# 30 Days Of React: JavaScript Refresher

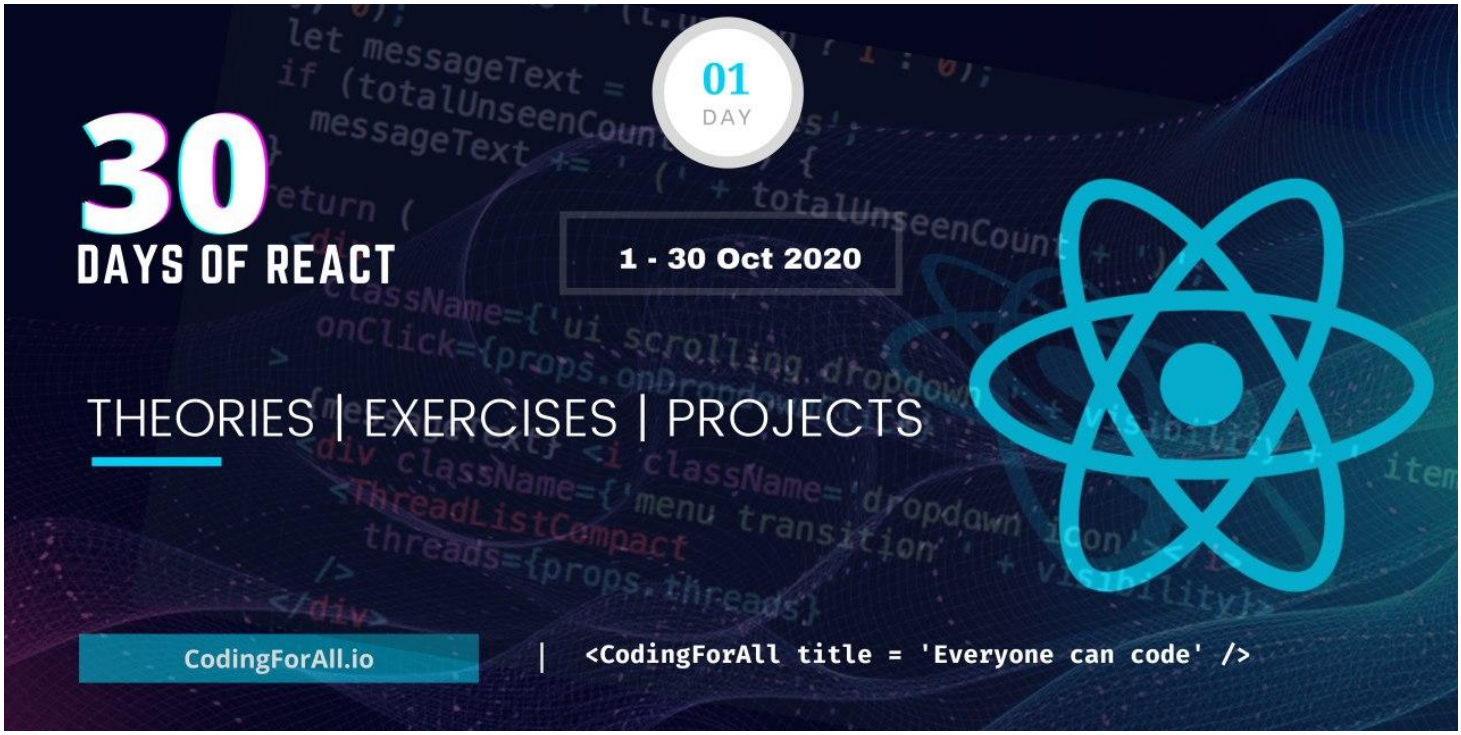
Twitter Follow

Author:

Asabeneh Yetayeh

October, 2020

<< Day 0 | Day 2 >>



# JavaScript Refresher

## 0. Adding JavaScript to a Web Page

JavaScript can be added to a web page in three different ways:

- ***Inline script***
- ***Internal script***
- ***External script***
- ***Multiple External scripts***

The following sections show different ways of adding JavaScript code to your web page.

### Inline Script

Create a project folder on your desktop or in any location, name it 30DaysOfJS and create an ***index.html*** file in the project folder. Then paste the following code and open it in a browser, for example Chrome.

```
<!DOCTYPE html>
<html>
  <head>
    <title>30DaysOfScript:Inline Script</title>
  </head>
  <body>
    <button onclick="alert('Welcome to 30DaysOfJavaScript!')">Click Me</button>
  </body>
</html>
```

Now, you just wrote your first inline script. We can create a pop up alert message using the *alert()* built-in function.

## Internal Script

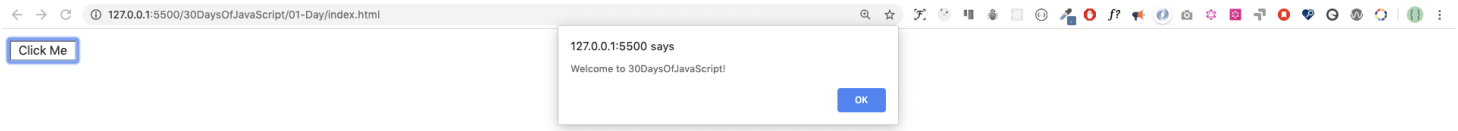
The internal script can be written in the *head* or the *body*, but it is preferred to put it on the body of the HTML document.
First, let us write on the head part of the page.

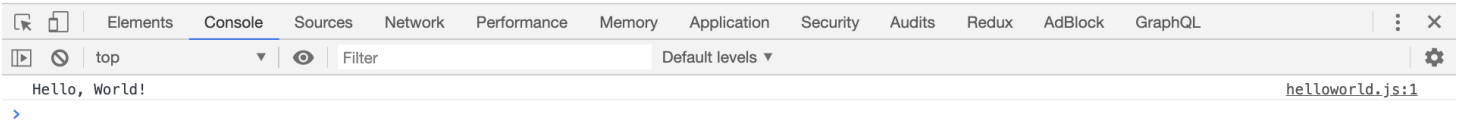
```
<!DOCTYPE html>
<html>
  <head>
    <title>30DaysOfScript:Internal Script</title>
    <script>
      console.log('Welcome to 30DaysOfJavaScript')
    </script>
  </head>
  <body></body>
</html>
```

This is how we write an internal script most of the time. Writing the JavaScript code in the body section is the most preferred option. Open the browser console to see the output from the console.log()

```
<!DOCTYPE html>
<html>
  <head>
    <title>30DaysOfScript:Internal Script</title>
  </head>
  <body>
    <button onclick="alert('Welcome to 30DaysOfJavaScript!');">Click Me</button>
    <script>
      console.log('Welcome to 30DaysOfJavaScript')
    </script>
  </body>
</html>
```

Open the browser console to see the output from the console.log()




## External Script

Similar to the internal script, the external script link can be on the header or body, but it is preferred to put it in the body.
First, we should create an external JavaScript file with .js extension. All files ending with .js extension. All files ending with .js extension are JavaScript files. Create a file named introduction.js inside your project directory and write the following code and link this .js file at the bottom of the body.

```
console.log('Welcome to 30DaysOfJavaScript')
```

External scripts in the *head*:

```
<!DOCTYPE html>
<html>
  <head>
    <title>30DaysOfJavaScript:External script</title>
    <script src="introduction.js"></script>
  </head>
  <body></body>
</html>
```

External scripts in the *body*:

```
<!DOCTYPE html>
<html>
  <head>
    <title>30DaysOfJavaScript:External script</title>
  </head>
  <body>
    //it could be in the header or in the body // Here is the recommended place
    to put the external script
    <script src="introduction.js"></script>
  </body>
</html>
```

Open the browser console to see the output of the console.log()

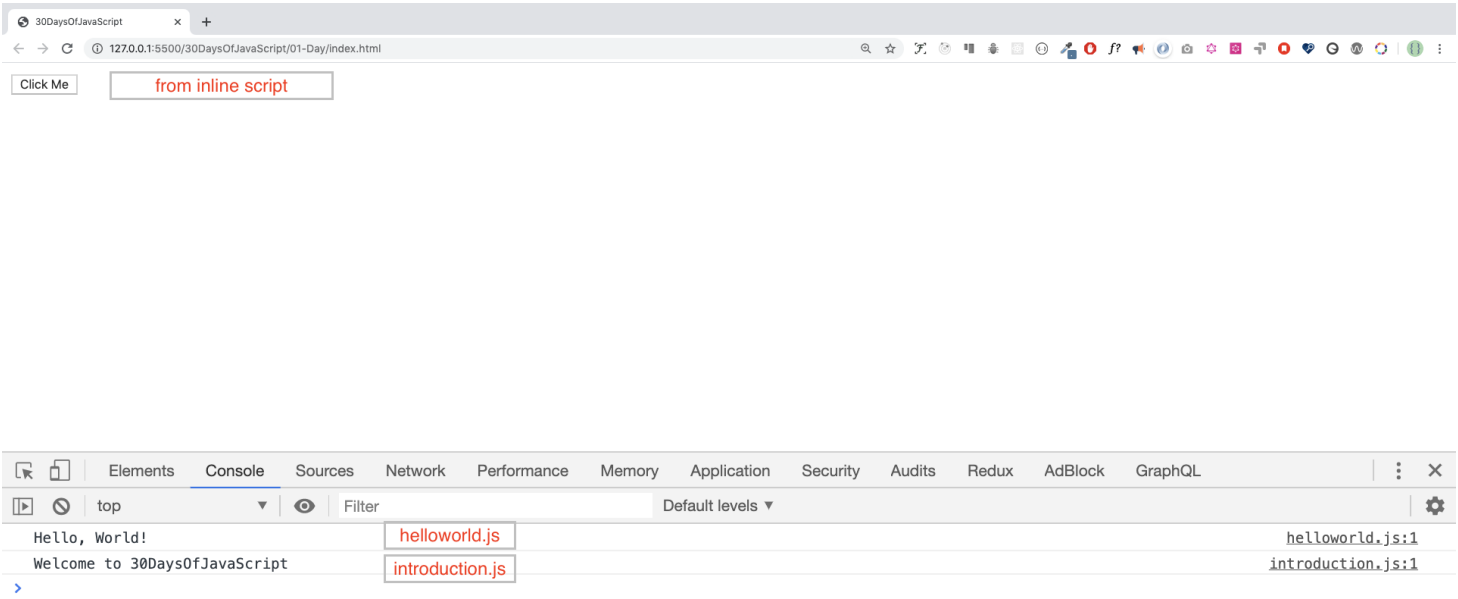
## Multiple External Scripts

We can also link multiple external JavaScript files to a web page.
Create a helloworld.js file inside the 30DaysOfJS folder and write the following code.

```
console.log('Hello, World!')
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Multiple External Scripts</title>
  </head>
  <body>
    <script src="./helloworld.js"></script>
    <script src="./introduction.js"></script>
  </body>
</html>
```

*Your main.js file should be below all other scripts*. It is very important to remember this.




# 1. Variables

We use *var*, *let* and *const* to declare a variable. The *var* is functions scope, however *let* and *const* are block scope. In this challenge we use ES6 and above features of JavaScript. Avoid using *var*.

```
let firstName = 'Asabeneh'
firstName = 'Eyob'

const PI = 3.14 // Not allowed to reassign PI to a new value
// PI = 3.
```

# 2. Data types

If you do not feel comfortable with data types check the following link

# 3. Arrays

In contrast to variables, an array can store *multiple values*. Each value in an array has an *index*, and each index has *a reference in a memory address*. Each value can be accessed by using their *indexes*. The index of an array starts from *zero*, and the index of the last element is less by one from the length of the array.

An array is a collection of different data types which are ordered and changeable(modifiable). An array allows storing duplicate elements and different data types. An array can be empty, or it may have different data type values.

## How to create an empty array

In JavaScript, we can create an array in different ways. Let us see different ways to create an array. It is very common to use *const* instead of *let* to declare an array variable. If you are using const it means you do not use that variable name again.

