

## Métodos avanzados de arrays

### ¿Para qué es esto?

La idea de este apartado es **volver a los array** y aprender **métodos avanzados**. Esto ya no es tanto de programación común sino que es más para aprender **como se trabaja de una forma moderna con javascript**. Además estos metodos son importantes para aprender un framework a futuro.

### Primero unos ejercicios

Crear varias funciones que sirvan para manejar ciertas acciones con arrays.



1. Crear una función que ordene un array de menor a mayor o de mayor a menor, según se lo indique y lo devuelva.

#### Ejemplo

```
const arrayOrdenado = ordenar([2, 10, 6, 1], "ascendente");  
const arrayOrdenado2 = ordenar([76, 28, 61, 1], "descendente");
```

2. Una función que busque dentro de un array pasado por parámetro, un elemento que también es pasado por parámetros y cuando lo encuentre, que lo devuelva. De no encontrarlo, retornará false.

#### Ejemplo

```
let numero1 = buscar([2, 10, 6, 1] , 6);  
let numero2 = buscar([2, 10, 6, 1] , 3);  
  
console.log(numero1);  6  
console.log(numero2);  false
```

# Funciones callback

## Introducción

Como te das cuenta, **es cansador** escribir el código de los dos ejercicios anteriores. Tener que programar **desde cero** la función para ordenar arrays y la función para buscar un elemento dentro, es tedioso y no está bueno.

Para suerte nuestra, **ya hay funciones programadas** que podemos usar. Los arrays ya tienen un método para ordenar que podemos usar para no tener que programar el algoritmo desde cero.

### **Recordemos**

Función: cualquier función independiente que podemos llamar. Como alert(), parseInt(), etc.

Método: cualquier función que dependa de otra cosa para ser llamada. Como write, que depende del objeto document y debemos usar la nomenclatura del punto.

Estos métodos avanzados de arrays, **implementan** lo que se llama en programación como **funciones callbacks**. Nosotros no tenemos que programar estas funciones desde cero, simplemente **las usamos** y listo. Pero es importante entender que son y como funcionan.

## Métodos que vamos a ver

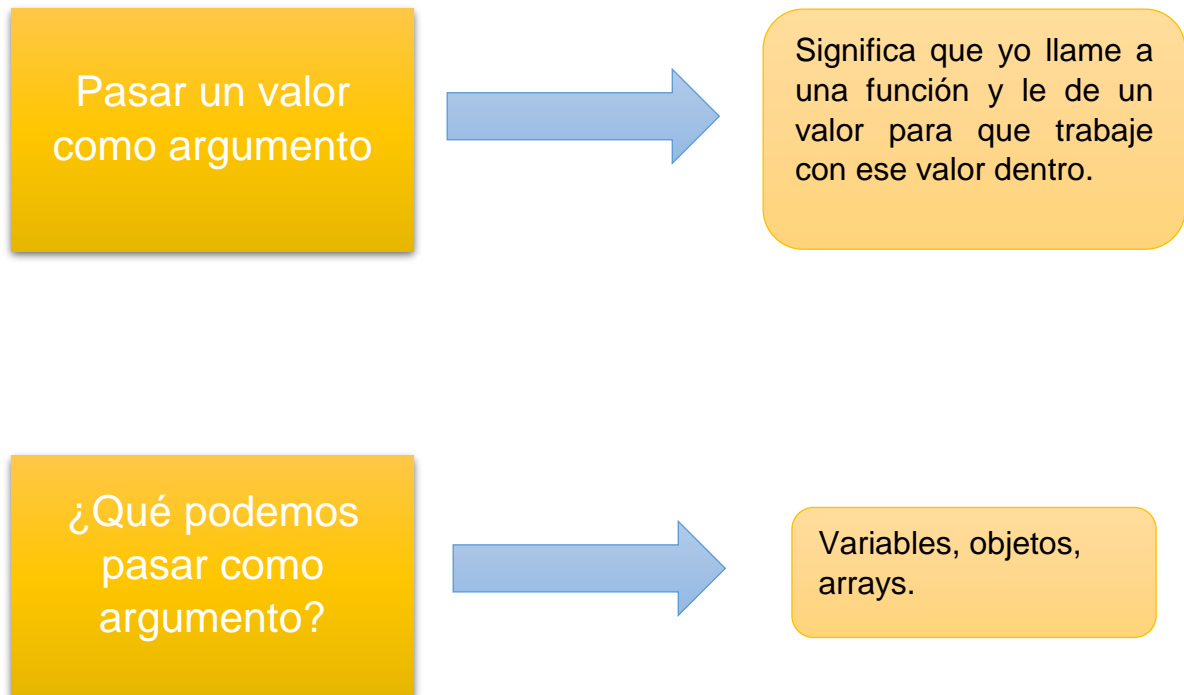
- array.sort() → Método que ordena los elementos del array
- array.forEach() → Recorre los elementos del array
- array.some() → Devuelve true si alguno de los elementos cumplen con una condición que nosotros le digamos
- array.every() → Devuelve true si todos los elementos cumplen con la condición
- array.map() → Transforma los elementos de un array y lo retorna.
- array.filter() → Retorna los elementos que cumplan con una condición que le digamos
- array.reduce() → Reduce cada elemento del array a un solo valor

