

# 30 Days Of JavaScript: Higher Order Functions

---



LinkedIn




Follow @asabeneh

Author: [Asabeneh Yetayeh](#)

January, 2020

[<< Day 8](#) | [Day 10 >>](#) Day 5

- [Day 9](#)
  - [Higher Order Function](#)
    - [Callback](#)
    - [Returning function](#)
    - [Setting time](#)
      - [Setting Interval using a setInterval function](#)
      - [Setting a time using a setTimeout](#)
  - [Functional Programming](#)
    - [forEach](#)
    - [map](#)
    - [filter](#)
    - [reduce](#)
    - [every](#)
    - [find](#)
    - [findIndex](#)
    - [some](#)
    - [sort](#)
      - [Sorting string values](#)
      - [Sorting Numeric values](#)
      - [Sorting Object Arrays](#)
  -  [Exercises](#)
    - [Exercises: Level 1](#)
    - [Exercises: Level 2](#)
    - [Exercises: Level 3](#)

## Day 9

---

### Higher Order Function

---

Higher order functions are functions which take other function as a parameter or return a function as a value. The function passed as a parameter is called callback.

## Callback

A callback is a function which can be passed as parameter to other function. See the example below.

```
// a callback function, the name of the function could be any name
const callback = (n) => {
  return n ** 2
}

// function that takes other function as a callback
function cube(callback, n) {
  return callback(n) * n
}

console.log(cube(callback, 3))
```

## Returning function

Higher order functions return function as a value

```
// Higher order function returning an other function
const higherOrder = n => {
  const doSomething = m => {
    const doWhatever = t => {
      return 2 * n + 3 * m + t
    }
    return doWhatever
  }
  return doSomething
}

console.log(higherOrder(2)(3)(10))
```

Let us see where we use call back functions. For instance the *forEach* method uses call back.

```
const numbers = [1, 2, 3, 4, 5]
const sumArray = arr => {
  let sum = 0
  const callback = function(element) {
    sum += element
  }
  arr.forEach(callback)
  return sum
}
```

```
}  
console.log(sumArray(numbers))
```

15

The above example can be simplified as follows:

```
const numbers = [1, 2, 3, 4]  
  
const sumArray = arr => {  
  let sum = 0  
  arr.forEach(function(element) {  
    sum += element  
  })  
  return sum  
}  
console.log(sumArray(numbers))
```

15

## Setting time

In JavaScript we can execute some activities in a certain interval of time or we can schedule(wait) for some time to execute some activities.

- setInterval
- setTimeout

### Setting Interval using a setInterval function

In JavaScript, we use setInterval higher order function to do some activity continuously with in some interval of time. The setInterval global method take a callback function and a duration as a parameter. The duration is in milliseconds and the callback will be always called in that interval of time.

```
// syntax  
function callback() {  
  // code goes here  
}  
setInterval(callback, duration)
```

```
function sayHello() {  
  console.log('Hello')
```

```
}  
setInterval(sayHello, 1000) // it prints hello in every second, 1000ms is 1s
```

## Setting a time using a setTimeout

In JavaScript, we use `setTimeout` higher order function to execute some action at some time in the future. The `setTimeout` global method take a callback function and a duration as a parameter. The duration is in milliseconds and the callback wait for that amount of time.

```
// syntax  
function callback() {  
  // code goes here  
}  
setTimeout(callback, duration) // duration in milliseconds  
  
function sayHello() {  
  console.log('Hello')  
}  
setTimeout(sayHello, 2000) // it prints hello after it waits for 2 seconds.
```

## Functional Programming

---

Instead of writing regular loop, latest version of JavaScript introduced lots of built in methods which can help us to solve complicated problems. All builtin methods take callback function. In this section, we will see *forEach*, *map*, *filter*, *reduce*, *find*, *every*, *some*, and *sort*.

### forEach

*forEach*: Iterate an array elements. We use *forEach* only with arrays. It takes a callback function with elements, index parameter and array itself. The index and the array optional.

```
arr.forEach(function (element, index, arr) {  
  console.log(index, element, arr)  
})  
// The above code can be written using arrow function  
arr.forEach((element, index, arr) => {  
  console.log(index, element, arr)  
})  
// The above code can be written using arrow function and explicit return  
arr.forEach((element, index, arr) => console.log(index, element, arr))  
  
let sum = 0;  
const numbers = [1, 2, 3, 4, 5];
```

```
numbers.forEach(num => console.log(num))
console.log(sum)
```

```
1
2
3
4
5
```

```
let sum = 0;
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => sum += num)
```

```
console.log(sum)
```

```
15
```

```
const countries = ['Finland', 'Denmark', 'Sweden', 'Norway', 'Iceland']
countries.forEach((element) => console.log(element.toUpperCase()))
```

```
FINLAND
DENMARK
SWEDEN
NORWAY
ICELAND
```