- Using Array constructor

```
// syntax
const arr = Array()
// or
// let arr = new Array()
console.log(arr) // []
```

- Using square brackets([])

```
// syntax
// This the most recommended way to create an empty list
const arr = []
console.log(arr)
```

## How to create an array with values

Array with initial values. We use *length* property to find the length of an array.

```
const numbers = [0, 3.14, 9.81, 37, 98.6, 100] // array of numbers
const fruits = ['banana', 'orange', 'mango', 'lemon'] // array of strings, fruits
const vegetables = ['Tomato', 'Potato', 'Cabbage', 'Onion', 'Carrot'] // array of strings, veget
const animalProducts = ['milk', 'meat', 'butter', 'yoghurt'] // array of strings, products
const webTechs = ['HTML', 'CSS', 'JS', 'React', 'Redux', 'Node', 'MongDB'] // array of web techr
const countries = ['Finland', 'Denmark', 'Sweden', 'Norway', 'Iceland'] // array of strings, cou

// Print the array and its length

console.log('Numbers:', numbers)
console.log('Number of numbers:', numbers.length)

console.log('Fruits:', fruits)
console.log('Number of fruits:', fruits.length)

console.log('Vegetables:', vegetables)
console.log('Number of vegetables:', vegetables.length)

console.log('Animal products:', animalProducts)
console.log('Number of animal products:', animalProducts.length)

console.log('Web technologies:', webTechs)
console.log('Number of web technologies:', webTechs.length)

console.log('Countries:', countries)
console.log('Number of countries:', countries.length)


Numbers: [0, 3.14, 9.81, 37, 98.6, 100]
Number of numbers: 6
Fruits: ['banana', 'orange', 'mango', 'lemon']
Number of fruits: 4
Vegetables: ['Tomato', 'Potato', 'Cabbage', 'Onion', 'Carrot']
Number of vegetables: 5
Animal products: ['milk', 'meat', 'butter', 'yoghurt']
Number of animal products: 4
Web technologies: ['HTML', 'CSS', 'JS', 'React', 'Redux', 'Node', 'MongDB']
Number of web technologies: 7
Countries: ['Finland', 'Estonia', 'Denmark', 'Sweden', 'Norway']
Number of countries: 5
```

- Array can have items of different data types

```
const arr = [
  'Asabeneh',
  250,
  true,
  { country: 'Finland', city: 'Helsinki' },
  { skills: ['HTML', 'CSS', 'JS', 'React', 'Python'] },
] // arr containing different data types
console.log(arr)
```

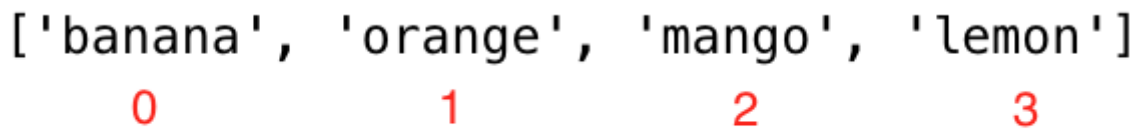
## Creating an array using split

As we have seen in the earlier section, we can split a string at different positions, and we can change to an array. Let us see the examples below.

```
let js = 'JavaScript'
const charsInJavaScript = js.split('')

console.log(charsInJavaScript) // ["J", "a", "v", "a", "S", "c", "r", "i", "p", "t"]

let companiesString = 'Facebook, Google, Microsoft, Apple, IBM, Oracle, Amazon'
const companies = companiesString.split(',')

console.log(companies) // ["Facebook", " Google", " Microsoft", " Apple", " IBM", " Oracle", " A
let txt =
  'I love teaching and empowering people. I teach HTML, CSS, JS, React, Python.'
const words = txt.split(' ')

console.log(words)
// the text has special characters think how you can just get only the words
// ["I", "love", "teaching", "and", "empowering", "people.", "I", "teach", "HTML,", "CSS,", "JS,
```

## Accessing array items using index

We access each element in an array using their index. An array index starts from 0. The picture below clearly shows the index of each element in the array.



```
const fruits = ['banana', 'orange', 'mango', 'lemon']
let firstFruit = fruits[0] // we are accessing the first item using its index

console.log(firstFruit) // banana

secondFruit = fruits[1]
console.log(secondFruit) // orange

let lastFruit = fruits[3]
console.log(lastFruit) // lemon
// Last index can be calculated as follows

let lastIndex = fruits.length - 1
lastFruit = fruits[lastIndex]

console.log(lastFruit) // lemon


const numbers = [0, 3.14, 9.81, 37, 98.6, 100] // set of numbers

console.log(numbers.length) // => to know the size of the array, which is 6
console.log(numbers) // -> [0, 3.14, 9.81, 37, 98.6, 100]
console.log(numbers[0]) //  -> 0
console.log(numbers[5]) //  -> 100

let lastIndex = numbers.length - 1
console.log(numbers[lastIndex]) // -> 100


const webTechs = [
  'HTML',
  'CSS',
  'JavaScript',
  'React',
  'Redux',
  'Node',
  'MongoDB',
] // List of web technologies

console.log(webTechs) // all the array items
console.log(webTechs.length) // => to know the size of the array, which is 7
console.log(webTechs[0]) //  -> HTML
console.log(webTechs[6]) //  -> MongoDB

let lastIndex = webTechs.length - 1
console.log(webTechs[lastIndex]) // -> MongoDB
```

```
const countries = [
  'Albania',
  'Bolivia',
  'Canada',
  'Denmark',
  'Ethiopia',
  'Finland',
  'Germany',
  'Hungary',
  'Ireland',
  'Japan',
  'Kenya',
] // List of countries

console.log(countries) // -> all countries in array
console.log(countries[0]) //  -> Albania
console.log(countries[10]) //  -> Kenya

let lastIndex = countries.length - 1
console.log(countries[lastIndex]) //  -> Kenya


const shoppingCart = [
  'Milk',
  'Mango',
  'Tomato',
  'Potato',
  'Avocado',
  'Meat',
  'Eggs',
  'Sugar',
] // List of food products

console.log(shoppingCart) // -> all shoppingCart in array
console.log(shoppingCart[0]) //  -> Milk
console.log(shoppingCart[7]) //  -> Sugar

let lastIndex = shoppingCart.length - 1
console.log(shoppingCart[lastIndex]) //  -> Sugar
```

## Modifying array element

An array is mutable(modifiable). Once an array is created, we can modify the contents of the array elements.

```
const numbers = [1, 2, 3, 4, 5]
numbers[0] = 10 // changing 1 at index 0 to 10
numbers[1] = 20 // changing  2 at index 1 to 20

console.log(numbers) // [10, 20, 3, 4, 5]

const countries = [
  'Albania',
  'Bolivia',
  'Canada',
  'Denmark',
  'Ethiopia',
  'Finland',
  'Germany',
  'Hungary',
  'Ireland',
  'Japan',
  'Kenya',
]

countries[0] = 'Afghanistan' // Replacing Albania by Afghanistan
let lastIndex = countries.length - 1
countries[lastIndex] = 'Korea' // Replacing Kenya by Korea

console.log(countries)


["Afghanistan", "Bolivia", "Canada", "Denmark", "Ethiopia", "Finland", "Germany", "Hungary", "Ir
```

## Methods to manipulate array

There are different methods to manipulate an array. These are some of the available methods to deal with arrays:*Array, length, concat, indexOf, slice, splice, join, toString, includes, lastIndexOf, isArray, fill, push, pop, shift, unshift*

### Array Constructor

Array:To create an array.

```
const arr = Array() // creates an an empty array
console.log(arr)

const eightEmptyValues = Array(8) // it creates eight empty values
console.log(eightEmptyValues) // [empty x 8]
```

### Creating static values with fill

fill: Fill all the array elements with a static value

```
const arr = Array() // creates an an empty array
console.log(arr)

const eightXvalues = Array(8).fill('X') // it creates eight element values filled with 'X'
console.log(eightXvalues) // ['X', 'X','X','X','X','X','X','X']

const eight0values = Array(8).fill(0) // it creates eight element values filled with '0'
console.log(eight0values) // [0, 0, 0, 0, 0, 0, 0, 0]

const four4values = Array(4).fill(4) // it creates 4 element values filled with '4'
console.log(four4values) // [4, 4, 4, 4]
```

### Concatenating array using concat

concat:To concatenate two arrays.

```
const firstList = [1, 2, 3]
const secondList = [4, 5, 6]
const thirdList = firstList.concat(secondList)

console.log(thirdList) // [1, 2, 3, 4, 5, 6]


const fruits = ['banana', 'orange', 'mango', 'lemon'] // array of fruits
const vegetables = ['Tomato', 'Potato', 'Cabbage', 'Onion', 'Carrot'] // array of vegetables
const fruitsAndVegetables = fruits.concat(vegetables) // concatenate the two arrays

console.log(fruitsAndVegetables)


["banana", "orange", "mango", "lemon", "Tomato", "Potato", "Cabbage", "Onion", "Carrot"]
```

### Getting array length

Length:To know the size of the array

```
const numbers = [1, 2, 3, 4, 5]
console.log(numbers.length) // -> 5 is the size of the array
```

### Getting index of an element in an array

indexOf:To check if an item exist in an array. If it exists it returns the index else it returns -1.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers.indexOf(5)) // -> 4
console.log(numbers.indexOf(0)) // -> -1
console.log(numbers.indexOf(1)) // -> 0
console.log(numbers.indexOf(6)) // -> -1
```

Check an element if it exist in an array.

- Check items in a list

```
// let us check if a banana exist in the array

const fruits = ['banana', 'orange', 'mango', 'lemon']
let index = fruits.indexOf('banana') // 0

if (index != -1) {
  console.log('This fruit does exist in the array')
} else {
  console.log('This fruit does not exist in the array')
}
// This fruit does exist in the array

// we can use also ternary here
index != -1
  ? console.log('This fruit does exist in the array')
  : console.log('This fruit does not exist in the array')

// let us check if a avocado exist in the array
let indexOfAvocado = fruits.indexOf('avocado') // -1, if the element not found index is -1
if (indexOfAvocado != -1) {
  console.log('This fruit does exist in the array')
} else {
  console.log('This fruit does not exist in the array')
}
// This fruit does not exist in the array
```

### Getting last index of an element in array

lastIndexOf: It gives the position of the last item in the array. If it exist, it returns the index else it returns -1.

```
const numbers = [1, 2, 3, 4, 5, 3, 1, 2]

console.log(numbers.lastIndexOf(2)) // 7
console.log(numbers.lastIndexOf(0)) // -1
console.log(numbers.lastIndexOf(1)) //  6
console.log(numbers.lastIndexOf(4)) //  3
console.log(numbers.lastIndexOf(6)) // -1
```

includes:To check if an item exist in an array. If it exist it returns the true else it returns false.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers.includes(5)) // true
console.log(numbers.includes(0)) // false
console.log(numbers.includes(1)) // true
console.log(numbers.includes(6)) // false

const webTechs = [
  'HTML',
  'CSS',
  'JavaScript',
  'React',
  'Redux',
  'Node',
  'MongoDB',
] // List of web technologies

console.log(webTechs.includes('Node')) // true
console.log(webTechs.includes('C')) // false
```

### Checking array

Array.isArray:To check if the data type is an array

```
const numbers = [1, 2, 3, 4, 5]
console.log(Array.isArray(numbers)) // true

const number = 100
console.log(Array.isArray(number)) // false
```

### Converting array to string

toString:Converts array to string

```
const numbers = [1, 2, 3, 4, 5]
console.log(numbers.toString()) // 1,2,3,4,5

const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']
console.log(names.toString()) // Asabeneh,Mathias,Elias,Brook
```

### Joining array elements

join: It is used to join the elements of the array, the argument we passed in the join method will be joined in the array and return as a string. By default, it joins with a comma, but we can pass different string parameter which can be joined between the items.

```
const numbers = [1, 2, 3, 4, 5]
console.log(numbers.join()) // 1,2,3,4,5

const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook']

console.log(names.join()) // Asabeneh,Mathias,Elias,Brook
console.log(names.join('')) //AsabenehMathiasEliasBrook
console.log(names.join(' ')) //Asabeneh Mathias Elias Brook
console.log(names.join(', ')) //Asabeneh, Mathias, Elias, Brook
console.log(names.join(' # ')) //Asabeneh # Mathias # Elias # Brook

const webTechs = [
  'HTML',
  'CSS',
  'JavaScript',
  'React',
  'Redux',
  'Node',
  'MongoDB',
] // List of web technologies

console.log(webTechs.join()) // "HTML,CSS,JavaScript,React,Redux,Node,MongoDB"
console.log(webTechs.join(' # ')) // "HTML # CSS # JavaScript # React # Redux # Node # MongoDB"
```

### Slice array elements

Slice: To cut out a multiple items in range. It takes two parameters:starting and ending position. It doesn't include the ending position.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers.slice()) // -> it copies all  item
console.log(numbers.slice(0)) // -> it copies all  item
console.log(numbers.slice(0, numbers.length)) // it copies all  item
console.log(numbers.slice(1, 4)) // -> [2,3,4] // it doesn't include the ending position
```

### Splice method in array

Splice: It takes three parameters:Starting position, number of times to be removed and number of items to be added.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers.splice()) // -> remove all items


const numbers = [1, 2, 3, 4, 5]
console.log(numbers.splice(0, 1)) // remove the first item


const numbers = [1, 2, 3, 4, 5, 6]
console.log(numbers.splice(3, 3, 7, 8, 9)) // -> [1, 2, 3, 7, 8, 9] //it removes three item and
```

### Adding item to an array using push

Push: adding item in the end. To add item to the end of an existing array we use the push method.

```
// syntax
const arr = ['item1', 'item2', 'item3']
arr.push('new item')

console.log(arr)
// ['item1', 'item2','item3','new item']


const numbers = [1, 2, 3, 4, 5]
numbers.push(6)

console.log(numbers) // -> [1,2,3,4,5,6]

numbers.pop() // -> remove one item from the end
console.log(numbers) // -> [1,2,3,4,5]


let fruits = ['banana', 'orange', 'mango', 'lemon']
fruits.push('apple')

console.log(fruits) // ['banana', 'orange', 'mango', 'lemon', 'apple']

fruits.push('lime')
console.log(fruits) // ['banana', 'orange', 'mango', 'lemon', 'apple', 'lime']
```

### Removing the end element using pop

pop: Removing item in the end.

```
const numbers = [1, 2, 3, 4, 5]
numbers.pop() // -> remove one item from the end

console.log(numbers) // -> [1,2,3,4]
```

### Removing an element from the beginning

shift: Removing one array element in the beginning of the array.

```
const numbers = [1, 2, 3, 4, 5]
numbers.shift() // -> remove one item from the beginning

console.log(numbers) // -> [2,3,4,5]
```

### Add an element from the beginning

unshift: Adding array element in the beginning of the array.

```
const numbers = [1, 2, 3, 4, 5]
numbers.unshift(0) // -> add one item from the beginning

console.log(numbers) // -> [0,1,2,3,4,5]
```

### Reversing array order

reverse: reverse the order of an array.

```
const numbers = [1, 2, 3, 4, 5]
numbers.reverse() // -> reverse array order

console.log(numbers) // [5, 4, 3, 2, 1]

numbers.reverse()
console.log(numbers) // [1, 2, 3, 4, 5]
```

