Day 9 - Higher Order Functions

HOF’s are functions that take another function as a parameter and return a function as a value. A function passed as a parameter is called a callback function.

Callback

A function which can be passed as a parameter to another function.

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

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

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

Returning function

HOF’s that return a function as a value

```
// Higher order function returning another 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))
```

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))

// Simplified
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 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

The setInterval higher order functions allows us to do some activity continuously within some interval of time. Takes a callback function and a duration as a parameter. Duration is in milliseconds and 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) // Prints hello in every second (1000 ms is 1 second)
```

Setting a time using a setTimeout

setTimeout is a higher order function in JS to execute some action as some time in the future. The setTimeout global method takes a callback function and a duration as a parameter. Duration is in milliseconds and the callback will 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) // prints Hello to the console after 2 seconds
```

Functional Programming

The latest version of JS introduced lots of built in methods which can help us solve complicated problems. All built-in methods take a callback function.

forEach

forEach: Iterates array elements. Only used with arrays. Takes a callback function with elements, index parameter and array itself. Index and the array are optional.

```
arr.forEach(function (element, index, arr) {
  console.log(index, element, arr)
})
// Above code can be written using arrow function notation
arr.forEach((element, index, arr) => {
  console.log(index, element, arr)
}
// 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(num) //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 array elements and modify the array elements. Takes a callback function with elements, index, array parameter and returns 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 = ['Chris', 'Alex', 'Cinthia']
const namesToUpperCase = names.map((name) => name.toUpperCase())
console.log(namesToUpperCase)
```

filter

filter: Filter out items which fulfill filtering conditions and return a new array.

```
//Filter countries containing land
const countriesContainingLand = countries.filter((country) =>
  country.includes('land')
)
// ['Finland', 'Ireland']
```

```
const countriesEndsByIa = countries.filter((country) =>
  country.endsWith('ia'))
)
// ['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},{name: 'John', score: 100}]
```

reduce

reduce: Takes a callback function. Callback function takes accumulator, current, and optional initial value as a parameter and returns a single value. Good practice to define an initial value for the accumulator value. If accumulator parameter not specified, by default accumulator will get array's first value. If array is an empty array, then JavaScript will throw an error.

```
arr.reduce((acc, cur) => {
  // some operations go 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. Returns a Boolean.

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

console.log(areAllStr)

const bools = [true, true, true, true]
const areAllTrue = bools.every((bool) => typeof bool === true)

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

find

find: Returns 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 = ['Chris', 'Alex', 'Cinthia']
const result = names.find((name) => name.length > 5)

console.log(result) // Cinthia

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: Returns the position of the first element which satisfies the condition.

```
const names = ['Chris', 'Alex']
const ages = [24, 19]

const result = names.findIndex((name) => name.length > 4)
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 = ['Chris', 'Cinthia']
const bools = [true, true, true, true]

const areSomeTrue = bools.some((b) => b === true)
console.log(areSomeTrue) // true

const areSomeStr = names.some((name) => typeof name === 'number')
console.log(areSomeStr) // false
```

sort

sort: Sort method arranges array elements either ascending or descending order. By default, sort() sorts values as strings. This works well for string array items but not numbers. If number values are sorted as strings, it gives the wrong result. Sort modifies the original array. 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

In the example below, 100 came first after sorting in ascending order. Sort converts the items to string, and since the 1 in 100 is the smaller than the other numbers, ‘100’ becomes the smallest. To avoid this, we use a compare call back function inside the sort method, which returns a negative, zero, or positive.

```
const numbers = [9.81, 3.14, 100, 37]
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.

```
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
// [{…}, {…}, {…}, {…}]
```