Contenido

[**ForEach** 1](#_Toc124096674)

[**Diferencia entre la estructura for y el método forEach** 1](#_Toc124096675)

[**Mostrar elementos de un array al usuario** 2](#_Toc124096676)

[**Mutable o Inmutable** 3](#_Toc124096677)

[**Diferencia entre mutabilidad e inmutabilidad** 4](#_Toc124096678)

[**Qué es el método map** 4](#_Toc124096679)

[**Diferencia entre la estructura for y el método map** 5](#_Toc124096680)

[**Diferencia entre forEach y map** 6](#_Toc124096681)

[**¿Pero también se pueden transformar los elementos con forEach?** 6](#_Toc124096682)

[**Map Reloaded** 7](#_Toc124096683)

[**Extrayendo datos necesarios** 8](#_Toc124096684)

[**Transformando objetos dentro de un array** 8](#_Toc124096685)

[**Filter** 9](#_Toc124096686)

[**Diferencia entre la estructura for y el método filter** 10](#_Toc124096687)

[**Filtrar elementos a partir de la propiedad de un objeto** 11](#_Toc124096688)

[**Reduce** 13](#_Toc124096689)

[**Diferencia entre la estructura for y el método reduce** 14](#_Toc124096690)

[**Uso del método reduce para una reducción** 15](#_Toc124096691)

[**Reduce Reloaded** 19](#_Toc124096692)

[**Objeto de frecuencias** 20](#_Toc124096693)

[**Some** 21](#_Toc124096694)

[**Diferencia entre la estructura for y el método some** 22](#_Toc124096695)

[**Every** 23](#_Toc124096696)

[**Diferencia entre la estructura for y el método every** 24](#_Toc124096697)

[**Find y FindIndex** 25](#_Toc124096698)

[**Diferencia entre la estructura for y los métodos find y findIndex** 26](#_Toc124096699)

[**Includes** 27](#_Toc124096700)

[**Índices positivos y negativos** 28](#_Toc124096701)

[**Diferencia entre la estructura for y el método includes** 28](#_Toc124096702)

[**Ejemplos utilizando el método includes** 29](#_Toc124096703)

[**Join** 30](#_Toc124096704)

[**Diferencia entre la estructura for y el método join** 30](#_Toc124096705)

[**Método split de strings** 32](#_Toc124096706)

[**Concat** 33](#_Toc124096707)

[**Diferencia entre la estructura for y el método concat** 33](#_Toc124096708)

[**Flat** 34](#_Toc124096709)

[**Diferencia entre la estructura for y el método flat** 35](#_Toc124096710)

[**FlatMap** 36](#_Toc124096711)

[**Mutable functions** 37](#_Toc124096712)

[**Método push** 38](#_Toc124096713)

[**Método unshift** 38](#_Toc124096714)

[**Método pop** 38](#_Toc124096715)

[**Método shift** 39](#_Toc124096716)

[**Método splice** 39](#_Toc124096717)

[**Sort** 40](#_Toc124096718)

[**Ordenamiento de palabras** 41](#_Toc124096719)

[**Ordenamiento de números** 42](#_Toc124096720)

[**Ordenamiento de objetos por su propiedad** 43](#_Toc124096721)

# **ForEach**

El método **forEach** de los arrays consiste **en ejecutar una función (callback)** **para cada uno de los elementos iterados**. Iterar significa repetir una acción varias veces.

Este método recibe **dos argumentos**:

* **La función** que itera cada elemento del array (obligatorio).
* **Un objeto** al que puede hacer referencia el contexto **this** en la función. Si se lo omite, será **undefined**. Recuerde que **this** en **arrow functions** es el **objeto global**.



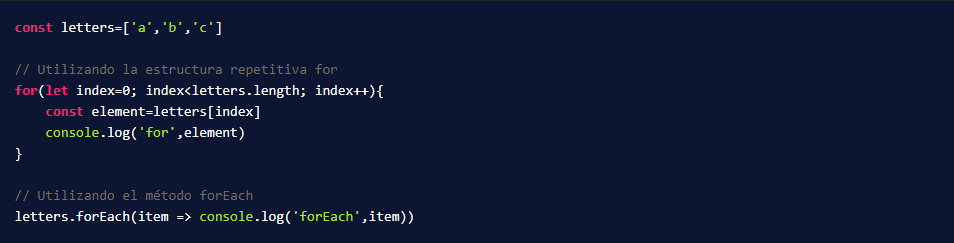
La función, que recibe como argumento el método **forEach**, utiliza tres parámetros opcionales:

* **El valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* **El índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* **El array que está iterando**.



## **Diferencia entre la estructura for y el método forEach**

Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.



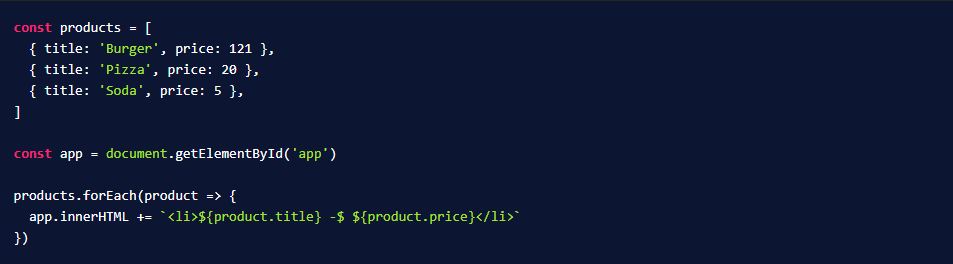
## **Mostrar elementos de un array al usuario**

Ahora que ya conoces cómo funciona el método **forEach**, utilízalo para agregar elementos al HTML y así que el usuario lo mire.

Por ejemplo, en un archivo HTML agrega una etiqueta **div** con un **id** app, que nos servirá para agregar nuestros elementos.



Después crea la lógica en un archivo de JavaScript.



Y listo, en pantalla aparecerán los elementos del array **products** con su título y precio.



# **Mutable o Inmutable**

Los conceptos de **mutabilidad** e **inmutabilidad** son muy importantes para los siguientes métodos de arrays. Existen métodos mutables que **cambian el array original** e inmutables que **devuelven un array diferente al original**.