### Sorting elements in array

sort: arrange array elements in ascending order. Sort takes a call back function, we will see how we use sort with a call back function in the coming sections.

```
const webTechs = [
  'HTML',
  'CSS',
  'JavaScript',
  'React',
  'Redux',
  'Node',
  'MongoDB',
]

webTechs.sort()
console.log(webTechs) // ["CSS", "HTML", "JavaScript", "MongoDB", "Node", "React", "Redux"]

webTechs.reverse() // after sorting we can reverse it
console.log(webTechs) // ["Redux", "React", "Node", "MongoDB", "JavaScript", "HTML", "CSS"]
```

## Array of arrays

Array can store different data types including an array itself. Let us create an array of arrays

```
const firstNums = [1, 2, 3]
const secondNums = [1, 4, 9]

const arrayOfArray = [
  [1, 2, 3],
  [1, 2, 3],
]
console.log(arrayOfArray[0]) // [1, 2, 3]

const frontEnd = ['HTML', 'CSS', 'JS', 'React', 'Redux']
const backEnd = ['Node', 'Express', 'MongoDB']
const fullStack = [frontEnd, backEnd]
console.log(fullStack) // [["HTML", "CSS", "JS", "React", "Redux"], ["Node", "Express", "MongoDE
console.log(fullStack.length) // 2
console.log(fullStack[0]) // ["HTML", "CSS", "JS", "React", "Redux"]
console.log(fullStack[1]) // ["Node", "Express", "MongoDB"]
```

# 💻 Exercise

### Exercise: Level 1

```
const countries = [
  'Albania',
  'Bolivia',
  'Canada',
  'Denmark',
  'Ethiopia',
  'Finland',
  'Germany',
  'Hungary',
  'Ireland',
  'Japan',
  'Kenya',
]

const webTechs = [
  'HTML',
  'CSS',
  'JavaScript',
  'React',
  'Redux',
  'Node',
  'MongoDB',
]
```

1. Declare an *empty* array;
2. Declare an array with more than 5 number of elements
3. Find the length of your array
4. Get the first item, the middle item and the last item of the array
5. Declare an array called *mixedDataTypes*, put different data types in the array and find the length of the array. The array size should be greater than 5
6. Declare an array variable name itCompanies and assign initial values Facebook, Google, Microsoft, Apple, IBM, Oracle and Amazon
7. Print the array using *console.log()*
8. Print the number of companies in the array
9. Print the first company, middle and last company
10. Print out each company
11. Change each company name to uppercase one by one and print them out
12. Print the array like as a sentence: Facebook, Google, Microsoft, Apple, IBM,Oracle and Amazon are big IT companies.

13. Check if a certain company exists in the itCompanies array. If it exist return the company else return a company is *not found*
14. Filter out companies which have more than one 'o' without the filter method
15. Sort the array using *sort()* method
16. Reverse the array using *reverse()* method
17. Slice out the first 3 companies from the array
18. Slice out the last 3 companies from the array
19. Slice out the middle IT company or companies from the array
20. Remove the first IT company from the array
21. Remove the middle IT company or companies from the array
22. Remove the last IT company from the array
23. Remove all IT companies

**Exercise: Level 2**

1. Create a separate countries.js file and store the countries array into this file, create a separate file web_techs.js and store the webTechs array into this file. Access both file in main.js file
2. First remove all the punctuations and change the string to array and count the number of words in the array

```
let text =
  'I love teaching and empowering people. I teach HTML, CSS, JS, React, Python.'
console.log(words)
console.log(words.length)
```

```
["I", "love", "teaching", "and", "empowering", "people", "I", "teach", "HTML", "CSS", "JS",
```

```
13
```

3. In the following shopping cart add, remove, edit items

```
const shoppingCart = ['Milk', 'Coffee', 'Tea', 'Honey']
```

   - add 'Meat' in the beginning of your shopping cart if it has not been already added
   - add Sugar at the end of you shopping cart if it has not been already added
   - remove 'Honey' if you are allergic to honey
   - modify Tea to 'Green Tea'
4. In countries array check if 'Ethiopia' exists in the array if it exists print 'ETHIOPIA'. If it does not exist add to the countries list.
5. In the webTechs array check if Sass exists in the array and if it exists print 'Sass is a CSS preprocess'. If it does not exist add Sass to the array and print the array.

6. Concatenate the following two variables and store it in a fullStack variable.

```
const frontEnd = ['HTML', 'CSS', 'JS', 'React', 'Redux']
const backEnd = ['Node', 'Express', 'MongoDB']

console.log(fullStack)


["HTML", "CSS", "JS", "React", "Redux", "Node", "Express", "MongoDB"]
```

**Exercise: Level 3**

1. The following is an array of 10 students ages:
   `js const ages = [19, 22, 19, 24, 20, 25, 26, 24, 25, 24]` - Sort the array and find the min and max age - Find the median age(one middle item or two middle items divided by two) - Find the average age(all items divided by number of items) - Find the range of the ages(max minus min) - Compare the value of (min - average) and (max - average), use *abs()* method
   1.Slice the first ten countries from the countries array
2. Find the middle country(ies) in the countries array
3. Divide the countries array into two equal arrays if it is even. If countries array is not even , one more country for the first half.

# 4. Conditionals

Conditional statements are used for make decisions based on different conditions.
By default , statements in JavaScript script executed sequentially from top to bottom. If the processing logic require so, the sequential flow of execution can be altered in two ways:

- Conditional execution: a block of one or more statements will be executed if a certain expression is true
- Repetitive execution: a block of one or more statements will be repetitively executed as long as a certain expression is true. In this section, we will cover *if*, *else* , *else if* statements. The comparison and logical operators we learned in the previous sections will be useful in here.

Conditions can be implementing using the following ways:

- if
- if else
- if else if else
- switch
- ternary operator

## If

In JavaScript and other programming languages the key word *if* is to used check if a condition is true and to execute the block code. To create an if condition, we need *if* keyword, condition inside a parenthesis and block of code inside a curly bracket({}).

```
// syntax
if (condition) {
  //this part of code runs for truthy condition
}
```

**Example:**

```
let num = 3
if (num > 0) {
  console.log(`${num} is a positive number`)
}
//  3 is a positive number
```

As you can see in the condition example above, 3 is greater than 0, so it is a positive number. The condition was true and the block of code was executed. However, if the condition is false, we won't see any results.

```
let isRaining = true
if (isRaining) {
  console.log('Remember to take your rain coat.')
}
```

The same goes for the second condition, if isRaining is false the if block will not be executed and we do not see any output. In order to see the result of a falsy condition, we should have another block, which is going to be *else*.

## If Else

If condition is true the first block will be executed, if not the else condition will be executed.

```
// syntax
if (condition) {
  // this part of code runs for truthy condition
} else {
  // this part of code runs for false condition
}
```

```
let num = 3
if (num > 0) {
  console.log(`${num} is a positive number`)
} else {
  console.log(`${num} is a negative number`)
}
//  3 is a positive number

num = -3
if (num > 0) {
  console.log(`${num} is a positive number`)
} else {
  console.log(`${num} is a negative number`)
}
//  -3 is a negative number


let isRaining = true
if (isRaining) {
  console.log('You need a rain coat.')
} else {
  console.log('No need for a rain coat.')
}
// You need a rain coat.

isRaining = false
if (isRaining) {
  console.log('You need a rain coat.')
} else {
  console.log('No need for a rain coat.')
}
// No need for a rain coat.
```

The last condition is false, therefore the else block was executed. What if we have more than two conditions? In that case, we would use *else if* conditions.

## If Else if Else

On our daily life, we make decisions on daily basis. We make decisions not by checking one or two conditions instead we make decisions based on multiple conditions. As similar to our daily life, programming is also full of conditions. We use *else if* when we have multiple conditions.

```
// syntax
if (condition) {
  // code
} else if (condition) {
  // code
} else {
  //  code
}
```

**Example:**

```
let a = 0
if (a > 0) {
  console.log(`${a} is a positive number`)
} else if (a < 0) {
  console.log(`${a} is a negative number`)
} else if (a == 0) {
  console.log(`${a} is zero`)
} else {
  console.log(`${a} is not a number`)
}
```

```
// if else if else
let weather = 'sunny'
if (weather === 'rainy') {
  console.log('You need a rain coat.')
} else if (weather === 'cloudy') {
  console.log('It might be cold, you need a jacket.')
} else if (weather === 'sunny') {
  console.log('Go out freely.')
} else {
  console.log('No need for rain coat.')
}
```

## Switch

Switch is an alternative for **if else if else else**.
The switch statement starts with a *switch* keyword followed by a parenthesis and code block. Inside the code block we will have different cases. Case block runs if the value in the switch statement parenthesis matches with the case value. The break statement is to terminate execution so the code execution does not go down after the condition is satisfied. The default block runs if all the cases don't satisfy the condition.

```
switch (caseValue) {
  case 1:
    // code
    break
  case 2:
    // code
    break
  case 3:
  // code
  default:
  // code
}
```

```
let weather = 'cloudy'
switch (weather) {
  case 'rainy':
    console.log('You need a rain coat.')
    break
  case 'cloudy':
    console.log('It might be cold, you need a jacket.')
    break
  case 'sunny':
    console.log('Go out freely.')
    break
  default:
    console.log(' No need for rain coat.')
}

// Switch More Examples
let dayUserInput = prompt('What day is today ?')
let day = dayUserInput.toLowerCase()

switch (day) {
  case 'monday':
    console.log('Today is Monday')
    break
  case 'tuesday':
    console.log('Today is Tuesday')
    break
  case 'wednesday':
    console.log('Today is Wednesday')
    break
  case 'thursday':
    console.log('Today is Thursday')
    break
  case 'friday':
    console.log('Today is Friday')
    break
  case 'saturday':
    console.log('Today is Saturday')
    break
  case 'sunday':
    console.log('Today is Sunday')
    break
  default:
    console.log('It is not a week day.')
}
```

// Examples to use conditions in the cases

```
let num = prompt('Enter number')
switch (true) {
  case num > 0:
    console.log('Number is positive')
    break
  case num == 0:
    console.log('Numbers is zero')
    break
  case num < 0:
    console.log('Number is negative')
    break
  default:
    console.log('Entered value was not a number')
}
```

## Ternary Operators

Ternary operator is very common in *React*. It is a short way to write if else statement. In React we use ternary operator in many cases.

To generalize, ternary operator is another way to write conditionals.

```
let isRaining = true
isRaining
  ? console.log('You need a rain coat.')
  : console.log('No need for a rain coat.')
```

# 💻 Exercises

### Exercises: Level 1

1. Get user input using prompt(“Enter your age:”). If user is 18 or older , give feedback:'You are old enough to drive' but if not 18 give another feedback stating to wait for the number of years he needs to turn 18.

   ```
   Enter your age: 30
   You are old enough to drive.

   Enter your age:15
   You are left with 3 years to drive.
   ```

2. Compare the values of myAge and yourAge using if … else. Based on the comparison and log the result to console stating who is older (me or you). Use prompt(“Enter your age:”) to get the age as input.

```
Enter your age: 30
You are 5 years older than me.
```

3. If a is greater than b return 'a is greater than b' else 'a is less than b'. Try to implement it in two ways
   - using if else
   - ternary operator.

```
let a = 4
let b = 3


  4 is greater than 3
```

4. Even numbers are divisible by 2 and the remainder is zero. How do you check, if a number is even or not using JavaScript?

```
Enter a number: 2
2 is an even number

Enter a number: 9
9 is is an odd number.
```

**Exercises: Level 2**

1. Write a code which can give grades to students according to theirs scores:
   - 80-100, A
   - 70-89, B
   - 60-69, C
   - 50-59, D
   - 0-49, F
2. Check if the season is Autumn, Winter, Spring or Summer.
   If the user input is :
   - September, October or November, the season is Autumn.
   - December, January or February, the season is Winter.
   - March, April or May, the season is Spring
   - June, July or August, the season is Summer
3. Check if a day is weekend day or a working day. Your script will take day as an input.

```
What is the day  today? Saturday
Saturday is a weekend.

What is the day today? saturDaY
Saturday is a weekend.

What is the day today? Friday
Friday is a working day.

What is the day today? FrIDAy
Friday is a working day.
```

**Exercises: Level 3**

1. Write a program which tells the number of days in a month.

```
Enter a month: January
January has 31 days.

Enter a month: JANUARY
January has 31 day

Enter a month: February
February has 28 days.

Enter a month: FEbruary
February has 28 days.
```

1. Write a program which tells the number of days in a month, now consider leap year.

# 5. Loops

In programming we use different loops to carry out repetitive tasks. Therefore, loop can help us to automate tedious and repetitive task. JavaScript has also different types of loops which we can use to work on repetitive task.

Imagine if your are asked to print Hello world one thousand times without a loop, it may take an hour or two to do this tedious task. However, using loop we can print it in less than a second.

Loops:

- for
- while
- do while
- for of

- forEach
- for in

A loop usually goes until the condition gets false. But sometimes we like to interrupt the loop or skip an item during iteration. We use *break* to interrupt the loop and *continue* to skip an item during iteration.