importantes

## Función pasada por parámetros

Esto puede explotar un poco la cabeza.

Recordemos que significaba pasar parámetros o pasar un valor como argumento.



Es decir, si quiero una función que haga la **suma de dos números**, yo le voy a **pasar** los números como argumento.

En definitiva, los argumentos son los **valores que usa la función para cumplir su tarea**.

**¿Qué opinas si te digo que también puedo llamar a una función y pasarle por parámetros otra función?**

Puede ser medio confuso pensar en una función que reciba otra función como argumento. Es como una función dentro de otra.

La idea de esto es que la primera función ejecute la segunda cuando ocurran ciertas cosas, ciertos eventos. Estas funciones se llaman **funciones callback**.

## Veamos los callback en código

Supongamos que tenemos una función llamada `uno()` y otra que sea anónima (no tiene nombre propio).

**La primera** función mandará un `console.log` avisando de que está dentro de la primera función. **La segunda** función mandará un `console.log` avisando de que está dentro de la segunda.

```
function uno(callback){  
  console.log("estoy dentro de la primera función");  
  
  callback();  
}
```

La primera función recibe la función anónima y le pone de nombre `callback`

Luego la función decide ejecutarla después del `console.log`

Hasta acá tenemos la definición de la función `uno`. Recibe la función por parámetros y la manda a ejecutar.

¿Ahora, en dónde está el código de la segunda función? ¿En donde se escribe el `callback`?



en la llamada a la función `uno`

```
uno( () => console.log("estoy dentro de la función 2") );
```

Dentro de la misma llamada, puedo escribir el código de la función 2.

Lo que pasó fue que el flujo principal llamó a la función uno y le dijo “tomá, vas a usar esta función, ejecutala luego del `console.log`”. y la función uno ejecuta su código y luego el de la función que recibió.

resultado en consola:

```
estoy dentro de la primera función
estoy dentro de la función 2
```

Si la función uno quisiera, mandaría a ejecutar la función que recibe la veces que quiera.

```
function uno(callback){
  console.log("estoy dentro de la primera función");
  callback();
  callback();
  callback();
  callback();
  console.log("estoy dentro de la primera función");
}

uno( () => console.log("estoy dentro de la función 2") );
```

```
estoy dentro de la primera función
estoy dentro de la función 2
estoy dentro de la función 2
estoy dentro de la función 2
estoy dentro de la función 2
estoy dentro de la primera función
```

O lo puede ejecutar cuando se cumpla una condición

```
function uno(callback){
  let numero = 2;

  console.log("estoy dentro de la primera función");

  if(numero > 3){
    callback();
  }

  console.log("estoy dentro de la primera función");
}

uno( () => console.log("estoy dentro de la función 2") );
```

```
estoy dentro de la primera función
estoy dentro de la primera función
```

En este caso, no se cumple la condición así que no ejecutará el callback

También definir que el callback va a recibir un parámetro que le da la función uno.

```
function uno(callback){  
  console.log("estoy dentro de la primera función");  
  callback("soy un callback");  
  console.log("estoy dentro de la primera función");  
}  
  
uno( (mensaje) => console.log(mensaje) );
```

La función llama al callback y le manda el mensaje por parámetros.

