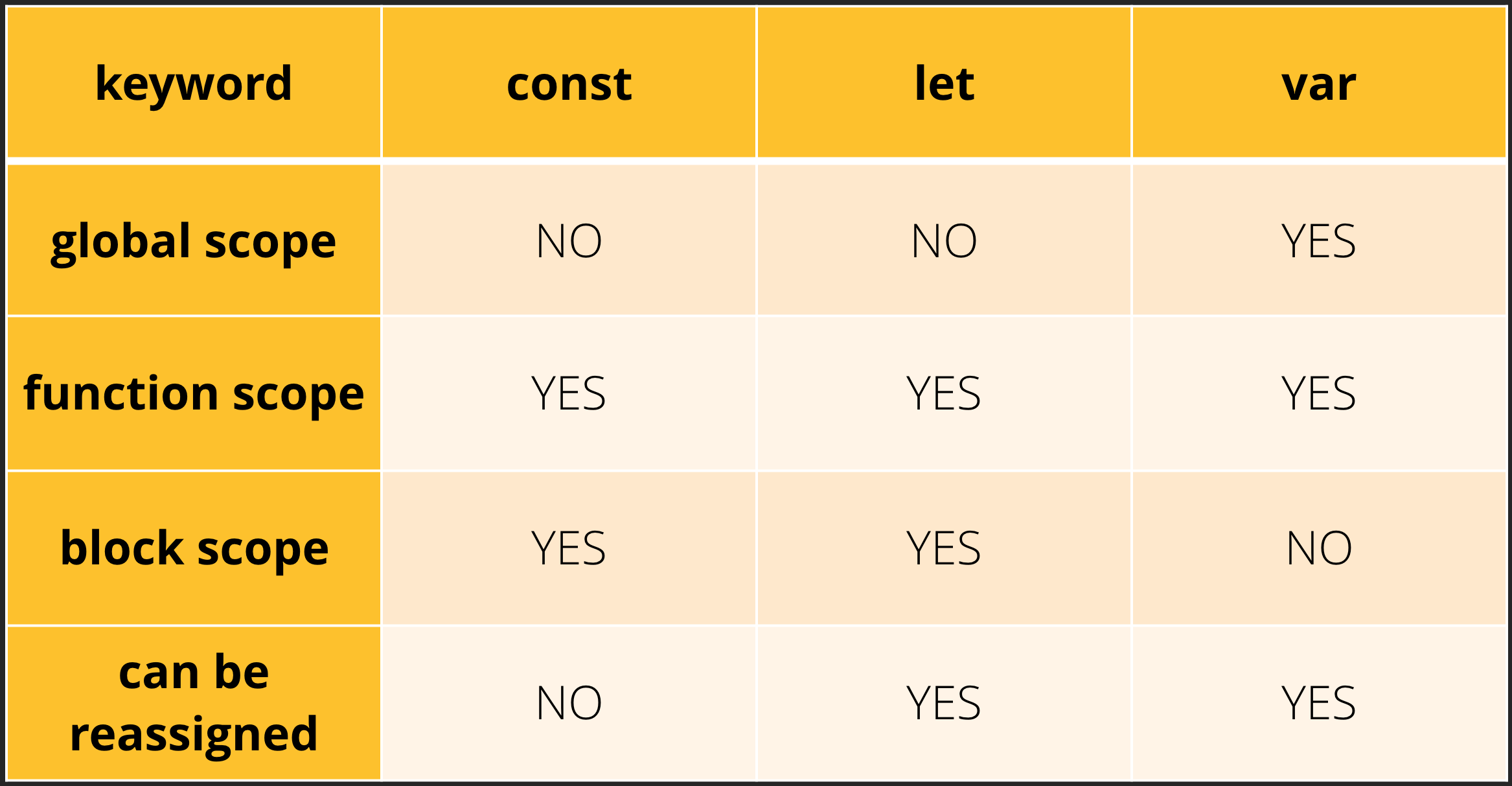
# **ES6: let y const, y arrow functions**

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**.

## **La nueva forma para declarar variables con let y const**

Hasta ahora aprendiste a declarar variables con **var**, sin embargo, a partir de la especificación de ES6 se agregaron nuevas formas para la declaración de variables.

Las nuevas palabras reservadas **let** y **const** resuelven varios problemas con **var** como el **scope**, **hoisting**, **variables globales**, redeclaración y reasignación de variables.



## **Variables redeclarada y reasignadas**

La **redeclaración** es volver a declarar una variable, y la **reasignación** es volver a asignar un valor. Entonces cada palabra reservada tiene una forma diferente de manejar variables:

* Una variable declarada con **var** puede ser redeclarada y reasignada.
* Una variable declarada con **let** puede ser reasignada, pero no redeclarada.
* Una variable declarada con **const** no puede ser redeclarada, ni reasignada. Su declaración y asignación debe ser en una línea, caso contrario habrá un error.

En conclusión, si intentas redeclarar una variable declarada con **let** y **const** habrá un error de “**variable ya declarada**” por otro lado, si intentas reasignar una variable declarada con **const** existirá un “**error de tipo**”.

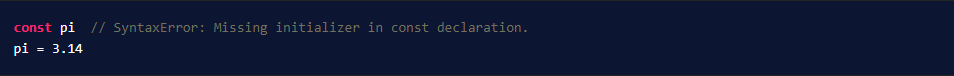
En los demás casos, JavaScript lo aceptará como válidos, algo problemático con **var**, por eso deja de utilizarlo.

## **Ejemplo de declaración y asignación en diferentes líneas**

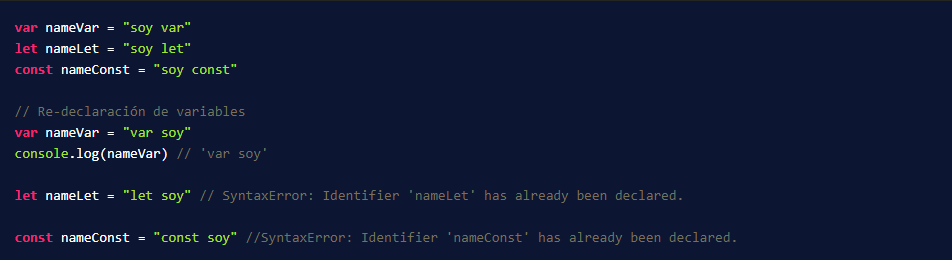


Aunque realmente lo que pasa si no asignas un valor en la declaración, JavaScript le asigna un valor **undefined**.