## Types of Loops

### 1. for

We use for loop when we know how many iteration we go. Let's see the following example

```
// for loop syntax

for (initialization, condition, increment/decrement) {
    code goes here
}
```

This code prints from 0 to 5.

```
for (let i = 0; i < 6; i++) {
  console.log(i)
}
```

For example if we want to sum all the numbers from 0 to 100.

```
let sum = 0
for (let i = 0; i < 101; i++) {
  sum += i
}

console.log(sum)
```

If we want to sum only even numbers:

```
let sum = 0
for (let i = 0; i < 101; i += 2) {
  sum += i
}

console.log(sum)

// or another way

let total = 0
for (let i = 0; i < 101; i++) {
  if (i % 2 == 0) {
    total += i
  }
}
console.log(total)
```

This code iterates through the array

```
const nums = [1, 2, 3, 4, 5]
for (let i = 0; i < 6; i++) {
  console.log(nums[i])
}
```

This code prints 5 to 0. Looping in reverse order

```
for (let i = 5; i >= 0; i--) {
  console.log(i)
}
```

The Code below can reverse an array.

```
const nums = [1, 2, 3, 4, 5]
const lastIndex = nums.length - 1
const newArray = []
for (let i = lastIndex; i >= 0; i--) {
  newArray.push(nums[i])
}

console.log(newArray)
```

**2. while**

We use the while loop when we do not know how man iteration we go in advance.

```
let count = prompt('Enter a positive number: ')
while (count > 0) {
  console.log(count)
  count--
}
```

### 3. do while

Do while run at least once if the condition is true or false

```
let count = 0
do {
  console.log(count)
  count++
} while (count < 11)
```

The code below runs ones though the condition is false

```
let count = 11
do {
  console.log(count)
  count++
} while (count < 11)
```

While loop is the least important loop in many programming languages.

### 4. for of

The for of loop is very handy to use it with array. If we are not interested in the index of the array a for of loop is preferable to regular for loop or forEach loop.

```
const numbers = [1, 2, 3, 4, 5]
for (const number of numbers) {
  console.log(number)
}

const countries = ['Finland', 'Sweden', 'Norway', 'Denmark', 'Iceland']
for (const country of countries) {
  console.log(country.toUpperCase())
}
```

### 5. forEach

If we are interested in the index of the array forEach is preferable to for of loop. The forEach array method takes a callback function, the callback function takes three arguments: the item, the index and

the array itself.

```
const numbers = [1, 2, 3, 4, 5]
numbers.forEach((number, i) => {
  console.log(number, i)
})

const countries = ['Finland', 'Sweden', 'Norway', 'Denmark', 'Iceland']
countries.forEach((country, i, arr) => {
  console.log(i, country.toUpperCase())
})
```

### 6. for in

The for in loop can be used with object literals to get the keys of the object.

```
const user = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  skills: ['HTML', 'CSS', 'JS', 'React', 'Node', 'Python', 'D3.js'],
}

for (const key in user) {
  console.log(key, user[key])
}
```

## Interrupting a loop and skipping an item

### break

Break is used to interrupt a loop.

```
for (let i = 0; i <= 5; i++) {
  if (i == 3) {
    break
  }
  console.log(i)
}

// 0 1 2
```

The above code stops if 3 found in the iteration process.

**continue**

We use the keyword continue to skip a certain iterations.

```
for (let i = 0; i <= 5; i++) {
  if (i == 3) {
    continue
  }
  console.log(i)
}
// 0 1 2 4 5
```

### Conclusions

- Regular for loop can be used anywhere when the number of iteration is known.
- While loop when the number of iteration is not know
- Do while loop and while loop are almost the same but do while loop run at least once even when the condition is false
- for of is used only for array
- forEach is used for array
- for in is used for object

## 6. Scope

Variable is the fundamental part in programming. We declare variable to store different data types. To declare a variable we use the key word *var*, *let* and *const*. A variable can declared at different scope. In this section we will see the scope, scope of variables when we use var or let.
Variables scopes can be:

- Window
- Global
- Local

Variable can be declared globally or locally or window scope. We will see both global and local scope. Anything declared without let, var or const is scoped at window level.

Let us image we have a scope.js file.

### Window Scope

Without using console.log() open your browser and check, you will see the value of a and b if you write a or b on the browser. That means a and b are already available in the window.

```
//scope.js
a = 'JavaScript' // is a window scope this found anywhere
b = 10 // this is a window scope variable
function letsLearnScope() {
  console.log(a, b)
  if (true) {
    console.log(a, b)
  }
}
console.log(a, b) // accessible
```

## Global scope

A globally declared variable can be accessed every where in the same file. But the term global is relative. It can be global to the file or it can be global relative to some block of codes.

```
//scope.js
let a = 'JavaScript' // is a global scope it will be found anywhere in this file
let b = 10 // is a global scope it will be found anywhere in this file
function letsLearnScope() {
  console.log(a, b) // JavaScript 10, accessible
  if (true) {
    let a = 'Python'
    let b = 100
    console.log(a, b) // Python 100
  }
  console.log(a, b)
}
letsLearnScope()
console.log(a, b) // JavaScript 10, accessible
```

## Local scope

A variable declared as local can be accessed only in certain block code.

```
//scope.js
let a = 'JavaScript' // is a global scope it will be found anywhere in this file
let b = 10 // is a global scope it will be found anywhere in this file
function letsLearnScope() {
  console.log(a, b) // JavaScript 10, accessible
  let c = 30
  if (true) {
    // we can access from the function and outside the function but
    // variables declared inside the if will not be accessed outside the if block
    let a = 'Python'
    let b = 20
    let d = 40
    console.log(a, b, c) // Python 20 30
  }
  // we can not access c because c's scope is only the if block
  console.log(a, b) // JavaScript 10
}
letsLearnScope()
console.log(a, b) // JavaScript 10, accessible
```

Now, you have an understanding of scope. A variable declared with *var* only scoped to function but variable declared with *let* or *const* is block scope(function block, if block, loop etc). Block in JavaScript is a code in between two curly brackets ({}).

```
//scope.js
function letsLearnScope() {
  var gravity = 9.81
  console.log(gravity)
}
// console.log(gravity), Uncaught ReferenceError: gravity is not defined

if (true) {
  var gravity = 9.81
  console.log(gravity) // 9.81
}
console.log(gravity) // 9.81

for (var i = 0; i < 3; i++) {
  console.log(i) // 1, 2, 3
}
console.log(i)
```

In ES6 and above there is *let* and *const*, so you will not suffer from the sneakiness of *var*. When we use *let* our variable is block scoped and it will not infect other parts of our code.

```
//scope.js
function letsLearnScope() {
  // you can use let or const, but gravity is constant I prefer to use const
  const gravity = 9.81
  console.log(gravity)
}
// console.log(gravity), Uncaught ReferenceError: gravity is not defined

if (true) {
  const gravity = 9.81
  console.log(gravity) // 9.81
}
// console.log(gravity), Uncaught ReferenceError: gravity is not defined

for (let i = 0; i < 3; i++) {
  console.log(i) // 1, 2, 3
}
// console.log(i), Uncaught ReferenceError: gravity is not defined
```

The scope *let* and *const* is the same. The difference is only reassigning. We can not change or reassign the value of const variable. I would strongly suggest you to use *let* and *const*, by using *let* and *const* you will writ clean code and avoid hard to debug mistakes. As a rule of thumb, you can use *let* for any value which change, *const* for any constant value, and for array, object, arrow function and function expression.

# 7. Object

Everything can be an object and objects do have properties and properties have values, so an object is a key value pair. The order of the key is not reserved, or there is no order.
To create an object literal, we use two curly brackets.

## Creating an empty object

An empty object

```
const person = {}
```

## Creating an objecting with values

Now, the person object has firstName, lastName, age, location, skills and isMarried properties. The value of properties or keys could be a string, number, boolean, an object, null, undefined or a function.

Let us see some examples of object. Each key has a value in the object.

```
const rectangle = {
  length: 20,
  width: 20,
}
console.log(rectangle) // {length: 20, width: 20}

const person = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  city: 'Helsinki',
  skills: [
    'HTML',
    'CSS',
    'JavaScript',
    'React',
    'Node',
    'MongoDB',
    'Python',
    'D3.js',
  ],
  isMarried: true,
}
console.log(person)
```

## Getting values from an object

We can access values of object using two methods:

- using . followed by key name if the key-name is a one word
- using square bracket and a quote

```
const person = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  city: 'Helsinki',
  skills: [
    'HTML',
    'CSS',
    'JavaScript',
    'React',
    'Node',
    'MongoDB',
    'Python',
    'D3.js',
  ],
  getFullName: function () {
    return `${this.firstName}${this.lastName}`
  },
  'phone number': '+3584545454545',
}

// accessing values using .
console.log(person.firstName)
console.log(person.lastName)
console.log(person.age)
console.log(person.location)

// value can be accessed using square bracket and key name
console.log(person['firstName'])
console.log(person['lastName'])
console.log(person['age'])
console.log(person['age'])
console.log(person['location'])

// for instance to access the phone number we only use the square bracket method
console.log(person['phone number'])
```

## Creating object methods

Now, the person object has getFullName properties. The getFullName is function inside the person object and we call it an object method. The *this* key word refers to the object itself. We can use the word *this* to access the values of different properties of the object. We can not use an arrow function as object method because the word this refers to the window inside an arrow function instead of the object itself. Example of object:

```
const person = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  city: 'Helsinki',
  skills: [
    'HTML',
    'CSS',
    'JavaScript',
    'React',
    'Node',
    'MongoDB',
    'Python',
    'D3.js',
  ],
  getFullName: function () {
    return `${this.firstName} ${this.lastName}`
  },
}

console.log(person.getFullName())
// Asabeneh Yetayeh
```

## Setting new key for an object

An object is a mutable data structure and we can modify the content of an object after it gets created.

Setting a new keys in an object

```
const person = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  city: 'Helsinki',
  skills: [
    'HTML',
    'CSS',
    'JavaScript',
    'React',
    'Node',
    'MongoDB',
    'Python',
    'D3.js',
  ],
  getFullName: function () {
    return `${this.firstName} ${this.lastName}`
  },
}
person.nationality = 'Ethiopian'
person.country = 'Finland'
person.title = 'teacher'
person.skills.push('Meteor')
person.skills.push('SasS')
person.isMarried = true

person.getPersonInfo = function () {
  let skillsWithoutLastSkill = this.skills
    .slice(0, this.skills.length - 1)
    .join(', ')
  let lastSkill = this.skills.slice(this.skills.length - 1)[0]

  let skills = `${skillsWithoutLastSkill}, and ${lastSkill}`
  let fullName = this.getFullName()
  let statement = `${fullName} is a ${this.title}.\nHe lives in ${this.country}.\nHe teaches ${s
  return statement
}
console.log(person)
console.log(person.getPersonInfo())


Asabeneh Yetayeh is a teacher.
He lives in Finland.
He teaches HTML, CSS, JavaScript, React, Node, MongoDB, Python, D3.js, Meteor, and SasS.
```

## Object Methods

There are different methods to manipulate an object. Let us see some of the available methods.

*Object.assign*: To copy an object without modifying the original object

```
const person = {
  firstName: 'Asabeneh',
  age: 250,
  country: 'Finland',
  city: 'Helsinki',
  skills: ['HTML', 'CSS', 'JS'],
  title: 'teacher',
  address: {
    street: 'Heitamienkatu 16',
    pobox: 2002,
    city: 'Helsinki',
  },
  getPersonInfo: function () {
    return `I am ${this.firstName} and I live in ${this.city}, ${this.country}. I am ${this.age}
  },
}

//Object methods: Object.assign, Object.keys, Object.values, Object.entries
//hasOwnProperty

const copyPerson = Object.assign({}, person)
console.log(copyPerson)
```

### Getting object keys using Object.keys()

*Object.keys*: To get the keys or properties of an object as an array

```
const keys = Object.keys(copyPerson)
console.log(keys) //['name', 'age', 'country', 'skills', 'address', 'getPersonInfo']
const address = Object.keys(copyPerson.address)
console.log(address) //['street', 'pobox', 'city']
```

### Getting object values using Object.values()

*Object.values*:To get values of an object as an array

```
const values = Object.values(copyPerson)
console.log(values)
```

### Getting object keys and values using Object.entries()

*Object.entries*:To get the keys and values in an array

```
const entries = Object.entries(copyPerson)
console.log(entries)
```

### Checking properties using hasOwnProperty()

*hasOwnProperty*: To check if a specific key or property exist in an object

```
console.log(copyPerson.hasOwnProperty('name'))
console.log(copyPerson.hasOwnProperty('score'))
```

🌕 You are astonishing. Now, you are super charged with the power of objects. You have just completed day 8 challenges and you are 8 steps a head into your way to greatness. Now do some exercises for your brain and for your muscle.