# **Diferencia entre mutabilidad e inmutabilidad**

Con lo mencionado anteriormente**, mutable es aquella acción que cambia el valor en la referencia en memoria del elemento del array original**, provocando que cambien el **original** y **la copia**. Inmutable **es la acción en la que se cambia el valor, pero en una referencia diferente del original**, provocando que el original siga igual.

La mutabilidad es más flexible y una buena opción si se requiere cambiar, actualizar o eliminar datos; pero esto puede ocasionar fallos o resultados erróneos en nuestra aplicación. La inmutabilidad es más exigente, te permite generar nuevas estructuras para manejarlas sin cambiar la original; pero esto puede provocar que la memoria colapse.

**¿Cuál es mejor?**

La respuesta es ninguna, cada uno te permite manejar estructuras de datos, por ende, es necesario identificar cuál forma es la adecuada a aplicar en un algoritmo.

# **Qué es el método map**

El método **map** consiste en crear un nuevo array a partir de los elementos originales transformados mediante una función (callback) y **es inmutable**.

La transformación implica cambiar cualquier elemento en otro, ya sea un número, un objeto, otro array. Las posibilidades son infinitas.

Este método recibe **dos argumentos**:

* **La función** que itera y transforma cada elemento del array (obligatorio).
* **Un objeto** al que puede hacer referencia el contexto **this** en la función. Si se lo omite, será **undefined**. Recuerde que **this** en **arrow functions es el objeto global**.



La función, que recibe como argumento el método map, utiliza tres parámetros:

* **El valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* **El índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* **El array que está iterando**.



## **Diferencia entre la estructura for y el método map**

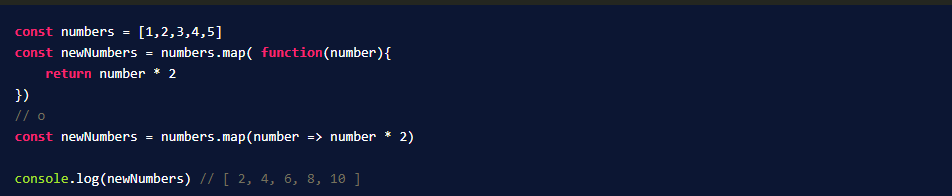
Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que transforme cada elemento de un array en el doble de sí mismo.

Si utilizamos la estructura **For**, necesitaremos un array adicional vacío y utilizarlo con el método **push** para agregar los elementos transformados. El método **push** es **mutable**.



Con el método **map**, solo debemos establecer la función que indique la transformación para cada elemento.



Recuerda siempre retornar un valor en la función callback del método.

## **Diferencia entre forEach y map**

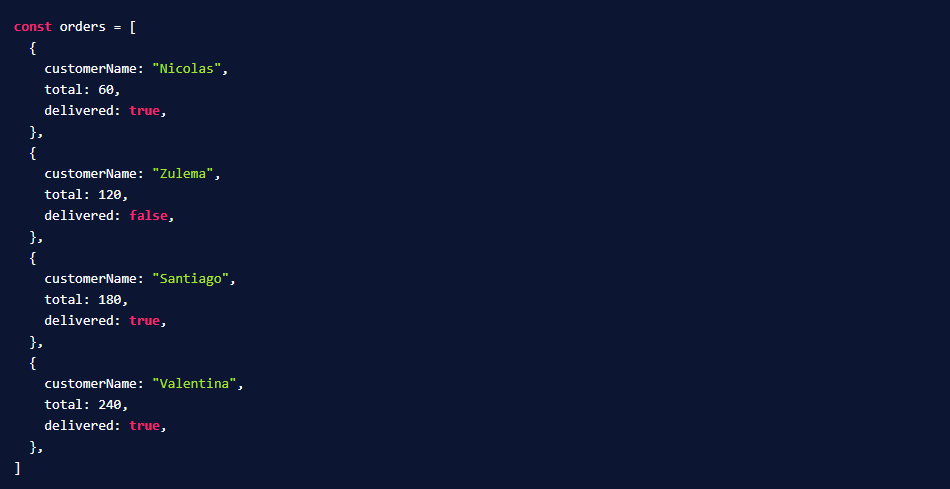
La principal diferencia entre estos dos es que forEach solamente **itera cada elemento**, mientras que map **itera y transforma cada elemento** en un nuevo array.

## **¿Pero también se pueden transformar los elementos con forEach?**

Sí, sin embargo, necesitas agregar cada elemento en cada iteración a un array nuevo, caso contrario puedes mutar el array original.

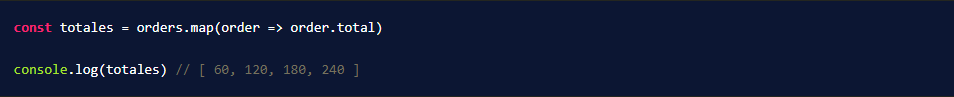
# **Map Reloaded**

Ahora que ya sabes cómo funciona el método map de arrays estudiemos diferentes situaciones que deberás manejar array de objetos como los siguientes datos de **orders**.



## **Extrayendo datos necesarios**

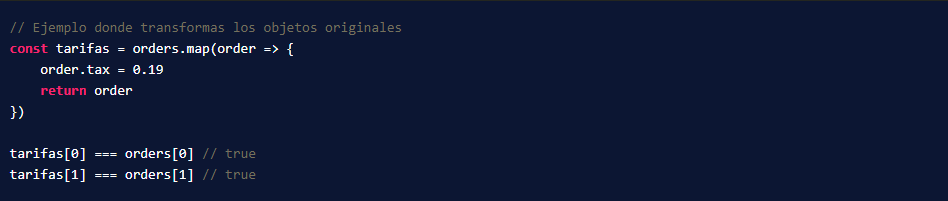
De un conjunto de datos a veces necesitas ciertos atributos para realizar una operación, extraer los nombres o cuantificar un suceso. El método **map** te permite extraer estos datos según los puedas utilizar.