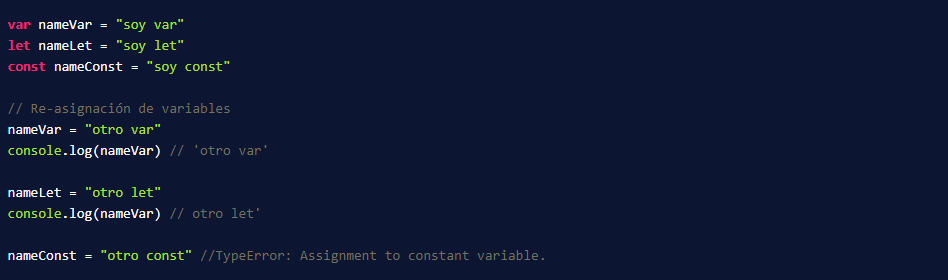
Ejemplo de declarar y asignar con **const** en diferentes líneas de código



Ejemplo de redeclaración de variables



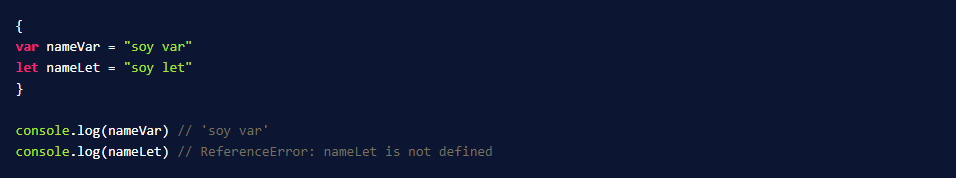
Ejemplo de reasignación de variables



Ten en cuenta que los errores pararán la ejecución de tu programa.

## **Scope**

En el tema del scope, **let** y **const** tienen un **scope de bloque** y **var** no.



## **Objeto global**

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 redeclaración de variables.

## **Funciones flecha**

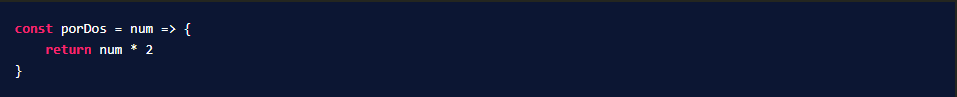
Las funciones flecha (arrow functions) consiste en una función anónima con la siguiente estructura:



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

## **Omitir paréntesis en las funciones flecha**

Si existe un solo parámetro, puedes omitir los paréntesis.

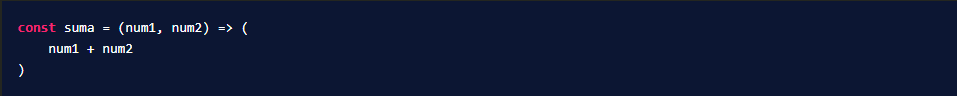


## **Retorno implícito**

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**.



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

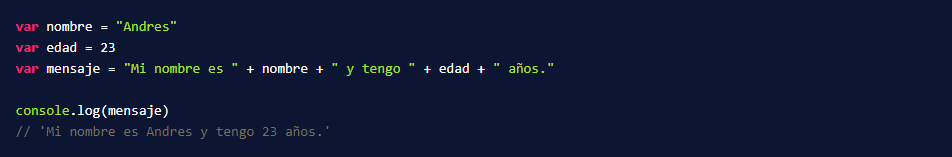


# **Strings**

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.

## **Concatenación de caracteres**

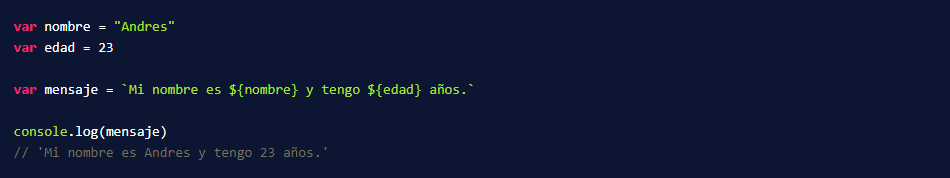
Antes de ES6, si querías crear una cadena larga o un mensaje elaborado, debías utilizar la concatenación. La concatenación de caracteres consiste en unir uno o varios caracteres, como si fuera una suma.



Esto trae varios **problemas en la legibilidad y mantenibilidad del código**. Se convierte cada vez más complejo en mensajes más extensos o el estar pendiente de agregar espacios antes o después de cada variable concatenada.

## **Cómo utilizar las plantillas literales**

Las plantillas literales añadidas en ES6, se emplea el carácter acento grave ( ` ), que no es una comilla simple ( ’ ), para envolver el mensaje. Para incluir las variables se utiliza la sintaxis **${variable}**.



De esta manera el código es más legible y que pueda mantenerse.

## **Plantilla multilínea**

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



Con ES6 solamente necesitas utilizar las plantillas literales.



# **Parametros por defecto**

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.

## **Cómo era utilizar valores por defecto antes de ES6**

Tal como puedes ver en el siguiente código, la función **sumar** recibe dos parámetros y retorna el valor total. Sin embargo, si alguien no decide poner alguno o todos los parámetros necesarios, pues que el programa no funcionará correctamente.



Antes de ES6, se debía establecer una variable y utilizar el operador OR ( ||) con el valor por defecto necesario. El carácter guion bajo (\_) lo utilizo para diferenciar el parámetro de la función de la variable declarada.



## **Cómo utilizar los parámetros por defecto**

Con los parámetros por defectos añadidos en ES6, eliminamos las declaraciones para mejorar la legibilidad y el mantenimiento del código de la siguiente manera:



Puedes utilizar cualquier valor, siempre que sea necesario.

## **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.



# **Asignación de desestructuración**

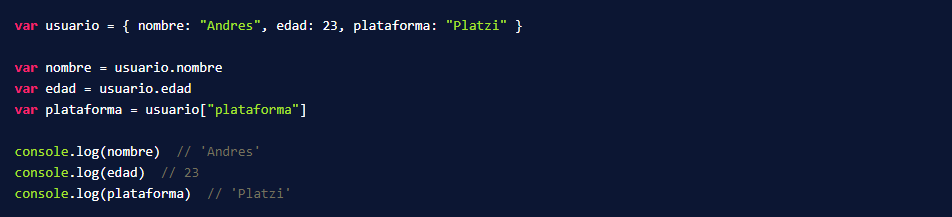
La desestructuración (**destructuring**) consiste en extraer los valores de arrays o propiedades de objetos en distintas variables.

## **Desestructuración de objetos**

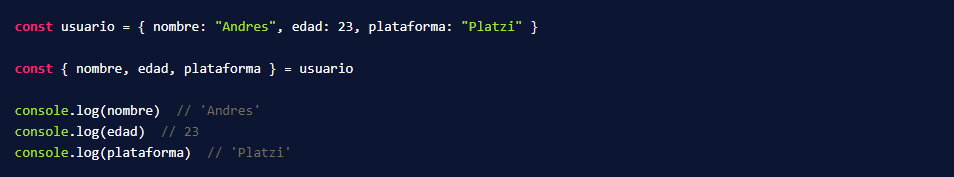
La desestructuración de objetos implica extraer las propiedades de un objeto en variables. Mediante el mismo nombre de la propiedad del objeto con la siguiente sintaxis:



Antes de ES6, necesitabas acceder al objeto con la notación punto o corchetes por cada propiedad que se necesita y asignar ese valor a una variable diferente.