# 💻 Exercises

### Exercises: Level 1

1. Create an empty object called dog
2. Print the the dog object on the console
3. Add name, legs, color, age and bark properties for the dog object. The bark property is a method which return *woof woof*
4. Get name, legs, color, age and bark value from the dog object
5. Set new properties the dog object: breed, getDogInfo

### Exercises: Level 2

1. Find the person who has many skills in the users object.
2. Count logged in users,count users having greater than equal to 50 points from the following object.

```
const users = {
  Alex: {
    email: 'alex@alex.com',
    skills: ['HTML', 'CSS', 'JavaScript'],
    age: 20,
    isLoggedIn: false,
    points: 30
  },
  Asab: {
    email: 'asab@asab.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'Redux', 'MongoDB', 'Express', 'React', 'Node'],
    age: 25,
    isLoggedIn: false,
    points: 50
  },
  Brook: {
    email: 'daniel@daniel.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'React', 'Redux'],
    age: 30,
    isLoggedIn: true,
    points: 50
  },
  Daniel: {
    email: 'daniel@alex.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'Python'],
    age: 20,
    isLoggedIn: false,
    points: 40
  },
  John: {
    email: 'john@john.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'React', 'Redux', 'Node.js'],
    age: 20,
    isLoggedIn: true,
    points: 50
  },
  Thomas: {
    email: 'thomas@thomas.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'React'],
    age: 20,
    isLoggedIn: false,
    points: 40
  },
  Paul: {
    email: 'paul@paul.com',
    skills: ['HTML', 'CSS', 'JavaScript', 'MongoDB', 'Express', 'React', 'Node'],
    age: 20,
    isLoggedIn: false,
    points: 40
  }
```

```
}```

3. Find people who are MERN stack developer from the users object
4. Set your name in the users object without modifying the original users object
5. Get all keys or properties of users object
6. Get all the values of users object
7. Use the countries object to print a country name, capital, populations and languages.

**Exercises: Level 3**

1. Create an object literal called *personAccount*. It has *firstName, lastName, incomes, expenses* properties and it has *totalIncome, totalExpense, accountInfo,addIncome, addExpense* and *accountBalance* methods. Incomes is a set of incomes and its description and expenses is a set of incomes and its description.
2. **** Questions:2, 3 and 4 are based on the following two arrays:users and products ()
```

```
const users = [
  {
    _id: 'ab12ex',
    username: 'Alex',
    email: 'alex@alex.com',
    password: '123123',
    createdAt: '08/01/2020 9:00 AM',
    isLoggedIn: false,
  },
  {
    _id: 'fg12cy',
    username: 'Asab',
    email: 'asab@asab.com',
    password: '123456',
    createdAt: '08/01/2020 9:30 AM',
    isLoggedIn: true,
  },
  {
    _id: 'zwf8md',
    username: 'Brook',
    email: 'brook@brook.com',
    password: '123111',
    createdAt: '08/01/2020 9:45 AM',
    isLoggedIn: true,
  },
  {
    _id: 'eefamr',
    username: 'Martha',
    email: 'martha@martha.com',
    password: '123222',
    createdAt: '08/01/2020 9:50 AM',
    isLoggedIn: false,
  },
  {
    _id: 'ghderc',
    username: 'Thomas',
    email: 'thomas@thomas.com',
    password: '123333',
    createdAt: '08/01/2020 10:00 AM',
    isLoggedIn: false,
  },
]

const products = [
  {
    _id: 'eedfcf',
    name: 'mobile phone',
    description: 'Huawei Honor',
    price: 200,
    ratings: [
```

```
      { userId: 'fg12cy', rate: 5 },
      { userId: 'zwf8md', rate: 4.5 },
    ],
    likes: [],
  },
  {
    _id: 'aegfal',
    name: 'Laptop',
    description: 'MacPro: System Darwin',
    price: 2500,
    ratings: [],
    likes: ['fg12cy'],
  },
  {
    _id: 'hedfcg',
    name: 'TV',
    description: 'Smart TV:Procaster',
    price: 400,
    ratings: [{ userId: 'fg12cy', rate: 5 }],
    likes: ['fg12cy'],
  },
]
```

Imagine you are getting the above users collection from a MongoDB database.
a. Create a function called signUp which allows user to add to the collection. If user exists, inform the user that he has already an account.
b. Create a function called signIn which allows user to sign in to the application

3. The products array has three elements and each of them has six properties.
   a. Create a function called rateProduct which rates the product
   b. Create a function called averageRating which calculate the average rating of a product
4. Create a function called likeProduct. This function will helps to like to the product if it is not liked and remove like if it was liked.

# 8. Functions

So far we have seen many builtin JavaScript functions. In this section, we will focus on custom functions. What is a function? Before we start making functions, lets understand what function is and why we need function?

A function is a reusable block of code or programming statements designed to perform a certain task. A function is declared by a function key word followed by a name, followed by parentheses (). A parentheses can take a parameter. If a function take a parameter it will be called with argument. A function can also take a default parameter. To store a data to a function, a function has to return

certain data types. To get the value we call or invoke a function.
Function makes code:

- clean and easy to read
- reusable
- easy to test

A function can be declared or created in couple of ways:

- *Declaration function*
- *Expression function*
- *Anonymous function*
- *Arrow function*

## Function Declaration

Let us see how to declare a function and how to call a function.

```
//declaring a function without a parameter
function functionName() {
  // code goes here
}
functionName() // calling function by its name and with parentheses
```

## Function without a parameter and return

Function can be declared without a parameter.

**Example:**

```
// function without parameter,  a function which make a number square
function square() {
  let num = 2
  let sq = num * num
  console.log(sq)
}

square() // 4

// function without parameter
function addTwoNumbers() {
  let numOne = 10
  let numTwo = 20
  let sum = numOne + numTwo

  console.log(sum)
}

addTwoNumbers() // a function has to be called by its name to be executed


function printFullName() {
  let firstName = 'Asabeneh'
  let lastName = 'Yetayeh'
  let space = ' '
  let fullName = firstName + space + lastName
  console.log(fullName)
}

printFullName() // calling a function
```

## Function returning value

Function can also return values, if a function does not return values the value of the function is undefined. Let us write the above functions with return. From now on, we return value to a function instead of printing it.

```
function printFullName() {
  let firstName = 'Asabeneh'
  let lastName = 'Yetayeh'
  let space = ' '
  let fullName = firstName + space + lastName
  return fullName
}
console.log(printFullName())
```

```
function addTwoNumbers() {
  let numOne = 2
  let numTwo = 3
  let total = numOne + numTwo
  return total
}

console.log(addTwoNumbers())
```

## Function with a parameter

In a function we can pass different data types(number, string, boolean, object, function) as a parameter.

```
// function with one parameter
function functionName(parm1) {
  //code goes her
}
functionName(parm1) // during calling or invoking one argument needed

function areaOfCircle(r) {
  let area = Math.PI * r * r
  return area
}

console.log(areaOfCircle(10)) // should be called with one argument

function square(number) {
  return number * number
}

console.log(square(10))
```

## Function with two parameters

```
// function with two parameters
function functionName(parm1, parm2) {
  //code goes her
}
functionName(parm1, parm2) // during calling or invoking two arguments needed
// Function without parameter doesn't take input, so lets make a function with parameters
function sumTwoNumbers(numOne, numTwo) {
  let sum = numOne + numTwo
  console.log(sum)
}
sumTwoNumbers(10, 20) // calling functions
// If a function doesn't return it doesn't store data, so it should return

function sumTwoNumbers(numOne, numTwo) {
  let sum = numOne + numTwo
  return sum
}

console.log(sumTwoNumbers(10, 20))
function printFullName(firstName, lastName) {
  return `${firstName} ${lastName}`
}
console.log(printFullName('Asabeneh', 'Yetayeh'))
```

## Function with many parameters

```
// function with multiple parameters
function functionName(parm1, parm2, parm3,...){
  //code goes here
}
functionName(parm1,parm2,parm3,...) // during calling or invoking three arguments needed


// this function takes array as a parameter and sum up the numbers in the array
function sumArrayValues(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum = sum + arr[i];
  }
  return sum;
}
const numbers = [1, 2, 3, 4, 5];
    //calling a function
console.log(sumArrayValues(numbers));


    const areaOfCircle = (radius) => {
      let area = Math.PI * radius * radius;
      return area;
    }
console.log(areaOfCircle(10))
```

## Function with unlimited number of parameters

Sometimes we do not know how many arguments the user going to pass. Therefore, we should know how to write a function which can take unlimited number of arguments. The way we do it has a significant difference between a function declaration(regular function) and arrow function. Let us see examples both in function declaration and arrow function.

### Unlimited number of parameters in regular function

A function declaration provides a function scoped arguments array like object. Any thing we passed as argument in the function can be accessed from arguments object inside the functions. Let us see an example

```
// Let us access the arguments object

function sumAllNums() {
 console.log(arguments)
}

sumAllNums(1, 2, 3, 4))
// Arguments(4) [1, 2, 3, 4, callee: ƒ, Symbol(Symbol.iterator): ƒ]



// function declaration

function sumAllNums() {
  let sum = 0
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i]
  }
  return sum
}

console.log(sumAllNums(1, 2, 3, 4)) // 10
console.log(sumAllNums(10, 20, 13, 40, 10))  // 93
console.log(sumAllNums(15, 20, 30, 25, 10, 33, 40))  // 173
```

### Unlimited number of parameters in arrow function

Arrow function does not have the function scoped arguments object. To implement a function which takes unlimited number of arguments in an arrow function we use spread operator followed by any parameter name. Any thing we passed as argument in the function can be accessed as array in the arrow function. Let us see an example

```
// Let us access the arguments object

const sumAllNums = (...args) => {
 // console.log(arguments), arguments object not found in arrow function
 // instead we use an a parameter followed by spread operator
 console.log(args)
}

sumAllNums(1, 2, 3, 4))
// [1, 2, 3, 4]
```

```
// function declaration

const sumAllNums = (...args) => {
  let sum = 0
  for (const element of args) {
    sum += element
  }
  return sum
}

console.log(sumAllNums(1, 2, 3, 4)) // 10
console.log(sumAllNums(10, 20, 13, 40, 10))  // 93
console.log(sumAllNums(15, 20, 30, 25, 10, 33, 40))  // 173
```

## Anonymous Function

Anonymous function or without name

```
const anonymousFun = function () {
  console.log(
    'I am an anonymous function and my value is stored in anonymousFun'
  )
}
```

## Expression Function

Expression functions are anonymous functions. After we create a function without a name and we assign it to a variable. To return a value from the function we should call the variable. Look at the example below.

```
// Function expression
const square = function (n) {
  return n * n
}

console.log(square(2)) // -> 4
```

## Self Invoking Functions

Self invoking functions are anonymous functions which do not need to be called to return a value.

```
;(function (n) {
  console.log(n * n)
})(2) // 4, but instead of just printing if we want to return and store the data, we do as showr

let squaredNum = (function (n) {
  return n * n
})(10)

console.log(squaredNum)
```

## Arrow Function

Arrow function is an alternative to write a function, however function declaration and arrow function have some minor differences.

Arrow function uses arrow instead of the keyword *function* to declare a function. Let us see both function declaration and arrow function.

```
// This is how we write normal or declaration function
// Let us change this declaration function to an arrow function
function square(n) {
  return n * n
}

console.log(square(2)) // 4

const square = (n) => {
  return n * n
}

console.log(square(2)) // -> 4

// if we have only one line in the code block, it can be written as follows, explicit return
const square = (n) => n * n // -> 4


const changeToUpperCase = (arr) => {
  const newArr = []
  for (const element of arr) {
    newArr.push(element.toUpperCase())
  }
  return newArr
}

const countries = ['Finland', 'Sweden', 'Norway', 'Denmark', 'Iceland']
console.log(changeToUpperCase(countries))

// ["FINLAND", "SWEDEN", "NORWAY", "DENMARK", "ICELAND"]
```

```
const printFullName = (firstName, lastName) => {
  return `${firstName} ${lastName}`
}

console.log(printFullName('Asabeneh', 'Yetayeh'))
```

The above function has only the return statement, therefore, we can explicitly return it as follows.

```
const printFullName = (firstName, lastName) => `${firstName} ${lastName}`

console.log(printFullName('Asabeneh', 'Yetayeh'))
```

## Function with default parameters

Sometimes we pass default values to parameters, when we invoke the function if we do not pass an argument the default value will be used. Both function declaration and arrow function can have a default value or values.

```
// syntax
// Declaring a function
function functionName(param = value) {
  //codes
}

// Calling function
functionName()
functionName(arg)
```

**Example:**

```
function greetings(name = 'Peter') {
  let message = `${name}, welcome to 30 Days Of JavaScript!`
  return message
}

console.log(greetings())
console.log(greetings('Asabeneh'))
```

```
function generateFullName(firstName = 'Asabeneh', lastName = 'Yetayeh') {
  let space = ' '
  let fullName = firstName + space + lastName
  return fullName
}

console.log(generateFullName())
console.log(generateFullName('David', 'Smith'))


function calculateAge(birthYear, currentYear = 2019) {
  let age = currentYear - birthYear
  return age
}

console.log('Age: ', calculateAge(1819))


function weightOfObject(mass, gravity = 9.81) {
  let weight = mass * gravity + ' N' // the value has to be changed to string first
  return weight
}

console.log('Weight of an object in Newton: ', weightOfObject(100)) // 9.81 gravity at the surfa
console.log('Weight of an object in Newton: ', weightOfObject(100, 1.62)) // gravity at surface
```

Let us see how we write the above functions with arrow functions

```
// syntax
// Declaring a function
const functionName = (param = value) => {
  //codes
}

// Calling function
functionName()
functionName(arg)
```

**Example:**

```
const greetings = (name = 'Peter') => {
  let message = name + ', welcome to 30 Days Of JavaScript!'
  return message
}

console.log(greetings())
console.log(greetings('Asabeneh'))
```

```
const generateFullName = (firstName = 'Asabeneh', lastName = 'Yetayeh') => {
  let space = ' '
  let fullName = firstName + space + lastName
  return fullName
}

console.log(generateFullName())
console.log(generateFullName('David', 'Smith'))


const calculateAge = (birthYear, currentYear = 2019) => currentYear - birthYear
console.log('Age: ', calculateAge(1819))


const weightOfObject = (mass, gravity = 9.81) => mass * gravity + ' N'

console.log('Weight of an object in Newton: ', weightOfObject(100)) // 9.81 gravity at the surfa
console.log('Weight of an object in Newton: ', weightOfObject(100, 1.62)) // gravity at surface
```

### Function declaration versus Arrow function

It ill be covered in other time

## 💻 Exercises

#### Exercises: Level 1

1. Declare a function *fullName* and it takes firstName, lastName as a parameter and it returns your full - name.
2. Declare a function *addNumbers* and it takes two two parameters and it returns sum.
3. Area of a circle is calculated as follows: *area = π x r x r*. Write a function which calculates _areaOfCircle
4. Temperature in oC can be converted to oF using this formula: *oF = (oC x 9/5) + 32*. Write a function which convert oC to oF *convertCelciusToFahrenheit*.
5. Body mass index(BMI) is calculated as follows: *bmi = weight in Kg / (height x height) in m2*. Write a function which calculates *bmi*. BMI is used to broadly define different weight groups in adults 20 years old or older.Check if a person is *underweight, normal, overweight* or *obese* based the information given below.
    - The same groups apply to both men and women.
    - *Underweight*: BMI is less than 18.5
    - *Normal weight*: BMI is 18.5 to 24.9
    - *Overweight*: BMI is 25 to 29.9
    - *Obese*: BMI is 30 or more

6. Write a function called *checkSeason*, it takes a month parameter and returns the season:Autumn, Winter, Spring or Summer.