## map

*map*: Iterate an array elements and modify the array elements. It takes a callback function with elements, index , array parameter and return a new array.

```
const modifiedArray = arr.map(function (element, index, arr) {
  return element
})
```

```
/*Arrow function and explicit return
const modifiedArray = arr.map((element,index) => element);
*/
//Example
const numbers = [1, 2, 3, 4, 5]
const numbersSquare = numbers.map((num) => num * num)
```

```
console.log(numbersSquare)
```

```
[1, 4, 9, 16, 25]
```

```
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']  
const namesToUpperCase = names.map((name) => name.toUpperCase())  
console.log(namesToUpperCase)
```

```
['ASABENEH', 'MATHIAS', 'ELIAS', 'BROOK']
```

```
const countries = [  
  'Albania',  
  'Bolivia',  
  'Canada',  
  'Denmark',  
  'Ethiopia',  
  'Finland',  
  'Germany',  
  'Hungary',  
  'Ireland',  
  'Japan',  
  'Kenya',  
]  
const countriesToUpperCase = countries.map((country) => country.toUpperCase())  
console.log(countriesToUpperCase)
```

```
/*  
// Arrow function  
const countriesToUpperCase = countries.map((country) => {  
  return country.toUpperCase();  
})  
//Explicit return arrow function  
const countriesToUpperCase = countries.map(country => country.toUpperCase());  
*/
```

```
['ALBANIA', 'BOLIVIA', 'CANADA', 'DENMARK', 'ETHIOPIA', 'FINLAND', 'GERMANY', 'HUNGARY', 'IREL
```

```
const countriesFirstThreeLetters = countries.map((country) =>  
  country.toUpperCase().slice(0, 3)  
)
```

```
["ALB", "BOL", "CAN", "DEN", "ETH", "FIN", "GER", "HUN", "IRE", "JAP", "KEN"]
```

## filter

*Filter:* Filter out items which full fill filtering conditions and return a new array.

```
//Filter countries containing land
const countriesContainingLand = countries.filter((country) =>
  country.includes('land')
)
console.log(countriesContainingLand)
```

```
['Finland', 'Ireland']
```

```
const countriesEndsByia = countries.filter((country) => country.endsWith('ia'))
console.log(countriesEndsByia)
```

```
['Albania', 'Bolivia', 'Ethiopia']
```

```
const countriesHaveFiveLetters = countries.filter(
  (country) => country.length === 5
)
console.log(countriesHaveFiveLetters)
```

```
['Japan', 'Kenya']
```

```
const scores = [
  { name: 'Asabeneh', score: 95 },
  { name: 'Lidiya', score: 98 },
  { name: 'Mathias', score: 80 },
  { name: 'Elias', score: 50 },
  { name: 'Martha', score: 85 },
  { name: 'John', score: 100 },
]
```

```
const scoresGreaterEighty = scores.filter((score) => score.score > 80)
console.log(scoresGreaterEighty)
```

```
[{name: 'Asabeneh', score: 95}, { name: 'Lidiya', score: 98 }, {name: 'Martha', score: 85}, {nam
```

## reduce

*reduce*: Reduce takes a callback function. The call back function takes accumulator, current, and optional initial value as a parameter and returns a single value. It is a good practice to define an initial value for the accumulator value. If we do not specify this parameter, by default accumulator will get array first value . If our array is an *empty array*, then Javascript will throw an error.

```
arr.reduce((acc, cur) => {  
  // some operations goes here before returning a value  
  return  
, initialValue)
```

```
const numbers = [1, 2, 3, 4, 5]  
const sum = numbers.reduce((acc, cur) => acc + cur, 0)
```

```
console.log(sum)
```

15

## every

*every*: Check if all the elements are similar in one aspect. It returns boolean

```
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']  
const areAllStr = names.every((name) => typeof name === 'string') // Are all strings?
```

```
console.log(areAllStr)
```

true

```
const bools = [true, true, true, true]  
const areAllTrue = bools.every((b) => b === true) // Are all true?
```

```
console.log(areAllTrue) // true
```

true



## find

*find*: Return the first element which satisfies the condition

```
const ages = [24, 22, 25, 32, 35, 18]
const age = ages.find((age) => age < 20)
```

```
console.log(age)
```

18

```
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']
const result = names.find((name) => name.length > 7)
console.log(result)
```

Asabeneh

```
const scores = [
  { name: 'Asabeneh', score: 95 },
  { name: 'Mathias', score: 80 },
  { name: 'Elias', score: 50 },
  { name: 'Martha', score: 85 },
  { name: 'John', score: 100 },
]

const score = scores.find((user) => user.score > 80)
console.log(score)
```

```
{ name: "Asabeneh", score: 95 }
```