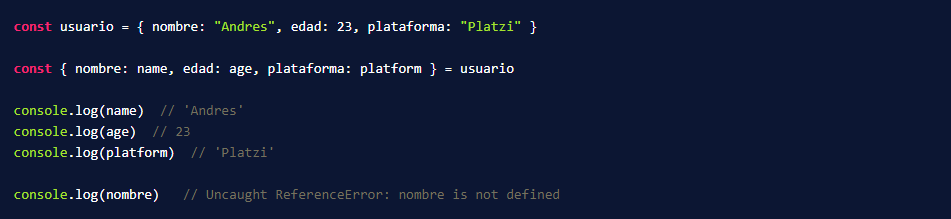


Con la desestructuración puedes realizar lo mismo, pero en una sola línea, provocando que el código seas más legible y mantenible.



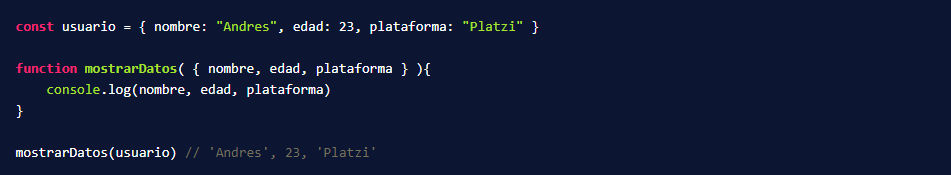
## **Cambiar el nombre de las variables con desestructuración**

Si no te agrada el nombre de la propiedad del objeto, puedes cambiarlo utilizando la siguiente sintaxis:



## **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.



## **Desestructuración de arrays**

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.



## **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.



## **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.

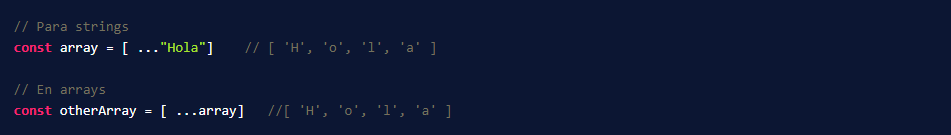


Como los arrays son un tipo de objeto, puedes utilizar la desestructuración de objetos mediante el índice y utilizando un nombre para la variable.



# **Spread Operator**

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.



También se utiliza para objetos, pero esta característica fue añadida en versiones posteriores de **ECMAScript** y es denominada **propiedades de propagación**.

## **Cómo copiar arrays utilizando el operador de propagación**

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**.

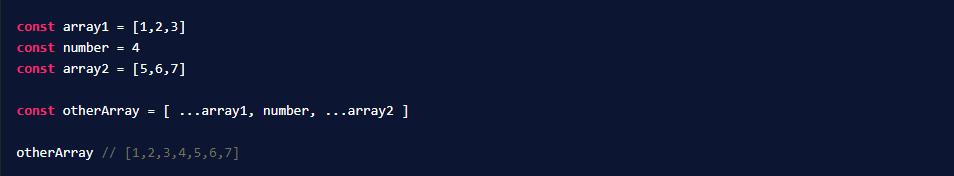


Para evitar esto, utiliza el **operador de propagación** para crear una copia del array que utilice una **referencia en memoria diferente al original**.



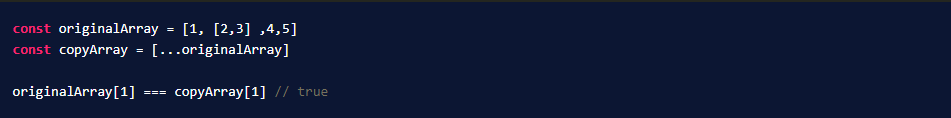
## **Cómo unir arrays y añadir elementos con el operador de propagación**

Para **unir** dos arrays con el operador de propagación, simplemente debes separarlos por comas en un array.



## **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 subelementos en cada nivel, **tendrán la misma referencia de memoria** en la **copia** **y en el** **original**.



La manera de solucionar es más compleja, tendrías que emplear el operador de propagación para cada elemento en cada nivel de profundidad.

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.

## **Parámetro rest**

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.

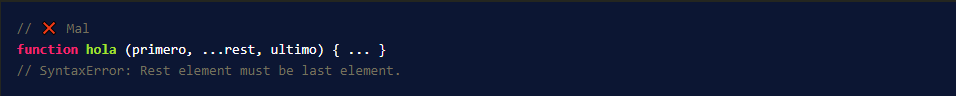


También sirve para obtener los elementos restantes de un array u objeto usando desestructuración.



## **Posición del parámetro rest**

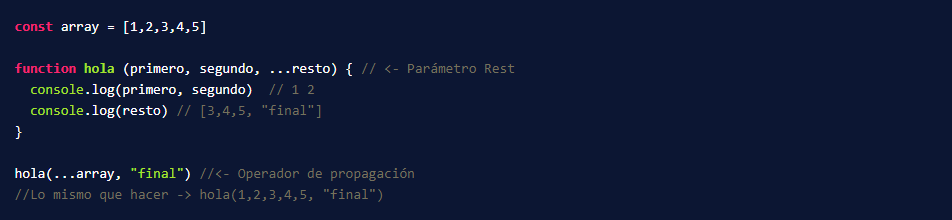
El parámetro rest **siempre** deberá estar en la **última posición** de los parámetros de la función, caso contrario existirá un error de sintaxis.



## **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**, mientras que el operador de propagación **expande los elementos de un iterable** en un array y no importa en qué lugar esté situado.



# **Object literals**

Los **objetos literales** consisten en crear objetos a partir de variables **sin repetir el nombre**. Antes de ES6, para crear un objeto a partir de variables consistía en la siguiente manera:

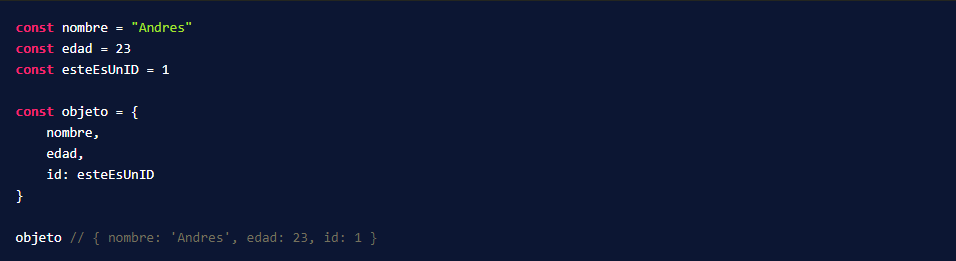


## **Cómo utilizar objetos literales**

Con los parámetros de objeto puedes obviar la repetición de nombres, JavaScript creará la propiedad a partir del nombre de la variable con su respectivo valor.



El resultado es el mismo, pero sin la necesidad de repetir palabras. Puedes combinarlo con variables que su propiedad tiene un nombre diferente.