**Exercises: Level 2**

1. Quadratic equation is calculated as follows: *ax2 + bx + c = 0*. Write a function which calculates value or values of a quadratic equation, *solveQuadEquation*.

```
console.log(solveQuadratic()) // {0}
console.log(solveQuadratic(1, 4, 4)) // {-2}
console.log(solveQuadratic(1, -1, -2)) // {2, -1}
console.log(solveQuadratic(1, 7, 12)) // {-3, -4}
console.log(solveQuadratic(1, 0, -4)) //{2, -2}
console.log(solveQuadratic(1, -1, 0)) //{1, 0}
```

2. Declare a function name *printArray*. It takes array as a parameter and it prints out each value of the array.
3. Write a function name *showDateTime* which shows time in this format: 08/01/2020 04:08 using the Date object.

```
showDateTime()
08/01/2020 04:08
```

4. Declare a function name *swapValues*. This function swaps value of x to y.

```
swapValues(3, 4) // x => 4, y=>3
swapValues(4, 5) // x = 5, y = 4
```

5. Declare a function name *reverseArray*. It takes array as a parameter and it returns the reverse of the array (don't use method).

```
console.log(reverseArray([1, 2, 3, 4, 5]))
//[5, 4, 3, 2, 1]
console.log(reverseArray(['A', 'B', 'C']))
//['C', 'B', 'A']
```

6. Declare a function name *capitalizeArray*. It takes array as a parameter and it returns the -capitalizedarray.
7. Declare a function name *addItem*. It takes an item parameter and it returns an array after adding the item
8. Declare a function name *removeItem*. It takes an index parameter and it returns an array after removing an ite
9. Declare a function name evensAndOdds . It takes a positive integer as parameter and it counts number of evens and odds in the number.

```
evensAndOdds(100);
The number of odds are 50.
The number of evens are 51.
```

13. Write a function which takes any number of arguments and return the sum of the arguments

```
sum(1, 2, 3) // -> 6
sum(1, 2, 3, 4) // -> 10
```

1. Declare a function name *userIdGenerator*. When this function is called it generates seven character id. The function return the id.

```
console.log(userIdGenerator());
41XTDbE
```

**Exercises: Level 3**

1. Declare a function name *userIdGeneratedByUser*. It doesn't take any parameter but it takes two inputs using prompt(). One of the input is the number of characters and the second input is the number of ids which are supposed to be generated.

```
userIdGeneratedByUser()
'kcsy2
SMFYb
bWmeq
ZXOYh
2Rgxf
'
userIdGeneratedByUser()
'1GCSgPLMaBAVQZ26
YD7eFwNQKNs7qXaT
ycArC5yrRupyG00S
UbGxOFI7UXSWAyKN
dIV0SSUTgAdKwStr
'
```

2. Write a function ***generateColors*** which can generate any number of hexa or rgb colors.

```
console.log(generateColors('hexa', 3)) // ['#a3e12f', '#03ed55', '#eb3d2b']
console.log(generateColors('hexa', 1)) // '#b334ef'
console.log(generateColors('rgb', 3)) // ['rgb(5, 55, 175)', 'rgb(50, 105, 100)', 'rgb(15, 
console.log(generateColors('rgb', 1)) // 'rgb(33,79, 176)'
```

3. Call your function *shuffleArray*, it takes an array as a parameter and it returns a shuffled array

4. Call your function *factorial*, it takes a whole number as a parameter and it return a factorial of the number
5. Call your function *isEmpty*, it takes a parameter and it checks if it is empty or not
6. Write a function called *average*, it takes an array parameter and returns the average of the items. Check if all the array items are number types. If not give return reasonable feedback.

# 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 function could be any name
const callback = (n) => {
  return n ** 2
}

// function take 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 were we use call back functions.For instance the *forEach* method uses call back.

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

}
console.log(sumArray(numbers))


10
```

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))


10
```

## setting time

In JavaScript we can execute some activity on certain interval of time or we can schedule(wait) for sometime to execute some activities.

- setInterval
- setTimeout

### setInterval

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, 2000) // it prints hello in every 2 seconds
```

**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.
```

# 10. Destructuring and Spreading

## What is Destructuring?

Destructuring is a way to unpack arrays, and objects and assigning to a distinct variable. Destructuring allows us to write clean and readable code.

## What can we destructure?

1. Arrays
2. Objects

### 1. Destructuring arrays

Arrays are a list of different data types ordered by their index. Let's see an example of arrays:

```
const numbers = [1, 2, 3]
const countries = ['Finland', 'Sweden', 'Norway']
```

We can access an item from an array using a certain index by iterating through the loop or manually as shown in the example below.

Accessing array items using a loop

```
for (const number of numbers) {
  console.log(number)
}

for (const country of countries) {
  console.log(country)
}
```

Accessing array items manually

```
const numbers = [1, 2, 3]
let num1 = numbers[0]
let num2 = numbers[1]
let num3 = numbers[2]
console.log(num1, num2, num3) // 1, 2, 3

const countries = ['Finland', 'Sweden', 'Norway']
let fin = countries[0]
let swe = countries[1]
let nor = countries[2]
console.log(fin, swe, nor) // Finland, Sweden, Norway
```

Most of the time the size of an array is big and we use a loop to iterate through each item of the arrays. Sometimes, we may have short arrays. If the array size is very short it is ok to access the items manually as shown above but today we will see a better way to access the array item which is destructuring.

Accessing array items using destructuring

```
const numbers = [1, 2, 3]
const [num1, num2, num3] = numbers
console.log(num1, num2, num3) // 1, 2, 3,

const constants = [2.72, 3.14, 9.81,37, 100]
const [e, pi, gravity, bodyTemp, boilingTemp] = constants
console.log(e, pi, gravity, bodyTemp, boilingTemp]
// 2.72, 3.14, 9.81, 37,100
const countries = ['Finland', 'Sweden', 'Norway']
const [fin, swe, nor] = countries
console.log(fin, swe, nor) // Finland, Sweden, Norway
```

During destructuring each variable should match with the index of the desired item in the array. For instance, the variable fin matches to index 0 and the variable nor matches to index 2. What would be the value of den if you have a variable den next nor?

```
const [fin, swe, nor, den] = countries
console.log(den) // undefined
```

If you tried the above task you confirmed that the value is undefined. Actually, we can pass a default value to the variable, and if the value of that specific index is undefined the default value will be used.

```
const countries = ['Finland', 'Sweden', undefined, 'Norway']
const [fin, swe, ice = 'Iceland', nor, den = 'Denmark'] = countries
console.log(fin, swe, ice, nor, den) // Finland, Sweden, Iceland, Norway, Denmark
```

Destructuring Nested arrays

```
const fullStack = [
  ['HTML', 'CSS', 'JS', 'React'],
  ['Node', 'Express', 'MongoDB']
]

const [frontEnd, backEnd] = fullstack
console.log(frontEnd, backEnd)

//["HTML", "CSS", "JS", "React"] , ["Node", "Express", "MongoDB"]

const fruitsAndVegetables = [['banana', 'orange', 'mango', 'lemon'],  ['Tomato', 'Potato', 'Cabb

const [fruits, vegetables] = fruitsAndVegetables
console.log(fruits, vegetables]

//['banana', 'orange', 'mango', 'lemon']

//['Tomato', 'Potato', 'Cabbage', 'Onion', 'Carrot']
```

Skipping an Item during destructuring

During destructuring if we are not interested in every item, we can omit a certain item by putting a comma at that index. Let's get only Finland, Iceland, and Denmark from the array. See the example below for more clarity:

```
const countries = ['Finland', 'Sweden', 'Iceland', 'Norway', 'Denmark']
const [fin, , ice, , den] = countries
console.log(fin, ice, den) // Finland, Iceland, Denmark
```

Getting the rest of the array using the spread operator
We use three dots(...) to spread or get the rest of an array during destructuring

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const [num1, num2, num3, ...rest] = nums
console.log(num1, num2, num3, rest) //1, 2, 3, [4, 5, 6, 7, 8, 9, 10]

const countries = [
  'Germany',
  'France',
  'Belgium',
  'Finland',
  'Sweden',
  'Norway',
  'Denmark',
  'Iceland',
]

let [gem, fra, , ...nordicCountries] = countries
console.log(gem, fra, nordicCountries)
// Germany, France,  ["Finland", "Sweden", "Norway", "Denmark", "Iceland"]
```

There many cases in which we use array destructuring, let's see the following example:

Destructuring when we loop through arrays

```
const countries = [
  ['Finland', 'Helsinki'],
  ['Sweden', 'Stockholm'],
  ['Norway', 'Oslo'],
]

for (const [country, city] of countries) {
  console.log(country, city)
}

const fullStack = [
  ['HTML', 'CSS', 'JS', 'React'],
  ['Node', 'Express', 'MongoDB'],
]

for (const [first, second, third, fourth] of fullStack) {
  console.log(first, second, third, fourt)
}
```

What do you think about the code snippet below? If you have started React Hooks already it may remind you of the useState hook.

```
const [x, y] = [2, (value) => value ** 2]
```

What is the value of x? And what is the value of y(x)? I leave this for you to figure out.

If you have used react hooks you are very familiar with this and as you may imagine it is destructuring. The initial value of count is 0 and the setCount is a method that changes the value of count.

```
const [count, setCount] = useState(0)
```

Now, you know how to destructure arrays. Let's move on to destructuring objects.

**2. Destructuring objects**

An object literal is made of key and value pairs. A very simple example of an object:

```
const rectangle = {
  width: 20,
  height: 10,
}
```

We access the value of an object using the following methods:

```
const rectangle = {
  width: 20,
  height: 10,
}

let width = rectangle.width
let height = recangle.height

// or

let width = rectangle[width]
let height = recangle[height]
```

But today, we will see how to access the value of an object using destructuring.

When we destructure an object the name of the variable should be exactly the same as the key or property of the object. See the example below.

```
const rectangle = {
  width: 20,
  height: 10,
}

let { width, height } = rectangle
console.log(width, height, perimeter) // 20, 10
```

What will be the value of we try to access a key which not in the object.

```
const rectangle = {
  width: 20,
  height: 10,
}

let { width, height, perimeter } = rectangleconsole.log(
  width,
  height,
  perimeter
) // 20, 10, undefined
```

The value of the perimeter in the above example is undefined.

Default value during object destructuring

Similar to the array, we can also use a default value in object destructuring.

```
const rectangle = {
  width: 20,
  height: 10,
}

let { width, height, perimeter = 200 } = rectangle
console.log(width, height, perimeter) // 20, 10, undefined
```

Renaming variable names

```
const rectangle = {
  width: 20,
  height: 10,
}

let { width: w, height: h } = rectangle
```

Let's also destructure, nested objects. In the example below, we have nested objects and we can destructure it in two ways.

We can just destructure step by step

```
const props = {
  user:{
    firstName:'Asabeneh',
    lastName:'Yetayeh',
    age:250
  },
  post:{
    title:'Destructuring and Spread',
    subtitle:'ES6',
    year:2020
},
skills:['JS', 'React', 'Redux', 'Node', 'Python']

}
}

const {user, post, skills} = props
const {firstName, lastName, age} = user
const {title, subtitle, year} = post
const [skillOne, skillTwo, skillThree, skillFour, skillFive] = skills
```

1. We can destructure it one step

```
const props = {
  user:{
    firstName:'Asabeneh',
    lastName:'Yetayeh',
    age:250
  },
  post:{
    title:'Destructuring and Spread',
    subtitle:'ES6',
    year:2020
},
skills:['JS', 'React', 'Redux', 'Node', 'Python']

}

}

const {user:{firstName, lastName, age}, post:{title, subtitle, year}, skills:[skillOne, skillTwo
```

Destructuring during loop through an array

```
const languages = [
  { lang: 'English', count: 91 },
  { lang: 'French', count: 45 },
  { lang: 'Arabic', count: 25 },
  { lang: 'Spanish', count: 24 },
  { lang: 'Russian', count: 9 },
  { lang: 'Portuguese', count: 9 },
  { lang: 'Dutch', count: 8 },
  { lang: 'German', count: 7 },
  { lang: 'Chinese', count: 5 },
  { lang: 'Swahili', count: 4 },
  { lang: 'Serbian', count: 4 },
]

for (const { lang, count } of languages) {
  console.log(`The ${lang} is spoken in ${count} countries.`)
}
```

Destructuring function parameter

```
const rectangle = { width: 20, height: 10 }
const calcualteArea = ({ width, height }) => width * height
const calculatePerimeter = ({ width, height } = 2 * (width + height))
```

## Exercises

Create a function called getPersonInfo. The getPersonInfo function takes an object parameter. The structure of the object and the output of the function is given below. Try to use both a regular way and destructuring and compare the cleanness of the code. If you want to compare your solution with my solution, check this link.