Definimos que el callback va a recibir un parámetro llamado mensaje y que lo va a mostrar en un console.log sea lo que sea

Resultado:

```
estoy dentro de la primera función  
soy un callback  
estoy dentro de la primera función
```

La función dos muestre el mensaje que le dio la función uno.

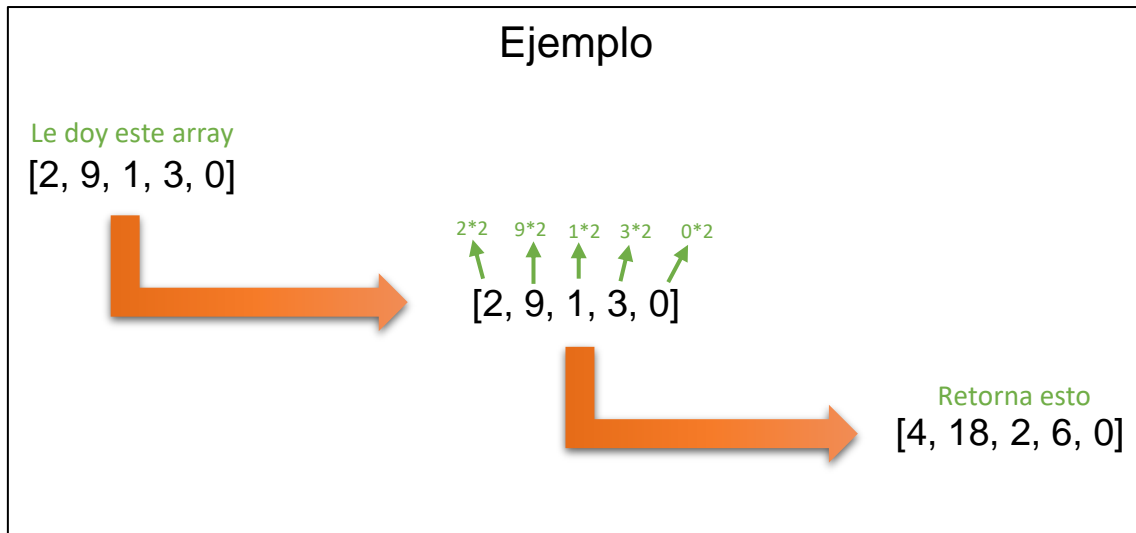
El beneficio de esto es poder extender una función. Poder ejecutar un código si se cumple un evento.

Por ejemplo, imaginemos que queremos que cuando el usuario apriete un `<button>` el programa muestre un formulario para ingresar datos. Una función estará esperando a que ocurra el evento y cuando ocurra, manda a ejecutar la función callback que muestra el formulario.

Otro ejemplo, se hace una petición de datos a una base de datos (la primera función se encarga de eso) y espera a que lleguen. Cuando lleguen, ejecuta la segunda función callback mandandoles los datos para que los muestre en consola (por ejemplo).

## Ejercicio sin callback

Programar una función que reciba un array y que transforme todos sus elementos de la siguiente manera: agarra cada elemento y lo multiplica por 2. Al finalizar, retorna el nuevo array.



## Código

```
function transformar(arr){  
  for(let i = 0; i < arr.length; i++){  
    //multiplico cada num por 2  
    arr[i] *= 2;  
  }  
  
  return arr;  
}  
  
//le mando el siguiente array  
console.log(transformar([2, 9, 1, 3, 0]));
```

► (5) [4, 18, 2, 6, 0]

Esto está bien, pero la pregunta es ¿qué pasa si el programador, en lugar de querer multiplicar todos los números por 2, quiera hacerlo por 3? En ese caso **se deberá crear otra función**. ¿Pero y si el programador quiera multiplicar por 4, o 5, o cualquier otro número? ¿cuántas funciones tendríamos? De seguro, **infinitas funciones**.