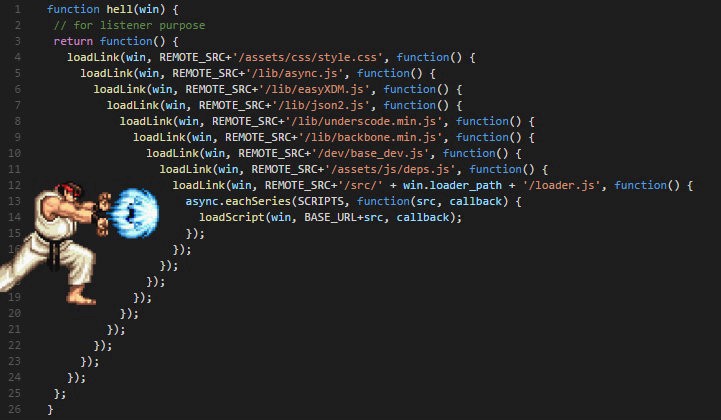


# **Promesas**

Una **promesa** es una forma de manejar el asincronismo en JavaScript y se representa como un objeto que puede generar un valor único a futuro, que tiene dos estados, o está **resuelta** o incluye una razón por la cual **no ha sido resuelta** la solicitud.

## **Cómo utilizar las promesas**

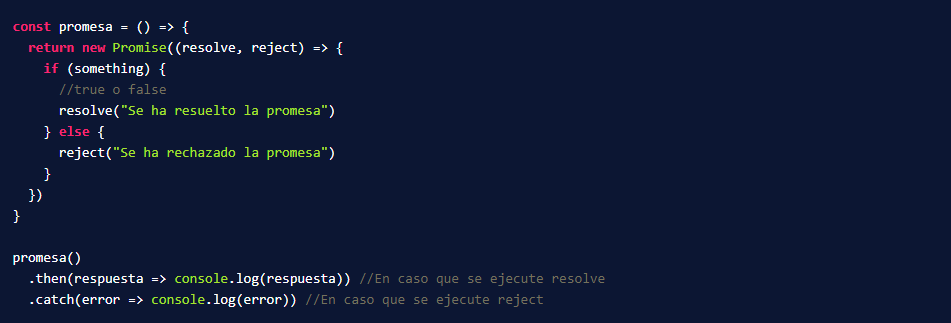
Solamente ten presente que la clase **Promise** y sus métodos **then** y **catch** fueron añadidos en ES6. Esto resuelve un problema del manejo del asincronismo con callbacks, llamado **Callback Hell**.



El argumento de la clase **Promise** es una función que recibe dos parámetros:

* **resolve**: Cuando la promesa es resuelta.
* **reject**: Cuando la promesa es rechazada.

Puedes utilizar cualquier nombre, siempre y cuando sepas su funcionamiento.



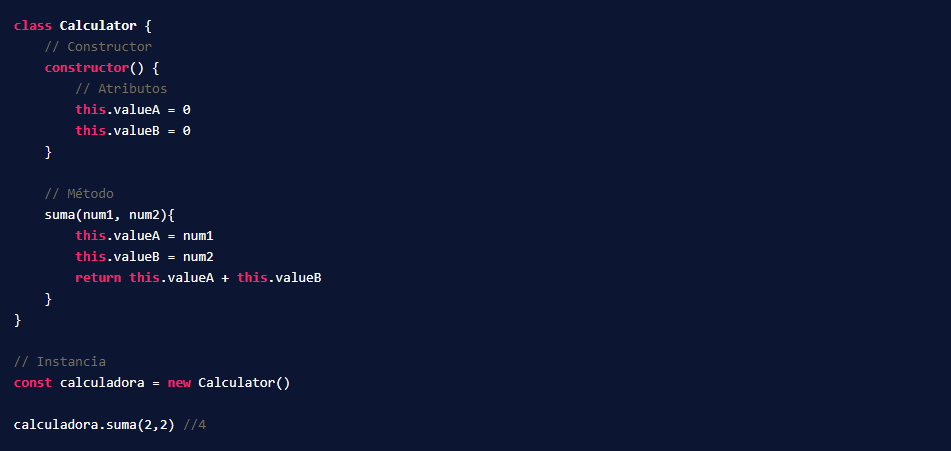
# **Clases**

La palabra reservada **class** es una forma para crear clases y manejar la herencia en JavaScript, permitiendo resolver problemas con el paradigma de programación orientada a objetos (POO).

## **Estructura de las clases en JavaScript**

La estructura de clases en JavaScript consiste en:

* Definir la clase con la palabra reservada **class**, seguido del nombre (por convención se empieza con mayúscula).
* La función constructora sirve para crear las variables necesarias en la instancia del objeto, a partir de los argumentos en la instancia.
* Para definir atributos necesitas el contexto **this**, que representa la instancia del objeto.
* Métodos para definir las acciones de la clase.
* Para crear una instancia, deberás declarar una variable e invocar la clase con la palabra reservada **new**.



# **Module**

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**

Para explicar cómo funciona las exportaciones e importaciones de código, 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:



## **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, antes de declarar la funcionalidad, o entre llaves **{}**.

Por ejemplo, en el archivo **math\_function.js** declaramos una función para sumar dos valores, el cual lo exportaremos.



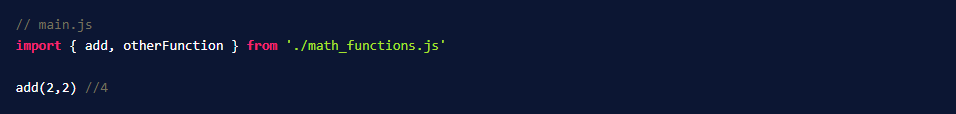


## **Qué son las importaciones de código**

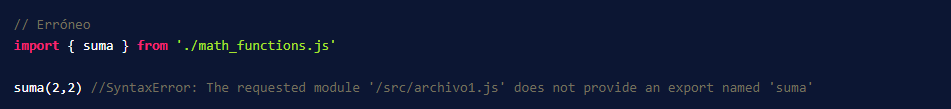
Las importaciones de código consisten en **usar funciones o variables de otros archivos** mediante la palabra reservada **import**, que deberán estar siempre lo más arriba del archivo y utilizando el mismo nombre que el archivo original.

Existen dos formas de exportar, antes de declarar la funcionalidad, o entre llaves **{}**.

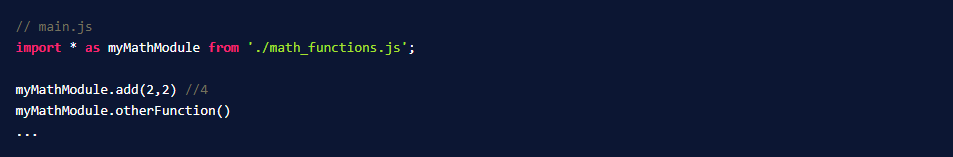
Por ejemplo, importamos la función **add** del archivo **math\_function.js** para utilizarla en un archivo **main.js**.



Si importamos el módulo con un nombre diferente, existirá un error de sintaxis.



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**.



## **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.



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



## **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.

## **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.



# **Generator**

Los **generadores** son funciones especiales que pueden pausar su ejecución, luego volver al punto donde se quedaron, recordando su scope y seguir retornando valores.