```
const person = {
  firstName: 'Asabeneh',
  lastName: 'Yetayeh',
  age: 250,
  country: 'Finland',
  job: 'Instructor and Developer',
  skills: [
    'HTML',
    'CSS',
    'JavaScript',
    'React',
    'Redux',
    'Node',
    'MongoDB',
    'Python',
    'D3.js',
  ],
  languages: ['Amharic', 'English', 'Suomi(Finnish)'],
}

/*
Asabeneh Yetayeh lives in Finland. He is  250 years old. He is an Instructor and Developer. He t
*/
```

## Spread or Rest Operator

When we destructure an array we use the spread operator(...) to get the rest elements as array. In addition to that we use spread operator to spread arr elements to another array.

### Spread operator to get the rest of array elements

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let [num1, num2, num3, ...rest] = nums

console.log(num1, num2, num3)
console.log(rest)


1 2 3
[4, 5, 6, 7, 8, 9, 10]
```

```
const countries = [
  'Germany',
  'France',
  'Belgium',
  'Finland',
  'Sweden',
  'Norway',
  'Denmark',
  'Iceland',
]

let [gem, fra, , ...nordicCountries] = countries

console.log(gem)
console.log(fra)
console.log(nordicCountries)
```

```
Germany
France
["Finland", "Sweden", "Norway", "Denmark", "Iceland"]
```

### Spread operator to copy array

```
const evens = [0, 2, 4, 6, 8, 10]
const evenNumbers = [...evens]

const odds = [1, 3, 5, 7, 9]
const oddNumbers = [...odds]

const wholeNumbers = [...evens, ...odds]

console.log(evenNumbers)
console.log(oddNumbers)
console.log(wholeNumbers)
```

```
[0, 2, 4, 6, 8, 10]
[1, 3, 5, 7, 9]
[0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

```
const frontEnd = ['HTML', 'CSS', 'JS', 'React']
const backEnd = ['Node', 'Express', 'MongoDB']
const fullStack = [...frontEnd, ...backEnd]

console.log(fullStack)
```

```
["HTML", "CSS", "JS", "React", "Node", "Express", "MongoDB"]
```

### Spread operator to copy object

We can copy an object using a spread operator

```
const user = {
  name: 'Asabeneh',
  title: 'Programmer',
  country: 'Finland',
  city: 'Helsinki',
}

const copiedUser = { ...user }
console.log(copiedUser)


{name: "Asabeneh", title: "Programmer", country: "Finland", city: "Helsinki"}
```

Modifying or changing the object while copying

```
const user = {
  name: 'Asabeneh',
  title: 'Programmer',
  country: 'Finland',
  city: 'Helsinki',
}

const copiedUser = { ...user, title: 'instructor' }
console.log(copiedUser)


{name: "Asabeneh", title: "instructor", country: "Finland", city: "Helsinki"}
```

### Spread operator with arrow function

Whenever we like to write an arrow function which takes unlimited number of arguments we use a spread operator. If we use a spread operator as a parameter, the argument passed when we invoke a function will change to an array.

```
const sumAllNums = (...args) => {
  console.log(args)
}

sumAllNums(1, 2, 3, 4, 5)
```

```
[1, 2, 3, 4, 5]
```

```
const sumAllNums = (...args) => {
  let sum = 0
  for (const num of args) {
    sum += num
  }
  return sum
}

console.log(sumAllNums(1, 2, 3, 4, 5))
```

```
15
```

# 11. Functional Programming

In this article, I will try to help you to have a very good understanding of the most common feature of JavaScript, *functional programming*.

*Functional programming* allows you to write shorter code, clean code, and also to solve complicated problems which might be difficult to solve in a traditional way.

In this article we will cover all JS functional programming methods:

- forEach
- map
- filter
- reduce
- find
- findIndex
- some
- every

## 1. forEach

We use forEach when we like to iterate through an array of items. The forEach is a higher-order function and it takes call-back as a parameter. The forEach method is used only with array and we use forEach if you are interested in each item or index or both.

```
// syntax in a normal or a function declaration

function callback(item, index, arr) {}
array.forEach(callback)

// or syntax in an arrow function
const callback = (item, i, arr) => {}
array.forEach(callback)
```

The call back function could be a function declaration or an arrow function.

Let see different examples

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']
countries.forEach(function (country, index, arr) {
  console.log(i, country.toUpperCase())
})
```

If there is no much code inside the code block we can use an arrow function and we can write it without a curly bracket. The index and the array parameters are optional, we can omit them.

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']
countries.forEach((country, i) => console.log(i, country.toUpperCase()))


0 "FINLAND"
1 "ESTONIA"
2 "SWEDEN"
3 "NORWAY"
```

For example if we like to change each country to uppercase and store it back to an array we write it as follows.

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']
const newCountries = []
countries.forEach((country) => newCountries.push(country))

console.log(newCountries) // ["Finland", "Estonia", "Sweden", "Norway"]
```

Let us see more examples. For instance if we want to sum an array of numbers we can use forEach or reduce. Let us see how we use forEach to sum all numbers in an array.

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

console.log(sum) // 15
```

Let us move to the next functional programming method which is going to be a map.

## 2. map

We use the map method whenever we like to modify an array. We use the map method only with arrays and it always returns an array.

```
// syntax in a normal or a function declaration

function callback(item, i) {
  return // code goes here
}

const modifiedArray = array.map(callback)

// or syntax in an arrow function

const callback = (item, i) => {
  return // code goes here
}
const modifiedArray = array.map(callback)
```

Now, let us modify the countries array using the map method. The index is an optional parameter

```
// Using function declaration

const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']

const newCountries = countries.map(function (country) {
  return country.toUpperCase()
})

console.log(newCountries)

// map using an arrow function call back

const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']
const newCountries = countries.map((country) => country.toUpperCase())

console.log(newCountries) // ["FINLAND", "ESTONIA", "SWEDEN", "NORWAY"]
```

As you can see that map is very handy to modify an array and to get an array back. Now, let us create an array of the length of the countries from the countries array.

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway']
const countriesLength = countries.map((country) => country.length)

console.log(countriesLength) // [7, 7, 6, 6]
```

Let us see another more example

```
const numbers = [1, 2, 3, 4, 5]
const squares = numbers.map((n) => n ** 2)

console.log(squares) // [1, 4, 9, 16, 25]
```

## 3. filter

As you may understand from the literal meaning of filter, it filters out items based on some criteria. The filter method like forEach and map is used with an array and it returns an array of the filtered items.

For instance if we want to filter out countries containing a substring land from an array of countries. See the example below:

```
// syntax in a normal or a function declaration
function callback(item) {
  return // boolean
}

const filteredArray = array.filter(callback)

// or syntax in an arrow function

const callback = (item) => {
  return // boolean
}
const filteredArray = array.filter(callback)


const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const countriesWithLand = countries.filter((country) =>
  country.includes('land')
)
console.log(countriesWithLand) // ["Finland", "Iceland"]
```

How about if we want to filter out countries not containing the substring land. We use negation to achieve that.

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const countriesWithLand = countries.filter(
  (country) => !country.includes('land')
)
console.log(countriesWithLand) // ["Estonia", "Sweden", "Norway"]
```

Let's see an additional example about the filter, let us filter even or odd numbers from an array of numbers

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const evens = numbers.filter((n) => n % 2 === 0)
const odds = numbers.filter((n) => n % 2 !== 0)
console.log(evens) // [0, 2, 4, 6, 8, 10]
console.log(odds) // [1, 3, 5, 7, 9]
```

Now, you know how to filter let us move on to the next functional programming, reduce.

## 4. reduce

Like forEach, map, and filter, reduce is also used with an array and it returns a single value. You can associate reduce with a blender. You put different fruits to a blend and you get a mix of fruit juice. The juice is the output from the reduction process.

We use the reduce method to sum all numbers in an array together, or to multiply items in an array or to concatenate items in an array. Let us see the following different example to make this explanation more clear.

```
// syntax in a normal or a function declaration

function callback(acc, cur) {
    return // code goes here
}

const reduced = array.reduce(callback, optionalInitialValue)

// or syntax in an arrow function

const reduced =  callback(acc, cur) => {
    return // code goes here
}
const reduced = array.reduce(callback)
```

The default initial value is 0. We can change the initial value if we want to change it.

For instance if we want to add all items in an array and if all the items are numbers we can use reduce.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const sum = numbers.reduce((acc, cur) => acc + cur)
console.log(sum) // 55
```

Reduce has a default initial value which is zero. Now, let us use a different initial value which is 5 in this case.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const sum = numbers.reduce((acc, cur) => acc + cur, 5)
console.log(sum) // 60
```

Let us concatenate strings using reduce

```
const strs = ['Hello', 'world', '!']
const helloWorld = strs.reduce((acc, cur) => acc + ' ' + cur)
console.log(helloWorld) // "Hello world !"
```

We can multiply items of an array using reduce and we will return the value.

```
const numbers = [1, 2, 3, 4, 5]
const value = numbers.reduce((acc, cur) => acc * cur)
console.log(value) // 120
```

Let us try it with an initial value

```
const numbers = [1, 2, 3, 4, 5]
const value = numbers.reduce((acc, cur) => acc * cur, 10)
console.log(value) // 1200
```

## 5. find

If we are interested in the first occurrence of a certain item or element in an array we can use find method. Unlike map and filter, find just return the first occurrence of an item instead of an array.

```
// syntax in a normal or a function declaration

function callback(item) {
return // code goes here
}

const item = array.find(callback)

// or syntax in an arrow function

const reduced =  callback(item) => {
return // code goes here
}
const item = array.find(callback)
```

Let find the first even number and the first odd number in the numbers array.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const firstEvenNum = numbers.find((n) => n % 2 === 0)
const firstOddNum = numbers.find((n) => n % 2 !== 0)
console.log(firstEvenNum) // 0
console.log(firstOddNum) // 1
```

Let us find the first country which contains a substring way

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const countryWithWay = countries.find((country) => country.includes('way'))
console.log(countriesWithWay) // Norway
```

Let us find the first country which has only six characters

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const sixCharsCountry = countries.find((country) => country.length === 6)
console.log(sixCharsCountry) // Sweden
```

Let us find the first country in the array which has the letter 'o'

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const index = countries.find((country) => country.includes('o'))
console.log(index // Estonia
```

## 6. findIndex

The findIndex method works like find but findIndex returns the index of the item. If we are interested in the index of a certain item or element in an array we can use findIndex. The findIndex return the index of the first occurrence of an item.

```
// syntax in a normal or a function declaration

function callback(item) {
return // code goes here
}

const index = array.findIndex(callback)

// or syntax in an arrow function

const reduced =  callback(item) => {
return // code goes here
}
const index = array.findIndex(callback)
```

Let us find the index of the first even number and the index of the first odd number in the numbers array.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const firstEvenIndex = numbers.findIndex((n) => n % 2 === 0)
const firstOddIndex = numbers.findIndex((n) => n % 2 !== 0)
console.log(firstEvenIndex) // 0
console.log(firstOddIndex) // 1
```

Let us find the index of the first country in the array which has exactly six characters

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const index = countries.findIndex((country) => country.length === 6)
console.log(index //2
```

Let us find the index of the first country in the array which has the letter 'o'.

```
const countries = ['Finland', 'Estonia', 'Sweden', 'Norway', 'Iceland']
const index = countries.findIndex((country) => country.includes('o'))
console.log(index // 1
```

Let us move on to the next functional programming, some.

## 7. some

The some method is used with array and return a boolean. If one or some of the items satisfy the criteria the result will be true else it will be false. Let us see it with an example.

In the following array some numbers are even and some are odd, so if I ask you a question, are there even numbers in the array then your answer will be yes. If I ask you also another question, are there odd numbers in the array then your answer will be yes. On the contrary, if I ask you, are all the numbers even or odd then your answer will be no because all the numbers are not even or odd.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const someAreEvens = numbers.some((n) => n % 2 === 0)
const someAreOdds = numbers.some((n) => n % 2 !== 0)
console.log(someAreEvens) // true
console.log(someAreOdds) // true
```

Let us another example

```
const evens = [0, 2, 4, 6, 8, 10]
const someAreEvens = evens.some((n) => n % 2 === 0)
const someAreOdds = evens.some((n) => n % 2 !== 0)
console.log(someAreEvens) // true
console.log(someAreOdds) // false
```

Now, let us see one more functional programming, every.

## 8. every

The method every is somehow similar to some but every item must satisfy the criteria. The method every like some returns a boolean.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const allAreEvens = numbers.every((n) => n % 2 === 0)
const allAreOdd s= numbers.every((n) => n % 2 !== 0)

console.log(allAreEven) // false
console.log(allAreOdd)  // false

const evens = [0, 2, 4, 6, 8, 10]
const someAreEvens = evens.some((n) => n % 2 === 0)
const someAreOdds = evens.some((n) => n % 2 !== 0)

console.log(someAreEvens) // true
console.log(someAreOdds)  // false
```

## Exercises

```
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. Print the price of each product using forEach
2. Print the product items as follows using forEach

```
The price of banana is 3 euros.
The price of mango is 6 euros.
The price of potato is unknown.
The price of avocado is 8 euros.
The price of coffee is 10 euros.
The price of tea is unknown.
```

3. Calculate the sum of all the prices using forEach
4. Create an array of prices using map and store it in a variable prices
5. Filter products with prices
6. Use method chaining to get the sum of the prices(map, filter, reduce)
7. Calculate the sum of all the prices using reduce only
8. Find the first product which doesn't have a price value
9. Find the index of the first product which does not have price value
10. Check if some products do not have a price value
11. Check if all the products have price value
12. Explain the difference between forEach, map, filter and reduce
13. Explain the difference between filter, find and findIndex
14. Explain the difference between some and every

# 12. Classes

JavaScript is an object oriented programming language. Everything in JavScript is an object, with its properties and methods. We create class to create an object. A Class is like an object constructor, or a "blueprint" for creating objects. We instantiate a class to create an object. The class defines attributes and the behavior of the object, while the object, on the other hand, represents the class.

Once we create a class we can create object from it whenever we want. Creating an object from a class is called class instantiation.

In the object section, we saw how to create an object literal. Object literal is a singleton. If we want to get a similar object , we have to write it. However, class allows to create many objects. This helps to reduce amount of code and repetition of code.

## Defining a classes

To define a class in JavaScript we need the keyword *class* , the name of a class in **CamelCase** and block code(two curly brackets). Let us create a class name Person.