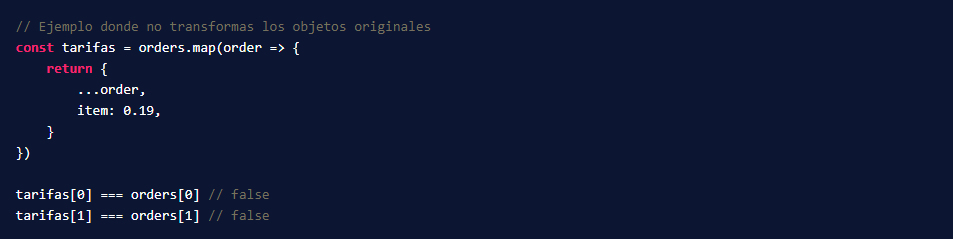
## **Transformando objetos dentro de un array**

De un conjunto de datos, de los cuales son objetos, a veces requieres añadir una propiedad nueva o eliminar una que no es necesaria. Si necesitas transformar objetos dentro de un array, debes tener en cuenta que cada objeto tiene una referencia en memoria propia. Por lo que, si transformas un objeto, también lo harás en el original como en el nuevo, aun cuando el método map es inmutable.

Los parámetros por referencia los debes tener presente cuando manejes objetos y arrays.



Una forma de evitar este comportamiento, es crear un nuevo objeto con el **spread operator** (operador de propagación).



# **Filter**

El método **filter** consiste en crear un nuevo array a partir de los elementos originales filtrados mediante una función (callback) que indica **la condición a cumplir** y **es inmutable**. Si la condición se cumple, retorna el elemento completo.

Este proceso recibe dos argumentos:

* **La función que itera y evalúa** si cada elemento del array si cumple con la condición especificada (obligatorio).
* **Un objeto** al que puede hacer referencia el contexto this en la función. Si se lo omite, será undefined. Recuerde que this en arrow functions es el objeto global.



La función, que recibe como argumento el método **filter**, utiliza tres parámetros:

* El **valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* El **índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* El **array que está iterando**.

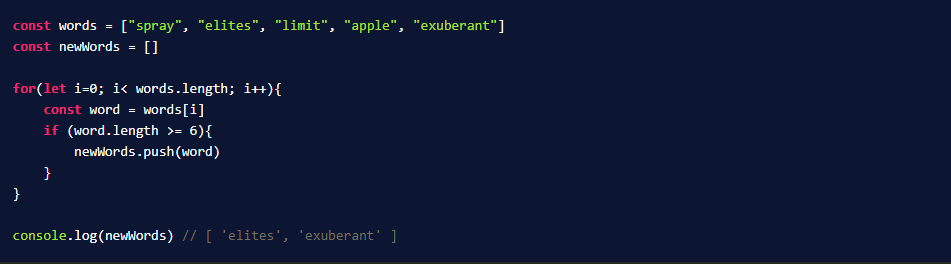


## **Diferencia entre la estructura for y el método filter**

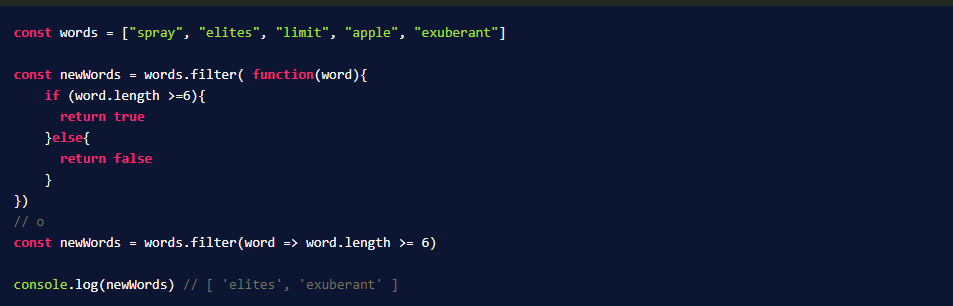
Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que filtre los elementos que tengan más de 6 letras en un array de palabras.

Si empleamos la estructura **for**, necesitaremos un array adicional vacío y usarlo con el método **push** para agregar los elementos que cumplan la condición. El método **push es mutable**.



Con el método **filter**, solo debemos establecer la función que indique la condición a cumplir para cada elemento.



Recuerda **siempre retornar un valor** en la función **callback** del método.

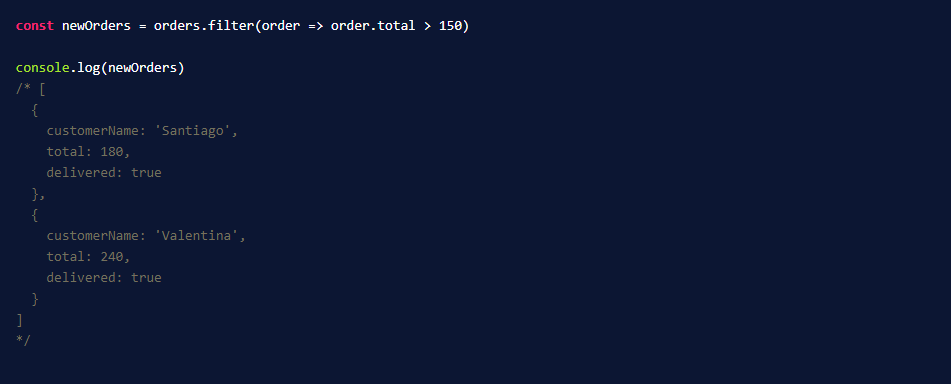
## **Filtrar elementos a partir de la propiedad de un objeto**

Con el método filter puedes filtrar los objetos de un array a partir de una condición referente a la propiedad de cada elemento.

Teniendo en cuenta que el nuevo array contendrá el objeto completo que haya cumplido con la condición especificada.



Por ejemplo, filtremos los elementos del array orders cuyo total sea mayor a 150.



# **Reduce**

El método **reduce es inmutable** y consiste retornar un solo valor del array iterado a partir de una función (callback) que indica de qué manera se itera cada elemento para reducirlo.