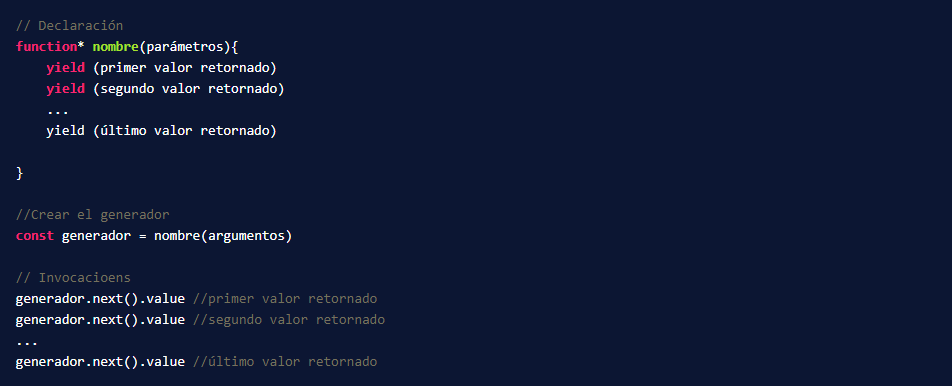
Estos se utilizan para guardar la totalidad de datos infinitos, a través de una función matemática a valores futuros. De esta manera ocupan poca memoria, con respecto a si creamos un *array* u *objeto*.

## **Cómo utilizar generadores**

La sintaxis de los generadores comprende lo siguiente:

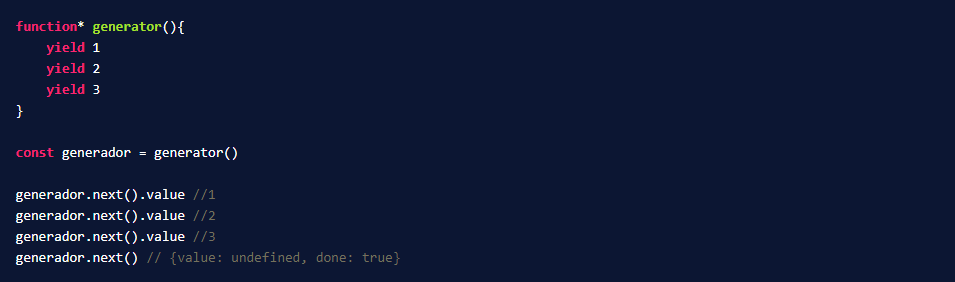
* La palabra reservada **function\*** (con el asterisco al final).
* La palabra reservada **yield** que hace referencia al valor retornado cada vez que se invoque, recordando el valor anterior.
* Crear una variable a partir de la función generadora.
* El método **next** devuelve un objeto que contiene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

Si el generador se lo invoca y ha retornado todos sus valores de **yield**, entonces devolverá el objeto con las propiedades **value** con **undefined** y un **done** con **true**.



## **Ejemplo de un generador**

Por ejemplo, creemos un generador para retornar tres valores.



* [Documentación de generadores](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Generator)

## **Cómo utilizar for of y for in**

Existen dos nuevas formas de utilizar ciclos repetitivos. El bucle **for** **valor** **of iterable** *recorre iterables*, como **arrays**, **Map**, **Set** e incluso un **generador**.

El valor es cada elemento del iterable puede tener cualquier nombre, por eso se inicia con **let nombre**.



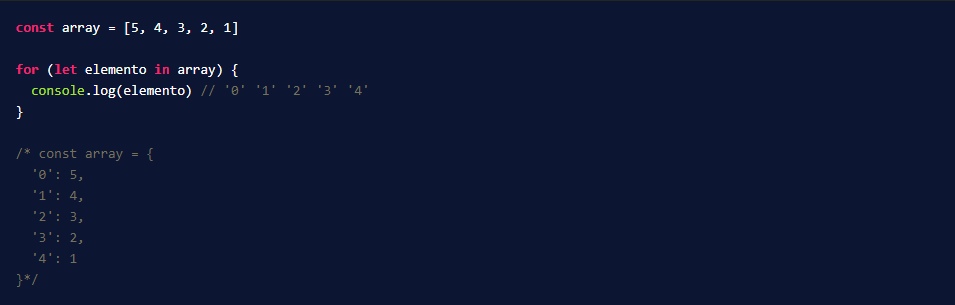
Sin embargo, debes tener en cuenta que solo podrás acceder a sus valores, y no a sus referencias, por lo que si quieres cambiar los elementos del array, necesitarás un índice **array[indice]**.



Si intentas recorrer un objeto de esta forma **for elemento of objeto**, te ocurrirá un error, porque **un objeto no es un iterable**. En su lugar puedes utilizar **for elemento in objeto**, que recorrerá las propiedades del objeto.



Sin embargo, si utilizas **for elemento in array**, no dará un error, pero el resultado no será el esperado, ya que los arrays **son un tipo de objeto donde cada propiedad es el índice del valor del array o del iterable**. Por lo que debes tener cuidado.



# **set-add**

**Set** es una nueva estructura de datos para almacenar **elementos únicos**, es decir, sin elementos repetidos.

## **Cómo utilizar los Sets**

Para iniciar un **Set**, se debe crear una instancia de su clase a partir de un iterable. Generalmente, un iterable es un array.



## **Cómo manipular los Sets**

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: exponentiation operator y array includes**

La siguiente versión de **ECMAScript** fue publicada en 2016. Las siguientes características de ES7 o ES2016 que aprenderás son: el método **includes** de *arrays* y el **operador de potenciación**.

## **Operador de potenciación**

El operador de potenciación (exponential operator) consiste en elevar una base a un exponente utilizando el doble asterisco (\*\*).

**base \*\* exponente**

Por ejemplo, el cubo de 2 es igual a 8, matemáticamente expresado sería: **2^3=8**.



## **Método includes**

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:

* El **elemento** a comparar.
* El **índice inicial** desde donde comparar hasta el último elemento.

## **Índices positivos y negativos**

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



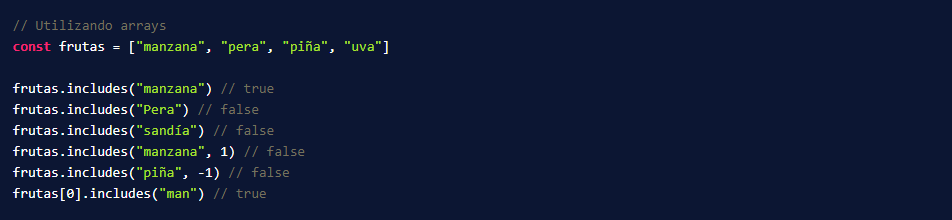
Los índices negativos comienzan desde -1 hasta el negativo de la longitud total del array, de **derecha** a **izquierda**.



## **Ejemplos utilizando el método includes**

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





# **ES8: object entries y object values**

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

## **Obtener los pares de valor de un objeto en un array**

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



## **Obtener las propiedades de un objeto en un array**

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



## **Obtener los valores de un objeto en un array**

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



# **string padding y trailing commas**

## **Rellenar un string o padding**

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.