Acá es cuando tenemos la ayuda de las funciones callback

## Código con callback

```
function transformar(arr, callback){
  for(let i = 0; i < arr.length; i++){
    //se captura el número
    let numero = arr[i];
    //se lo manda al callback para que trabaje con él y lo retorna transformado
    arr[i] = callback(numero);
  }

  return arr;
}

//array
const nuevoArray = transformar([2, 9, 1, 3, 0], (num) => num*2);
console.log(nuevoArray);
```

Le indico de que manera quiero que transforme los números



```
► (5) [4, 18, 2, 6, 0]
```

Es exactamente el mismo resultado

De esta forma, si queremos transformar cada uno de los elementos de formas diferentes, basta con escribir diferente el algoritmo del callback

```
console.log(transformar([2, 9, 1, 3, 0], (num) => num*10));
console.log(transformar([2, 9, 1, 3, 0], (num) => "a"));
console.log(transformar([2, 9, 1, 3, 0], (num) => Math.pow(num, 3)));
console.log(transformar([2, 9, 1, 3, 0], (num) => num/2));
```

```
► (5) [20, 90, 10, 30, 0]
► (5) ["a", "a", "a", "a", "a"]
► (5) [8, 729, 1, 27, 0]
► (5) [1, 4.5, 0.5, 1.5, 0]
```

Para suerte nuestra, esta función **ya está programada** en javascript y trabaja con callbacks. No debemos programarla desde cero. Se llama map y pertenece a los arrays. Es decir, para llamarla, debemos hacerlo desde un array, ya que es un método.

`arreglo.map()` más adelante vamos a ver como utilizar este método.



## Ejercicios con callbacks

Estos ejercicios servirán para practicar la lógica de resolución de problemas. Estos ejercicios **no los vas a programar desde cero habitualmente en tu día a día**. Solo son ejercicios para practicar.

1. Crear una función llamada `ordenar()`, que reciba un array como primer parámetro y reciba un callback como segundo parámetro que se encargue de determinar si ordenar de forma ascendente o descendente.

La función `ordenar` se encarga de recorrer el array que recibe y en cada iteración, ejecuta el callback mandándole dos valores: el anterior y el siguiente.

El callback recibe esos dos valores (`a`, `b`) y debe retornar un booleano dependiendo del resultado de una resta. Debe comprobar:

    si el resultado de `a - b` es menor que cero (de ser así, se ordena ascendentemente).

    Si el resultado de `b - a` es menor que cero (de ser así, se ordena descendentemente).

2. Crear una función llamada `porCada()`, que reciba un array y un callback con el algoritmo que va a ejecutar por cada ejecución del `foreach`.

La función `porCada` se encarga de recorrer el array y de llamar al callback en cada vuelta de bucle y pasarle cada elemento.

En la llamada a la función se debe escribir cualquier algoritmo, por ejemplo, que muestre los elementos que obtiene.

```
porCada([2,3,4,1], (elem) => document.write(elem + "\n"));
```

3. Crear una función llamada `algunos()` y lo que tiene que hacer es retornar `true` si al menos uno de los elementos del array cumple con una condición puesta en el callback. Si no hay uno o más elementos que cumplan la función, se retorna `false`.

### Ejemplo

```
algunos([25, 42, 18, 17, 19], (edad) => edad < 18)      true
```

```
algunos([15, 17, 32, 17, 20], (edad) => edad < 18)      true
```

```
algunos([25, 42, 18, 20, 19], (edad) => edad < 18)      false
```

4. Crear una función llamada `todos()` que sea igual a la anterior función, pero con la diferencia de que retorna `true` si TODOS los elementos cumplen con la condición.
5. Crear una función `transformar()`, que reciba un array y un callback. La función se encarga de retornar un nuevo array con los elementos transformados. El algoritmo del callback se encarga de decir cómo transformar cada elemento del array.

### Ejemplo

```
transformar([25, 42, 18, 17, 19], (num) => num*10)
transformar(["Pablo", "Franco"], (nombre) => "nombre: "+nombre)
transformar([false, false, false], (state) => !state)
```

6. Crear una función llamada `filtro()`. La función debe retornar un nuevo array con los elementos que cumplan con una condición escrita en el callback.

### Ejemplo

```
filtro([25, 42, 18, 17, 19], (num) => num<20)    [18, 17, 19]
filtro(["Pablo", "Franco", "José", "Francisco"], (nombre) => nombre[0] == "F")
    ["Franco", "Francisco"]
transformar([objPersona1, objPersona2, objPersona3], (obj) => obj.edad >= 18)
    [objPersona1] //imagineos que persona1 es el único mayor de edad
```

7. Crear una función llamada `reducir` que se encargue de reducir los valores de un array a un solo valor.