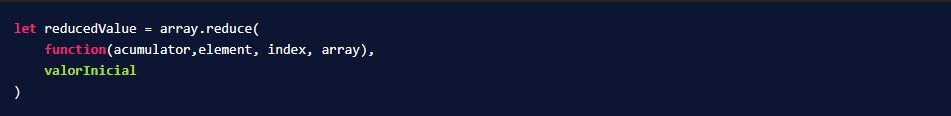
Este método recibe **dos argumentos**:

* **La función que itera** y reduce cada elemento del array. (obligatorio)
* **El valor inicial** que utilizará como argumento la función. Si no se especifica, en la primera iteración el valor inicial será el primer elemento del array y no ejecuta la función.



La función, que recibe como argumento el método **reduce**, utiliza cuatro parámetros:

* El **valor acumulado por la función** (callback). En la primera iteración será igual al valor inicial del argumento del método. (obligatorio)
* El **valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente. (obligatorio)
* El **índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* El **array que está iterando**.



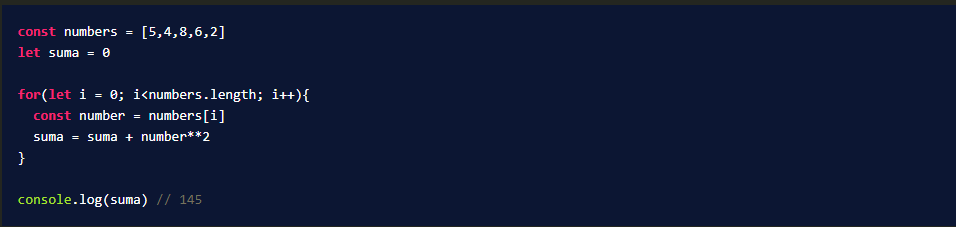
## **Diferencia entre la estructura for y el método reduce**

Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura for, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que calcule la suma de los cuadrados de los elementos de un array.

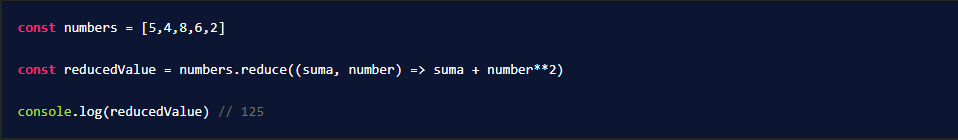
**Uso de la estructura for para una reducción**

Si utilizamos la estructura for, necesitaremos una variable acumuladora para sumar los elementos en cada iteración.

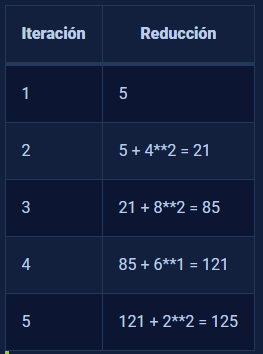


## **Uso del método reduce para una reducción**

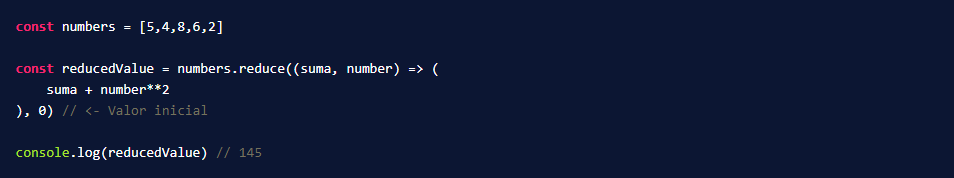
Con el método **reduce**, solo debemos establecer la función que indique la reducción para cada elemento.



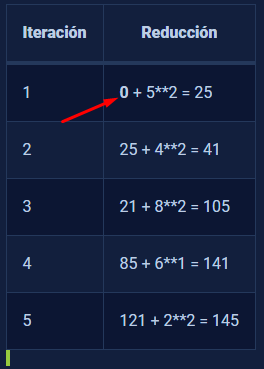
Observa que, si no especificamos el **valor inicial** del método, entonces tomará el primer elemento sin ejecutar la función reductora.



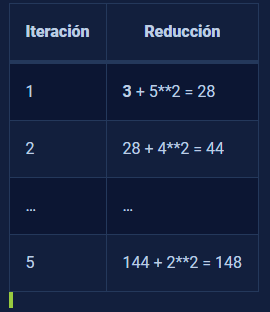
Por lo que debes **especificar el valor inicial** para solucionar este problema.

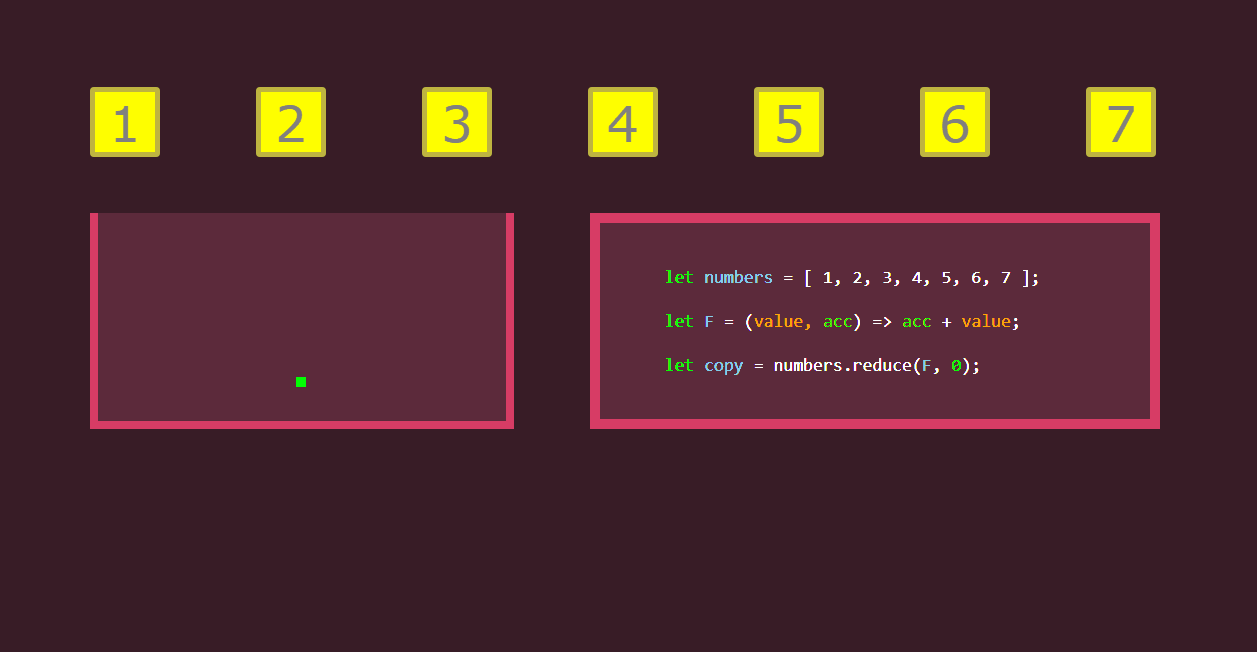


De esta manera se ejecutará la función reductora adecuadamente.