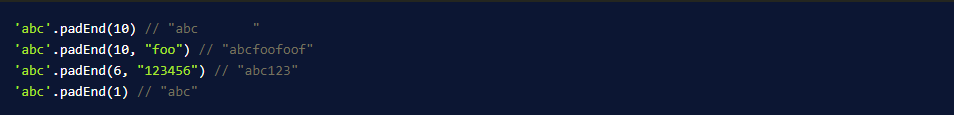
## **Método padStart**

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



## **Método padEnd**

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



## **Trailing commas**

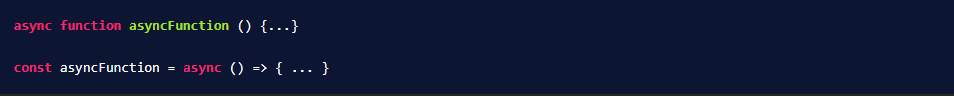
Las **trailing commas** consisten en comas al final de objetos o arrays que faciliten añadir nuevos elementos y evitar errores de sintaxis.



# **funciones asíncronas**

## **Cómo utilizar funciones asíncronas**

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



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**.



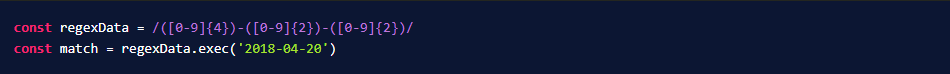
## **¿Cuál es la mejor forma de manejar promesas, then o async/await?**

Ambas son muy útiles, manejar ambas te hará un mejor desarrollador.

# **ES9: expresiones regulares**

Las expresiones regulares o **RegEx** (regular expresions) 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.



## **Propiedades de propagación**

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.



## **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.

De esta manera el segundo objeto tendrá una referencia en memoria diferente al original.



## **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**.



La manera de solucionar esto es más compleja, tendrías que utilizar el operador de propagación para cada elemento en cada nivel de profundidad.

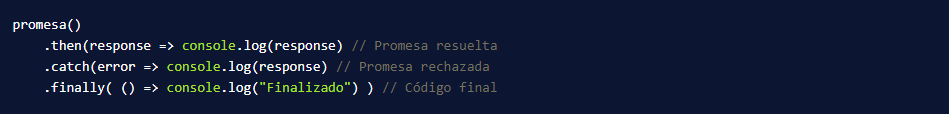
Sin embargo, recientemente salió una forma de crear una copia profunda con [**StructuredClone**](https://developer.mozilla.org/en-US/docs/Web/API/structuredClone). Aunque es una característica muy reciente, así que revisa que [navegadores tienen soporte](https://caniuse.com/?search=structuredClone).



# **Promise.finally**

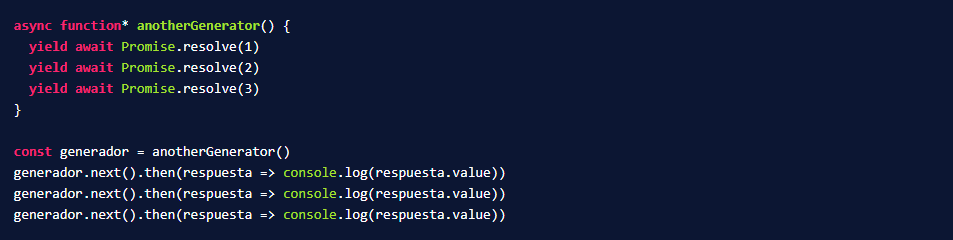
## **Método finally en promesas**

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



## **Generadores asíncronos**

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



## **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**.

El valor es cada elemento del iterable puede tener cualquier nombre, por eso se inicia con **let** **nombre**.



# **ES10: flat-map y trimStart-trimEnd**

## **Qué es el aplanamiento de arrays**

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

## **Método flat**

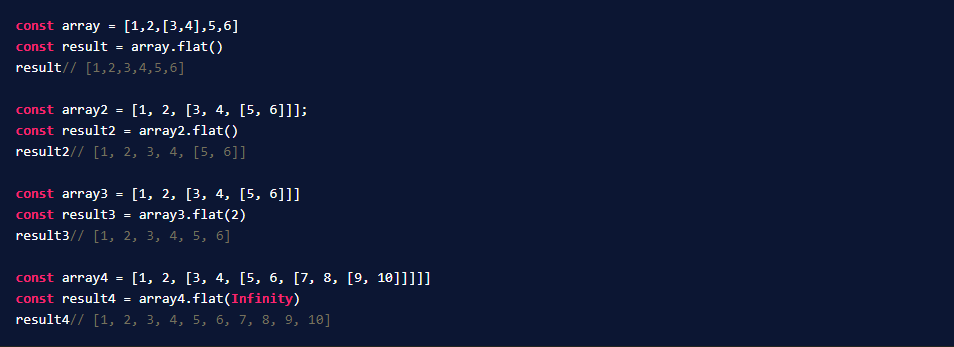
El método **flat** devuelve un *array* donde los sub-arrays han sido propagados hasta una profundidad especificada.

Este método es **inmutable**, 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**.

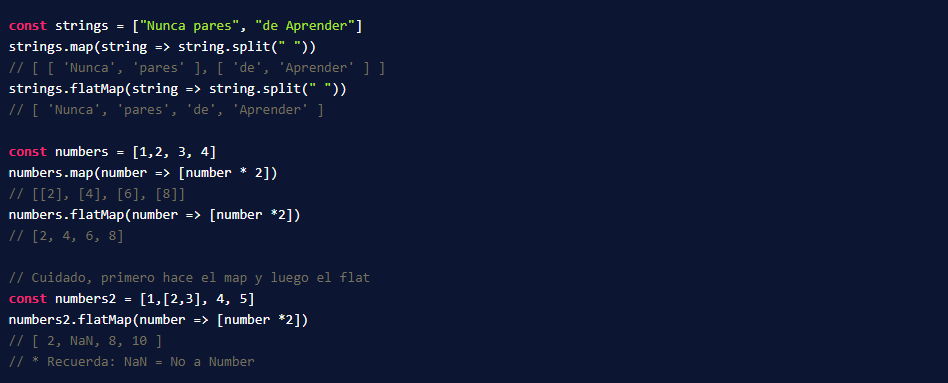


## **Método flatMap**

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 aplana en una sola profundidad (como si fuera **flat**).

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

Este método recibe los mismos argumentos que el método **map**.



## **Eliminar espacios en blanco de un string**

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**.



# **try catch y fromEntries**

## **Parámetro opcional de catch**

El parámetro opcional de catch permite omitir el error si es necesario.

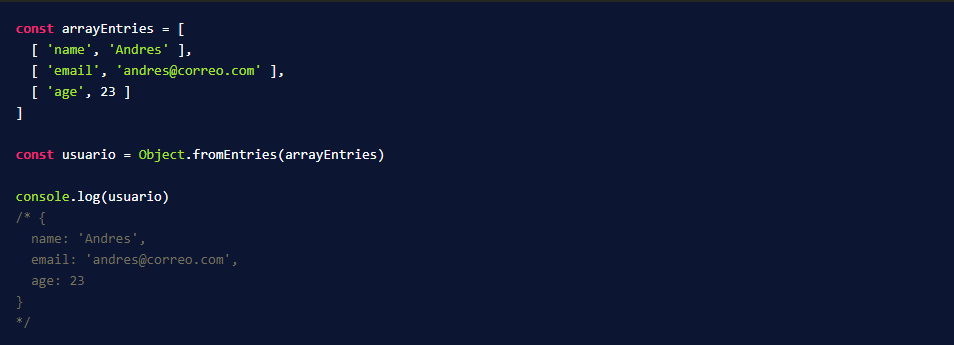


Aunque siempre es recomendable manejar el error como parámetro, ya que tiene más información del problema.