## findIndex

*findIndex*: Return the position of the first element which satisfies the condition

```
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']
const ages = [24, 22, 25, 32, 35, 18]

const result = names.findIndex((name) => name.length > 7)
console.log(result) // 0
```

```
const age = ages.findIndex((age) => age < 20)
console.log(age) // 5
```

## some

*some*: Check if some of the elements are similar in one aspect. It returns boolean

```
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']
const bools = [true, true, true, true]

const areSomeTrue = bools.some((b) => b === true)

console.log(areSomeTrue) //true

const areAllStr = names.some((name) => typeof name === 'number') // Are all strings ?
console.log(areAllStr) // false
```

## sort

*sort*: The sort methods arranges the array elements either ascending or descending order. By default, the **sort()** method sorts values as strings. This works well for string array items but not for numbers. If number values are sorted as strings and it give us wrong result. Sort method modify the original array. It is recommended to copy the original data before you start using *sort* method.

### Sorting string values

```
const products = ['Milk', 'Coffee', 'Sugar', 'Honey', 'Apple', 'Carrot']
console.log(products.sort()) // ['Apple', 'Carrot', 'Coffee', 'Honey', 'Milk', 'Sugar']
//Now the original products array is also sorted
```

### Sorting Numeric values

As you can see in the example below, 100 came first after sorted in ascending order. Sort converts items to string, since '100' and other numbers compared, 1 which the beginning of the string '100' became the smallest. To avoid this, we use a compare call back function inside the sort method, which return a negative, zero or positive.

```
const numbers = [9.81, 3.14, 100, 37]
// Using sort method to sort number items provide a wrong result. see below
console.log(numbers.sort()) //[100, 3.14, 37, 9.81]
numbers.sort(function (a, b) {
  return a - b
})
```

```
console.log(numbers) // [3.14, 9.81, 37, 100]

numbers.sort(function (a, b) {
  return b - a
})
console.log(numbers) //[100, 37, 9.81, 3.14]
```

## Sorting Object Arrays

Whenever we sort objects in an array, we use the object key to compare. Let us see the example below.

```
objArr.sort(function (a, b) {
  if (a.key < b.key) return -1
  if (a.key > b.key) return 1
  return 0
})

// or

objArr.sort(function (a, b) {
  if (a['key'] < b['key']) return -1
  if (a['key'] > b['key']) return 1
  return 0
})

const users = [
  { name: 'Asabeneh', age: 150 },
  { name: 'Brook', age: 50 },
  { name: 'Eyob', age: 100 },
  { name: 'Elias', age: 22 },
]
users.sort((a, b) => {
  if (a.age < b.age) return -1
  if (a.age > b.age) return 1
  return 0
})
console.log(users) // sorted ascending
// [{...}, {...}, {...}, {...}]
```

🟡 You are doing great. Never give up because great things take time. You have just completed day 9 challenges and you are 9 steps a head in to your way to greatness. Now do some exercises for your brain and for your muscle.



## Exercises

### Exercises: Level 1

```
const countries = ['Finland', 'Sweden', 'Denmark', 'Norway', 'IceLand']
const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const products = [
  { product: 'banana', price: 3 },
  { product: 'mango', price: 6 },
  { product: 'potato', price: ' ' },
  { product: 'avocado', price: 8 },
  { product: 'coffee', price: 10 },
  { product: 'tea', price: ' ' },
]
```

1. Explain the difference between **forEach**, **map**, **filter**, and **reduce**.
2. Define a callback function before you use it in **forEach**, **map**, **filter** or **reduce**.
3. Use **forEach** to console.log each country in the countries array.
4. Use **forEach** to console.log each name in the names array.
5. Use **forEach** to console.log each number in the numbers array.
6. Use **map** to create a new array by changing each country to uppercase in the countries array.
7. Use **map** to create an array of countries length from countries array.
8. Use **map** to create a new array by changing each number to square in the numbers array
9. Use **map** to change to each name to uppercase in the names array
10. Use **map** to map the products array to its corresponding prices.
11. Use **filter** to filter out countries containing **land**.
12. Use **filter** to filter out countries having six character.
13. Use **filter** to filter out countries containing six letters and more in the country array.
14. Use **filter** to filter out country start with 'E';
15. Use **filter** to filter out only prices with values.
16. Declare a function called **getStringLists** which takes an array as a parameter and then returns an array only with string items.
17. Use **reduce** to sum all the numbers in the numbers array.
18. Use **reduce** to concatenate all the countries and to produce this sentence: **Estonia, Finland, Sweden, Denmark, Norway, and IceLand are north European countries**
19. Explain the difference between **some** and **every**
20. Use **some** to check if some names' length greater than seven in names array
21. Use **every** to check if all the countries contain the word land
22. Explain the difference between **find** and **findIndex**.
23. Use **find** to find the first country containing only six letters in the countries array
24. Use **findIndex** to find the position of the first country containing only six letters in the countries array
25. Use **findIndex** to find the position of **Norway** if it doesn't exist in the array you will get -1.