Si se ingresa otro valor inicial, por ejemplo **3**, entonces cambiaría la primera iteración.





# **Reduce Reloaded**

Ahora que ya sabes cómo funciona el método reduce de arrays, podemos utilizarlo para crear un objeto con la frecuencia de cada elemento de un array, es decir, cuántas veces aparece.

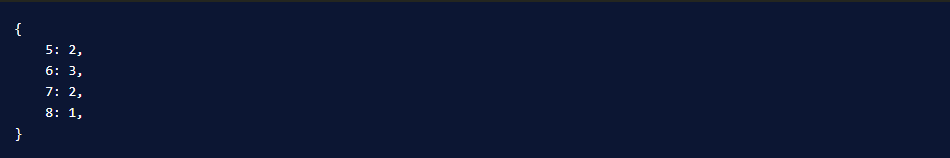
## **Objeto de frecuencias**

Para obtener un objeto de frecuencias de cada elemento de un array es necesario tener presente las siguientes consideraciones:

* Establecer un objeto vacío como valor inicial del método reduce.
* El objeto vacío también será nuestro acumulador.
* Verificar si el elemento ya existe en nuestro objeto de frecuencias.
* Si no existe, creamos la propiedad referente al elemento del array y le inicializamos en 1.
* Si ya existe solamente debemos aumentar en una unidad la propiedad de nuestro objeto referente al elemento del array.
* Finalmente, debes retornar el objeto dentro de la función del método reduce.
* Por ejemplo, con el siguiente array llamado items:

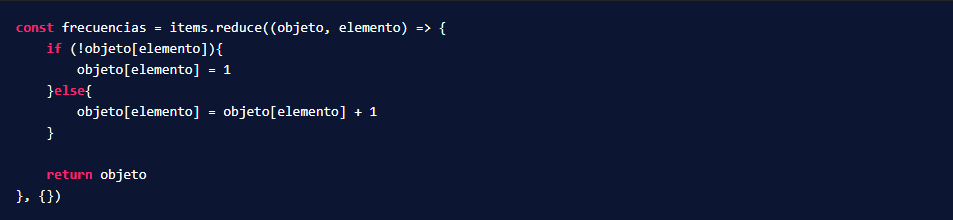


El objeto de frecuencias será el siguiente:



**Utilizando el método reduce para crear un objeto de frecuencias**

Una vez entendido la entrada y salida del algoritmo, entonces el código será utilizado de la siguiente manera:



De esta manera obtendrás el objeto de frecuencias. Puedes utilizar este algoritmo para contar elementos de cualquier array.

# **Some**

El método **some** es inmutable y consiste retornar un valor lógico verdadero **si existe al menos un elemento que cumpla la condición** establecida en la función (**callback**).

Este método recibe dos argumentos:

* La función que itera y evalúa cada elemento del array hasta que al menos uno cumpla con la condición especificada (obligatorio).
* Un objeto al que puede hacer referencia el contexto this en la función. Si se lo omite, será undefined. Recuerde que this en arrow functions es el objeto global.



La función, que recibe como argumento el método some, utiliza tres parámetros:

* **El valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* El **índice del elemento iterado**. Es decir, si es la primera iteración, será el índice **0**, y así sucesivamente.
* El **array que está iterando**.



## **Diferencia entre la estructura for y el método some**

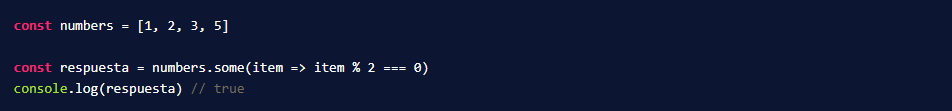
Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que indique si en un array existe al menos un número **par**.

Si utilizamos la estructura **for**, necesitaremos una variable de tipo booleana con el valor **false** e iterar hasta que la condición se cumpla. La palabra reservada **break** rompe el ciclo repetitivo.



Con el método **some** solo debemos establecer la función que indique la condición a cumplir para cada elemento.



# **Every**

El método everyes **inmutable** y consiste retornar un valor lógico verdadero si **todos los elementos cumplen con la condición** establecida en la función (**callback**).

Este método recibe dos argumentos:

* **La** **función** que itera y evalúa cada elemento del array hasta que al menos uno cumpla con la condición especificada (obligatorio).
* **Un** **objeto** al que puede hacer referencia el contexto **this** en la función. Si se lo omite, será **undefined**. Recuerde que this en arrow functions es el objeto global.



La función, que recibe como argumento el método **every**, utiliza tres parámetros:

* El **valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* El **índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* El **array que está iterando**.



## **Diferencia entre la estructura for y el método every**

Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que indique si en un array, todos los elementos son menores o iguales a 40.

Si utilizamos la estructura **for**, necesitaremos una variable de tipo booleana con el valor **true** e iterar hasta que una condición contraria al enunciado se cumpla. La palabra reservada **break** rompe el ciclo repetitivo.



Con el método **every**, solo debemos establecer la función que indique la condición a cumplir para cada elemento.



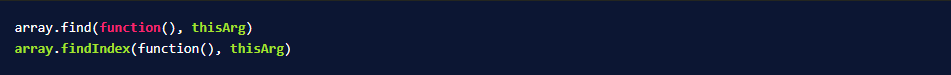
# **Find y FindIndex**

Los métodos **find** y **findIndex** consisten en encontrar el **primer elemento** **de un array** que cumpla con la condición especificada en la función (callback).