## **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**].

Se considera la operación inversa de **Object.entries**().

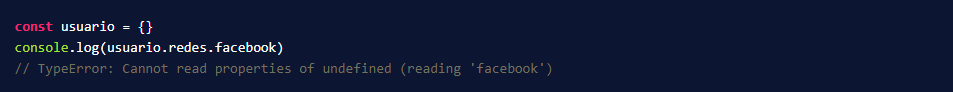


# **ES11: optional chaining**

Cuando intentas acceder a propiedades de un objeto que no existen, JavaScript te retornará **undefined**.



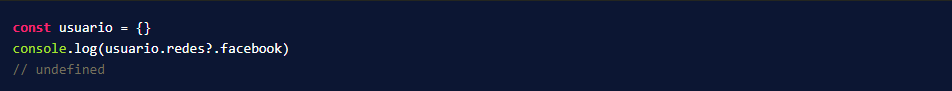
Al acceder a una propiedad más profunda de un objeto, que previamente fue evaluada como **undefined**, el **programa se detendrá y mostrará un error de tipo**.



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.



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.

## **No abuses del encadenamiento opcional**

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

Por ejemplo, en un objeto **usuario** que siempre existe, pero la propiedad redes es opcional, entonces se debería escribir **usuario.redes?.facebook** y no **usuario?.redes?.facebook**.

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.

# **BigInt y Nullish**

## **Big Int, enteros muy grandes**

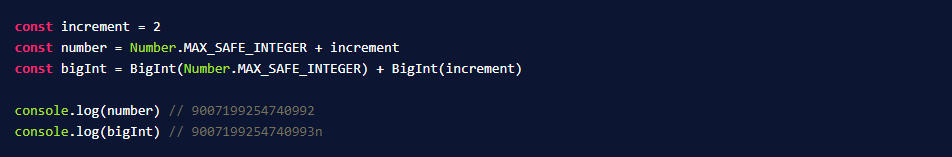
El nuevo dato primitivo **bigint** permite manejar números enteros muy grandes. Existen dos formas de crear un bigint: el número entero seguido de **n** o mediante la función **BigInt**



JavaScript tiene límites numéricos, un máximo **Number.MAX\_SAFE\_INTEGER** y un mínimo **Number.MIN\_SAFE\_INTEGER**.



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



Se añade la misma cantidad a ambos tipos de datos, sin embargo, el tipo numérico da un resultado diferente al esperado.

## **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**.



## **Diferencia entre el operador OR y el Nullish coalescing**

El operador **OR** (||) **evalúa un valor falsey**. Un valor **falsy** 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 **falsy** 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.

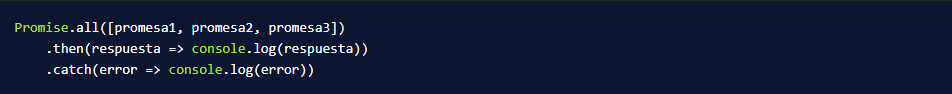


# **Promise.allSettled**

En alguna situación necesitarás manejar varias promesas y obtener sus resultados. ¿Cómo? Utilizando los métodos **Promise.all** y **Promise.allSettled**.

## **Promise.all**

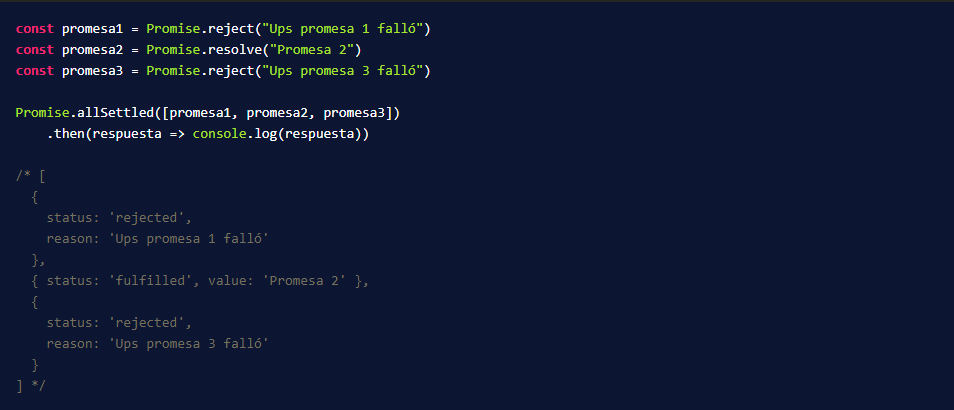
Sirve para manejar varias promesas al mismo tiempo. Recibe como argumento un array de promesas.



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.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**.



¿Debería usar **Promise.allSettled** en lugar de **Promise.all**?

**No**, porque ambas son muy útiles dependiendo cómo quieras manejar tus promesas.

# **globalThis y matchAll**

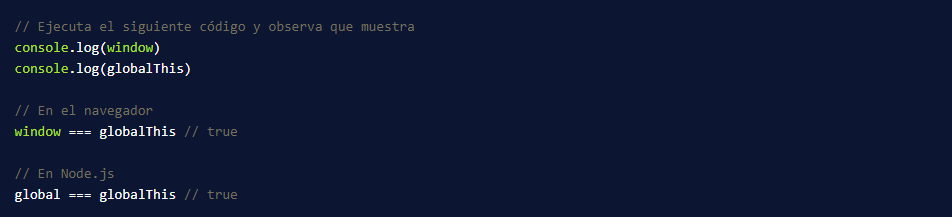
## **Objeto global para cualquier plataforma**

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.

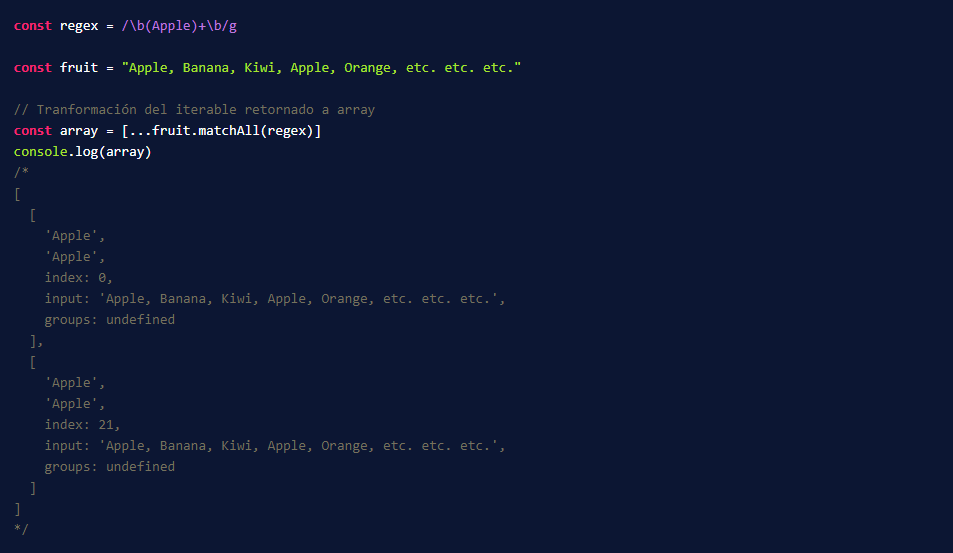


## **Método matchAll para expresiones regulares**

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.