26. Use *findIndex* to find the position of *Russia* if it doesn't exist in the array you will get -1.

## Exercises: Level 2

1. Find the total price of products by chaining two or more array iterators(eg. `arr.map(callback).filter(callback).reduce(callback)`)
2. Find the sum of price of products using only `reduce` `reduce(callback)`
3. Declare a function called *categorizeCountries* which returns an array of countries which have some common pattern(you find the countries array in this repository as `countries.js`(eg 'land', 'ia', 'island','stan')).
4. Create a function which return an array of objects, which is the letter and the number of times the letter use to start with a name of a country.
5. Declare a *getFirstTenCountries* function and return an array of ten countries. Use different functional programming to work on the `countries.js` array
6. Declare a *getLastTenCountries* function which which returns the last ten countries in the `countries` array.
7. Find out which *letter* is used many *times* as initial for a country name from the `countries` array (eg. Finland, Fiji, France etc)

## Exercises: Level 3

1. Use the countries information, in the data folder. Sort countries by name, by capital, by population
2. \*\*\* Find the 10 most spoken languages:

```
// Your output should look like this
console.log(mostSpokenLanguages(countries, 10))
[
  {country: 'English',count:91},
  {country: 'French',count:45},
  {country: 'Arabic',count:25},
  {country: 'Spanish',count:24},
  {country: 'Russian',count:9},
  {country: 'Portuguese', count:9},
  {country: 'Dutch',count:8},
  {country: 'German',count:7},
  {country: 'Chinese',count:5},
  {country: 'Swahili',count:4}
]
```

```
// Your output should look like this
console.log(mostSpokenLanguages(countries, 3))
[
  {country: 'English',count: 91},
  {country: 'French',count: 45},
  {country: 'Arabic',count: 25},
]
```

```
]` ``
```

### 3. \*\*\* Use countries\_data.js file create a function which create the ten most populated countries

```
console.log(mostPopulatedCountries(countries, 10))

[
  {country: 'China', population: 1377422166},
  {country: 'India', population: 1295210000},
  {country: 'United States of America', population: 323947000},
  {country: 'Indonesia', population: 258705000},
  {country: 'Brazil', population: 206135893},
  {country: 'Pakistan', population: 194125062},
  {country: 'Nigeria', population: 186988000},
  {country: 'Bangladesh', population: 161006790},
  {country: 'Russian Federation', population: 146599183},
  {country: 'Japan', population: 126960000}
]

console.log(mostPopulatedCountries(countries, 3))
[
  {country: 'China', population: 1377422166},
  {country: 'India', population: 1295210000},
  {country: 'United States of America', population: 323947000}
]
` ``
```

### 4. \*\*\* Try to develop a program which calculate measure of central tendency of a sample(mean, median, mode) and measure of variability(range, variance, standard deviation). In addition to those measures find the min, max, count, percentile, and frequency distribution of the sample. You can create an object called statistics and create all the functions which do statistical calculations as method for the statistics object. Check the output below.

```
const ages = [31, 26, 34, 37, 27, 26, 32, 32, 26, 27, 27, 24, 32, 33, 27, 25, 26, 38, 37,

console.log('Count:', statistics.count()) // 25
console.log('Sum: ', statistics.sum()) // 744
console.log('Min: ', statistics.min()) // 24
console.log('Max: ', statistics.max()) // 38
console.log('Range: ', statistics.range()) // 14
console.log('Mean: ', statistics.mean()) // 30
console.log('Median: ', statistics.median()) // 29
console.log('Mode: ', statistics.mode()) // {'mode': 26, 'count': 5}
console.log('Variance: ', statistics.var()) // 17.5
console.log('Standard Deviation: ', statistics.std()) // 4.2
```

```
console.log('Variance: ',statistics.var()) // 17.5
console.log('Frequency Distribution: ',statistics.freqDist()) # [(20.0, 26), (16.0, 27), (12.0, 32), (8.0, 37), (8.0, 34), (8.0, 3)]

console.log(statistics.describe())
Count: 25
Sum: 744
Min: 24
Max: 38
Range: 14
Mean: 30
Median: 29
Mode: (26, 5)
Variance: 17.5
Standard Deviation: 4.2
Frequency Distribution: [(20.0, 26), (16.0, 27), (12.0, 32), (8.0, 37), (8.0, 34), (8.0, 3)]
```

🎉 CONGRATULATIONS ! 🎉

<< [Day 8](#) | [Day 10](#) >>