En el caso de **find** retornará el **elemento completo**, si cumple con la condición, caso contrario **retornará undefined**. El método **findIndex** retornará el **índice del elemento encontrado**, caso contrario retornará **-1**.

Estos procedimientos reciben dos argumentos:

* La **función** que itera y evalúa cada elemento del array hasta encuentre uno que cumpla con la condición especificada (obligatorio).
* Un **objeto** al que puede hacer referencia el contexto **this** en la función. Si se lo omite, será **undefined**. Recuerde que this en arrow functions es el objeto global.



La función, que recibe como argumento los métodos find y findIndex, utiliza tres parámetros:

* El **valor actual del elemento iterado**. Es decir, si es la primera iteración, será el primer elemento, y así sucesivamente.
* El **índice del elemento iterado**. Es decir, si es la primera iteración, será el índice 0, y así sucesivamente.
* El **array que está iterando**.



## **Diferencia entre la estructura for y los métodos find y findIndex**

Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

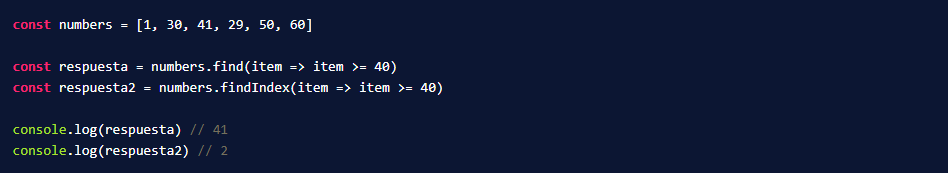
Por ejemplo, hagamos un algoritmo que devuelva el primer elemento que sea mayor a 40 de un array.

Si utilizamos la estructura **for**, necesitaremos una variable con el valor **undefined** e iterar hasta que se cumpla la condición. La palabra reservada **break** rompe el ciclo repetitivo.

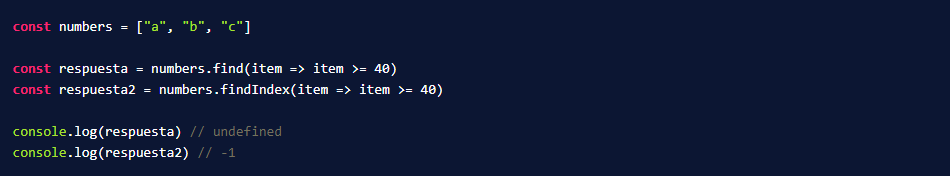


Si se necesita el índice, en lugar de una variable con el valor de **undefined**, debería estar un valor de **-1**. Y también cambiar el valor del elemento por el del índice del **for**.

Con los métodos **find** y **findIndex**, solo debemos establecer la función que indique la condición a cumplir para cada elemento.



Recuerda que, si los métodos **find** y **findIndex** no encuentran **EL** elemento, devolverán **undefined** y **-1**, respectivamente.



# **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**.



## **Diferencia entre la estructura for y el método includes**

Los métodos de *arrays* nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Por ejemplo, hagamos un algoritmo que indique si en un array existe un elemento en específico.

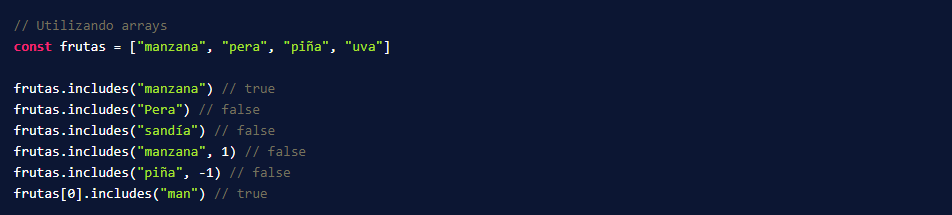
Si utilizamos la estructura **for**, necesitaremos una variable de tipo booleana con el valor **false** e iterar hasta que encuentre el elemento específico. La palabra reservada **break** rompe el ciclo repetitivo.



## **Ejemplos utilizando el método includes**

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





# **Join**

El método **join** une los elementos del array, mediante una separación, y retorna un **string**. Si un elemento es **undefined** o **null**, se convierte en una **cadena vacía**.

Este procedimiento recibe un argumento:

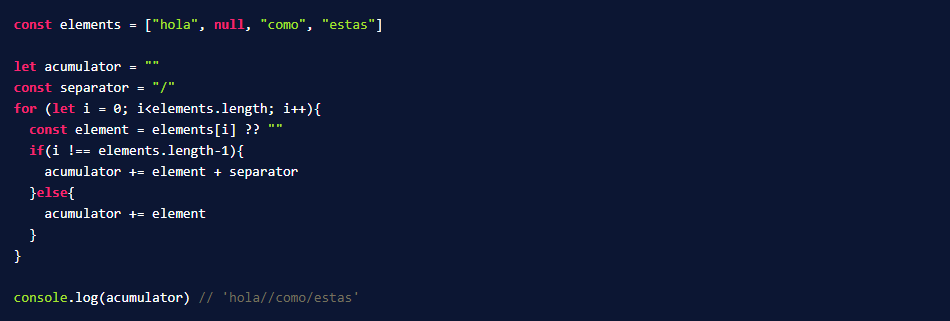
* La **separación** de cada elemento del array al unirlos.



## **Diferencia entre la estructura for y el método join**

Los métodos de arrays nos permiten realizar algoritmos con una menor cantidad de líneas que una estructura **for**, con un resultado igual o parecido.

Si utilizamos la estructura **for** para recrear el método **join**, necesitaremos una variable con el valor acumuladora con un string vacío y otra con el valor del separador. Se debe evaluar si existe elementos **null** o **undefined**, se lo puede realizar con el operador **nullish coalescing**.



Con el método **join** solamente debemos establecer el separador de cada elemento como argumento.