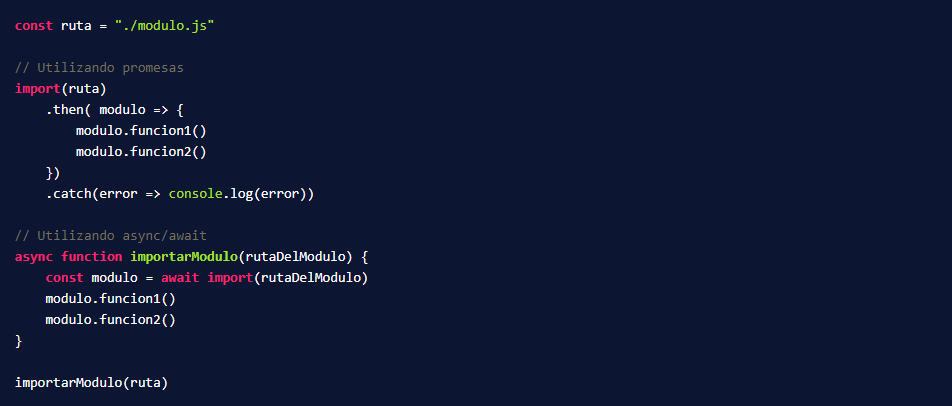


# **dynamic Import**

## **Cómo utilizar importación dinámica**

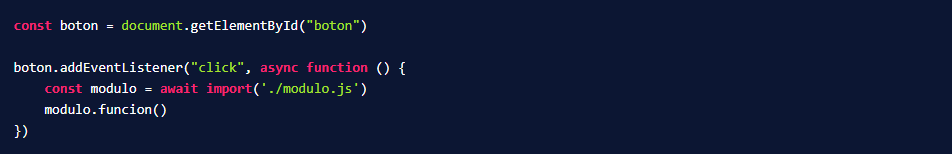
La importación dinámica consiste en cargar los módulos cuando el usuario los vaya a utilizar, y no al iniciar la aplicación. Esto permite que la página web sea más rápida, porque descarga menos recursos.

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



## **Ejemplo utilizando importación dinámica**

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.



Puedes usar las herramientas de desarrollador para visualizar la descarga de archivos al realizar clic en el botón.

# **ES12: numeric-separators y replaceAll**

## **Separadores numéricos**

Los separadores numéricos ayudan a la legibilidad de cantidades con varias cifras. Se utiliza el carácter 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.



De esta manera puedes identificar el número rápidamente.

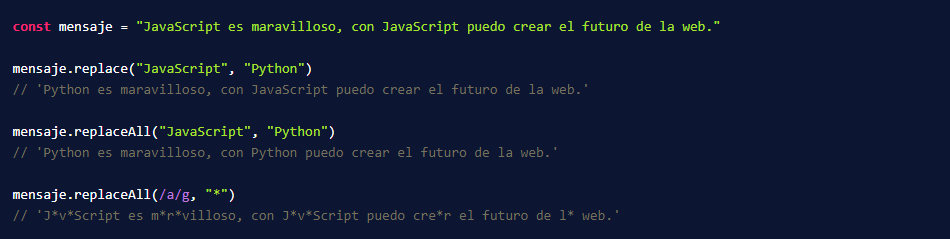
## **Método replaceAll**

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.

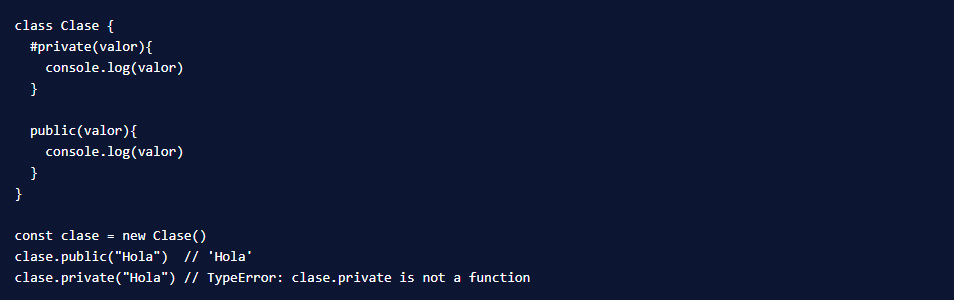
Este procedimiento fue creado para solucionar el problema que tenía el método **replace**, que realizaba la misma función de reemplazar elementos, pero solamente **una sola vez** por invocación.



# **promise-any y métodos privados**

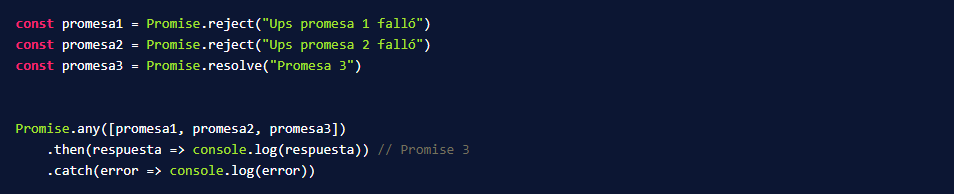
## **Métodos privados de clases**

Los métodos privados consiste en limitar el acceso a propiedades y métodos agregando el carácter 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.



## **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.



# **ES13: at**

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

**array.at(índice)**

## **Índices positivos y negativos en arrays**

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.

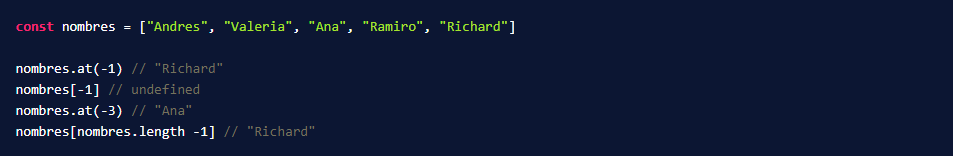


Los índices negativos comienzan desde **-1** hasta el negativo de la longitud total del array, de **derecha** a **izquierda**. El índice **-1** es la última posición.



## **Cómo utilizar el método at**

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



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.



# **top level await en el consumo de una API**

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.

## **Cómo utilizar top level await**

Anterior a ECMAScript 13, cuando se introdujo funciones asíncronas, si utilizabas await fuera de async, existirá un error de sintáxis.

// Error