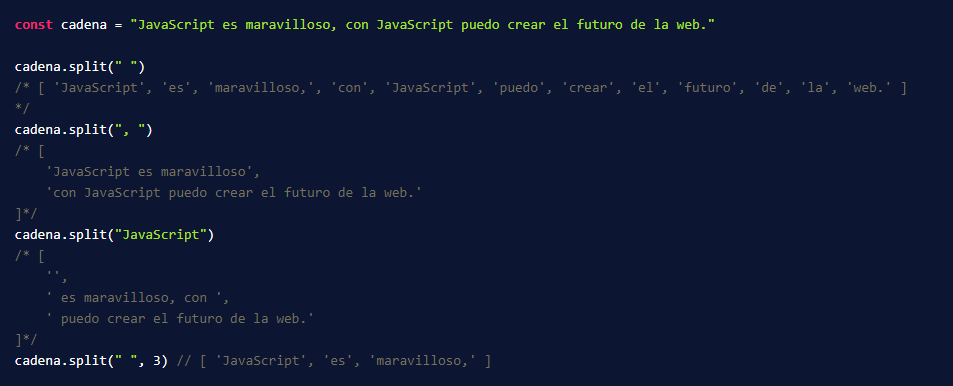


## **Método split de strings**

El método **split** de strings, es lo contrario que el método **join**, consiste en separar un string en substrings, indicando un valor a separar. Este método retornará un array de los elementos separados.

Este método recibe dos argumentos:

* El **separador** que especifica el conjunto de caracteres a separar en substrings.
* El **límite** de elementos separados a retornar.



# **Concat**

El método **concat** es inmutable y consiste en crear un nuevo *array* a partir de la unión de otros valores o arrays especificados como argumentos.

Este método recibe uno o varios argumentos:

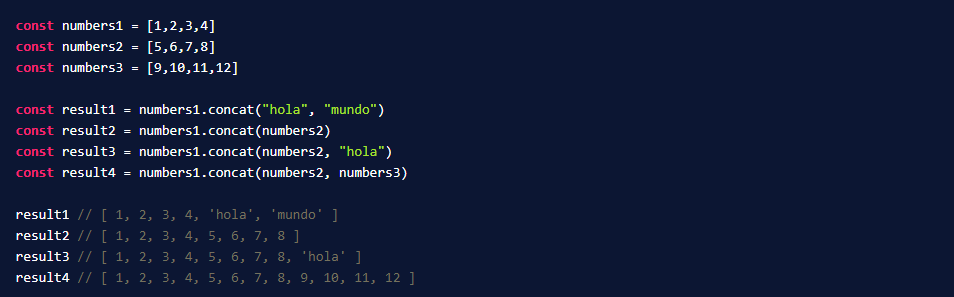
Valores cualesquiera y/o arrays para concatenar.



## **Diferencia entre la estructura for y el método concat**

Si deseas utilizar una estructura **for** para concatenar arrays, debes copiar el primer array sin su referencia en memoria para que no exista mutabilidad. Puedes realizar una copia con el **spread operator** (operador de propagación) o con el nuevo método **structuredClone**. También puedes utilizar el método **push** si no importa la mutabilidad del array original.

Con el método **concat**, solo debemos establecer el/los elemento/s a concatenar a un array de manera inmutable, es decir, los elementos originales no cambiarán.



# **Flat**

El método **flat** es inmutable que consiste en retornar un *array* donde los sub-arrays han sido aplanados hasta una profundidad especificada. **El aplanamiento consiste en transformar un array de arrays a una sola dimensión**.

Este procedimiento 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**.

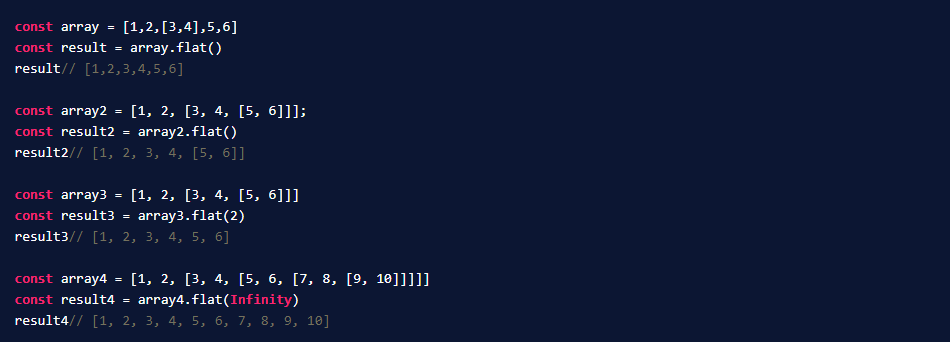


## **Diferencia entre la estructura for y el método flat**

Si se utiliza la estructura **for** para aplanar un array de arrays, es necesario utilizar otra estructura **for** para cada sub-nivel del array o recursión.



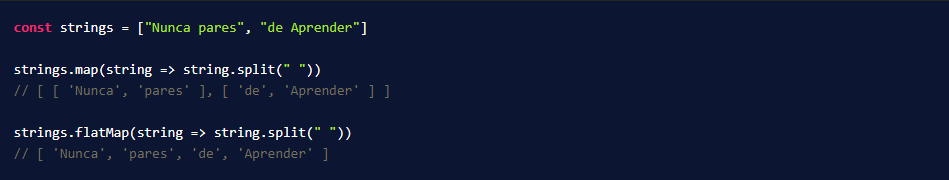
Con el método **flat** solamente es necesario indicar la profundidad de aplanamiento del array.



# **FlatMap**

El método **flatMap** es inmutable y consiste en la 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 procedimiento recibe los mismos argumentos que el método **map**.





Cuidado con el método **flatMap**, primero realiza el **map** y luego el **flat**.



# **Mutable functions**

Las funciones mutables consisten en cambiar el array original. Estos métodos son:

* push
* unshift
* pop
* shift
* splice
* sort

## **Método push**

El método **push** agrega uno o varios elementos al final del array original. El método recibe como argumento los valores a agregar. Retorna el número de elementos del array mutado.



## **Método unshift**

El método **unshift** agrega uno o varios elementos al inicio del array original. El método recibe como argumento los valores a agregar. Retorna el número de elementos del array mutado.



## **Método pop**

El método **pop** extrae el elemento del final del array original. El método no recibe ningún argumento. Retorna el elemento extraído, si no se guarda en una variable, el Garbage Collection{target="\_blank"} lo elimina.



## **Método shift**

El método **shift** extrae el elemento del inicio del array original. El método no recibe ningún argumento. Retorna el elemento extraído, si no se guarda en una variable, el Garbage Collection{target="\_blank"} lo elimina.

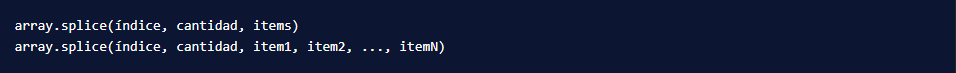


## **Método splice**

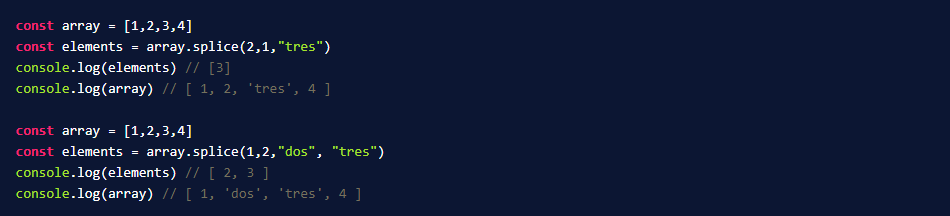
El método **splice** extrae uno o varios elementos del array original a partir del índice y los reemplaza con otro elemento especificado como argumento. Retorna un array con los elementos extraídos, si no se guarda en una variable, el Garbage Collection{target="\_blank"} lo elimina.

El método **splice** recibe tres argumentos:

* El **índice** donde comenzará a cambiar el array.
* La **cantidad de elementos** que serán reemplazados.
* **Uno** o **varios elementos** que reemplazarán a los originales del array.



Ejemplos utilizando el método splice



# **Sort**

El método **sort** es mutable y consiste en ordenar un array a partir de los valores Unicode de los caracteres y este retorna un array con la misma referencia en memoria que el original.

Este proceso recibe **un argumento**:

* Una **función** **de comparación** que compara cada elemento con otro. Por defecto, evalúa el valor Unicode del carácter.



La función comparativa, que recibe como argumento el método **sort**, utiliza dos parámetros:

* El **primer elemento** a comparar.
* El **segundo elemento** a comparar.

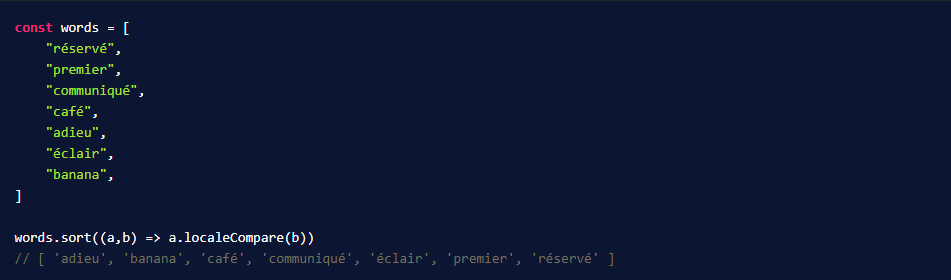


Si la función comparativa retorna un número mayor que 0, entonces el primer elemento se sitúa antes que el segundo. Si es menor que 0, entonces el segundo elemento se sitúa antes que el primero. Esto es importante, ya que ordenar los elementos por el valor Unicode provoca resultados inesperados.

## **Ordenamiento de palabras**

Si ordenas un array de palabras, puede ordenar correctamente por el valor Unicode de los caracteres de la palabra. Sin embargo, en algunos navegadores o entornos de ejecución (como Node) puede que esto falle, por lo tanto, debes utilizar la función **localeCompare**.

Ten en cuenta que las mayúsculas, minúsculas y caracteres con tilde tienen un valor Unicode diferente, por lo que debes asegurarte que todas las palabras estén en la misma condición.



## **Ordenamiento de números**

Si ordenas un array de números, provoca un ordenamiento inesperado porque ordena por el valor Unicode de los caracteres del número.



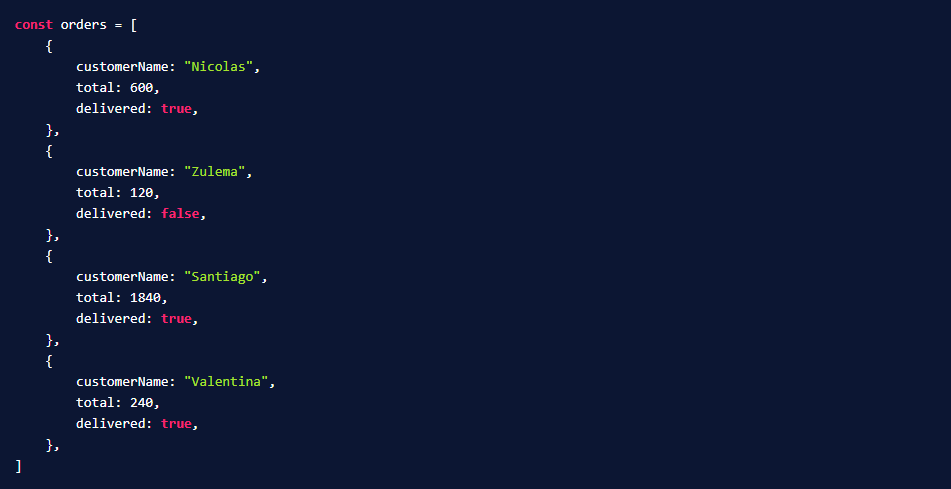
Para arreglar este comportamiento, es necesario utilizar la siguiente función comparativa:

* De manera ascendente **(a, b) => a - b**
* De manera descendente **(a, b) => b – a**



## **Ordenamiento de objetos por su propiedad**

A partir de la función comparativa puedes ordenar los elementos de cualquier forma. Cuando necesites ordenar un array de objetos, compara una propiedad de tipo numérica de la misma forma que el ordenamiento de números, ya sea de manera descendente o ascendente.



Por ejemplo, creemos un algoritmo que ordene los pedidos del array orders por su valor total.