```
// syntax
class ClassName {
    //  code goes here
}
```

**Example:**

```
class Person {
  // code goes here
}
```

We have created an Person class but it does not have any thing inside.

## Class Instantiation

Instantiation class means creating an object from a class. We need the keyword *new* and we call the name of the class after the word new.

Let us create a dog object from our Person class.

```
class Person {
  // code goes here
}
const person = new Person()
console.log(person)


Person {}
```

As you can see, we have created a person object. Since the class did not have any properties yet the object is also empty.

Let use the class constructor to pass different properties for the class.

## Class Constructor

The constructor is a builtin function which allows as to create a blueprint for our object. The constructor function starts with a keyword constructor followed by a parenthesis. Inside the parenthesis we pass the properties of the object as parameter. We use the *this* keyword to attach the constructor parameters with the class.

The following Person class constructor has firstName and lastName property. These properties are attached to the Person class using *this* keyword. *This* refers to the class itself.

```
class Person {
  constructor(firstName, lastName) {
    console.log(this) // Check the output from here
    this.firstName = firstName
    this.lastName = lastName
  }
}

const person = new Person()

console.log(person)


Person {firstName: undefined, lastName}
```

All the keys of the object are undefined. When ever we instantiate we should pass the value of the properties. Let us pass value at this time when we instantiate the class.

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh')

console.log(person1)


Person {firstName: "Asabeneh", lastName: "Yetayeh"}
```

As we have stated at the very beginning that once we create a class we can create many object using the class. Now, let us create many person objects using the Person class.

```
class Person {
  constructor(firstName, lastName) {
    console.log(this) // Check the output from here
    this.firstName = firstName
    this.lastName = lastName
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh')
const person2 = new Person('Lidiya', 'Tekle')
const person3 = new Person('Abraham', 'Yetayeh')

console.log(person1)
console.log(person2)
console.log(person3)


Person {firstName: "Asabeneh", lastName: "Yetayeh"}
Person {firstName: "Lidiya", lastName: "Tekle"}
Person {firstName: "Abraham", lastName: "Yetayeh"}
```

Using the class Person we created three persons object. As you can see our class did not many properties let us add more properties to the class.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    console.log(this) // Check the output from here
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')

console.log(person1)


Person {firstName: "Asabeneh", lastName: "Yetayeh", age: 250, country: "Finland", city: "Helsink
```

## Default values with constructor

The constructor function properties may have a default value like other regular functions.

```
class Person {
  constructor(
    firstName = 'Asabeneh',
    lastName = 'Yetayeh',
    age = 250,
    country = 'Finland',
    city = 'Helsinki'
  ) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
  }
}

const person1 = new Person() // it will take the default values
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')

console.log(person1)
console.log(person2)
```

```
Person {firstName: "Asabeneh", lastName: "Yetayeh", age: 250, country: "Finland", city: "Helsink
Person {firstName: "Lidiya", lastName: "Tekle", age: 28, country: "Finland", city: "Espoo"}
```

## Class methods

The constructor inside a class is a builtin function which allow us to create a blueprint for the object. In a class we can create class methods. Methods are JavaScript functions inside the class. Let us create some class methods.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')

console.log(person1.getFullName())
console.log(person2.getFullName())


Asabeneh Yetayeh
test.js:19 Lidiya Tekle
```

## Properties with initial value

When we create a class for some properties we may have an initial value. For instance if you are playing a game, you starting score will be zero. So, we may have a starting score or score which is zero. In other way, we may have an initial skill and we will acquire some skill after some time.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
    this.score = 0
    this.skills = []
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')

console.log(person1.score)
console.log(person2.score)

console.log(person1.skills)
console.log(person2.skills)


0
0
[]
[]
```

A method could be regular method or a getter or a setter. Let us see, getter and setter.

## getter

The get method allow us to access value from the object. We write a get method using keyword *get* followed by a function. Instead of accessing properties directly from the object we use getter to get the value. See the example bellow

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
    this.score = 0
    this.skills = []
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
  get getScore() {
    return this.score
  }
  get getSkills() {
    return this.skills
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')

console.log(person1.getScore) // We do not need parenthesis to call a getter method
console.log(person2.getScore)

console.log(person1.getSkills)
console.log(person2.getSkills)


0
0
[]
[]
```

## setter

The setter method allow us to modify the value of certain properties. We write a setter method using keyword *set* followed by a function. See the example bellow.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
    this.score = 0
    this.skills = []
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
  get getScore() {
    return this.score
  }
  get getSkills() {
    return this.skills
  }
  set setScore(score) {
    this.score += score
  }
  set setSkill(skill) {
    this.skills.push(skill)
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')

person1.setScore = 1
person1.setSkill = 'HTML'
person1.setSkill = 'CSS'
person1.setSkill = 'JavaScript'

person2.setScore = 1
person2.setSkill = 'Planning'
person2.setSkill = 'Managing'
person2.setSkill = 'Organizing'

console.log(person1.score)
console.log(person2.score)

console.log(person1.skills)
console.log(person2.skills)
```

```
1
1
["HTML", "CSS", "JavaScript"]
["Planning", "Managing", "Organizing"]
```

Do not be puzzled by the difference between regular method and a getter. If you know how to make a regular method you are good. Let us add regular method called getPersonInfo in the Person class.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
    this.score = 0
    this.skills = []
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
  get getScore() {
    return this.score
  }
  get getSkills() {
    return this.skills
  }
  set setScore(score) {
    this.score += score
  }
  set setSkill(skill) {
    this.skills.push(skill)
  }
  getPersonInfo() {
    let fullName = this.getFullName()
    let skills =
      this.skills.length > 0 &&
      this.skills.slice(0, this.skills.length - 1).join(', ') +
        ` and ${this.skills[this.skills.length - 1]}`
    let formattedSkills = skills ? `He knows ${skills}` : ''

    let info = `${fullName} is ${this.age}. He lives ${this.city}, ${this.country}. ${formattedS
    return info
  }
}

const person1 = new Person('Asabeneh', 'Yetayeh', 250, 'Finland', 'Helsinki')
const person2 = new Person('Lidiya', 'Tekle', 28, 'Finland', 'Espoo')
const person3 = new Person('John', 'Doe', 50, 'Mars', 'Mars city')

person1.setScore = 1
person1.setSkill = 'HTML'
person1.setSkill = 'CSS'
person1.setSkill = 'JavaScript'

person2.setScore = 1
person2.setSkill = 'Planning'
```

```
person2.setSkill = 'Managing'
person2.setSkill = 'Organizing'

console.log(person1.getScore)
console.log(person2.getScore)

console.log(person1.getSkills)
console.log(person2.getSkills)
console.log(person3.getSkills)

console.log(person1.getPersonInfo())
console.log(person2.getPersonInfo())
console.log(person3.getPersonInfo())
```

```
1
1
["HTML", "CSS", "JavaScript"]
["Planning", "Managing", "Organizing"]
[]
Asabeneh Yetayeh is 250. He lives Helsinki, Finland. He knows HTML, CSS and JavaScript
Lidiya Tekle is 28. He lives Espoo, Finland. He knows Planning, Managing and Organizing
John Doe is 50. He lives Mars city, Mars.
```

## Static method

The static keyword defines a static method for a class. Static methods are not called on instances of the class. Instead, they are called on the class itself. These are often utility functions, such as functions to create or clone objects. An example of static method is *Date.now()*. The *now* method is called directly from the class.

```
class Person {
  constructor(firstName, lastName, age, country, city) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.country = country
    this.city = city
    this.score = 0
    this.skills = []
  }
  getFullName() {
    const fullName = this.firstName + ' ' + this.lastName
    return fullName
  }
  get getScore() {
    return this.score
  }
  get getSkills() {
    return this.skills
  }
  set setScore(score) {
    this.score += score
  }
  set setSkill(skill) {
    this.skills.push(skill)
  }
  getPersonInfo() {
    let fullName = this.getFullName()
    let skills =
      this.skills.length > 0 &&
      this.skills.slice(0, this.skills.length - 1).join(', ') +
        ` and ${this.skills[this.skills.length - 1]}`

    let formattedSkills = skills ? `He knows ${skills}` : ''

    let info = `${fullName} is ${this.age}. He lives ${this.city}, ${this.country}. ${formattedS
    return info
  }
  static favoriteSkill() {
    const skills = ['HTML', 'CSS', 'JS', 'React', 'Python', 'Node']
    const index = Math.floor(Math.random() * skills.length)
    return skills[index]
  }
  static showDateTime() {
    let now = new Date()
    let year = now.getFullYear()
    let month = now.getMonth() + 1
    let date = now.getDate()
    let hours = now.getHours()
    let minutes = now.getMinutes()
```

```
    if (hours < 10) {
      hours = '0' + hours
    }
    if (minutes < 10) {
      minutes = '0' + minutes
    }

    let dateMonthYear = date + '.' + month + '.' + year
    let time = hours + ':' + minutes
    let fullTime = dateMonthYear + ' ' + time
    return fullTime
  }
}

console.log(Person.favoriteSkill())
console.log(Person.showDateTime())


Node
15.1.2020 23:56
```

The static methods are methods which can be used as utility functions.

## Inheritance

Using inheritance we can access all the properties and the methods of the parent class. This reduces repetition of code. If you remember, we have a Person parent class and we will create children from it. Our children class could be student, teach etc.

```
// syntax
class ChildClassName extends {
 // code goes here
}
```

Let us create a Student child class from Person parent class.

```
class Student extends Person {
  saySomething() {
    console.log('I am a child of the person class')
  }
}

const s1 = new Student('Asabeneh', 'Yetayeh', 'Finland', 250, 'Helsinki')
console.log(s1)
console.log(s1.saySomething())
console.log(s1.getFullName())
console.log(s1.getPersonInfo())
```

```
Student {firstName: "Asabeneh", lastName: "Yetayeh", age: "Finland", country: 250, city: "Helsir
I am a child of the person class
Asabeneh Yetayeh
Student {firstName: "Asabeneh", lastName: "Yetayeh", age: "Finland", country: 250, city: "Helsir
Asabeneh Yetayeh is Finland. He lives Helsinki, 250.
```

## Overriding methods

As you can see, we manage to access all the methods in the Person Class and we used it in the Student child class. We can customize the parent methods, we can add additional properties to a child class. If we want to customize, the methods and if we want to add extra properties, we need to use the constructor function the child class too. In side the constructor function we call the super() function to access all the properties from the parent class. The Person class didn't have gender but now let us give gender property for the child class, Student. If the same method name used in the child class, the parent method will be overridden.

```
class Student extends Person {
  constructor(firstName, lastName, age, country, city, gender) {
    super(firstName, lastName, age, country, city)
    this.gender = gender
  }

  saySomething() {
    console.log('I am a child of the person class')
  }
  getPersonInfo() {
    let fullName = this.getFullName()
    let skills =
      this.skills.length > 0 &&
      this.skills.slice(0, this.skills.length - 1).join(', ') +
        ` and ${this.skills[this.skills.length - 1]}`

    let formattedSkills = skills ? `He knows ${skills}` : ''
    let pronoun = this.gender == 'Male' ? 'He' : 'She'

    let info = `${fullName} is ${this.age}. ${pronoun} lives in ${this.city}, ${this.country}. $
    return info
  }
}

const s1 = new Student(
  'Asabeneh',
  'Yetayeh',
  250,
  'Finland',
  'Helsinki',
  'Male'
)
const s2 = new Student('Lidiya', 'Tekle', 28, 'Finland', 'Helsinki', 'Female')
s1.setScore = 1
s1.setSkill = 'HTML'
s1.setSkill = 'CSS'
s1.setSkill = 'JavaScript'

s2.setScore = 1
s2.setSkill = 'Planning'
s2.setSkill = 'Managing'
s2.setSkill = 'Organizing'

console.log(s1)

console.log(s1.saySomething())
console.log(s1.getFullName())
console.log(s1.getPersonInfo())

console.log(s2.saySomething())
```

```
console.log(s2.getFullName())
console.log(s2.getPersonInfo())
```

```
Student {firstName: "Asabeneh", lastName: "Yetayeh", age: 250, country: "Finland", city: "Helsir
Student {firstName: "Lidiya", lastName: "Tekle", age: 28, country: "Finland", city: "Helsinki",
I am a child of the person class
Asabeneh Yetayeh
Student {firstName: "Asabeneh", lastName: "Yetayeh", age: 250, country: "Finland", city: "Helsir
Asabeneh Yetayeh is 250. He lives in Helsinki, Finland. He knows HTML, CSS and JavaScript
I am a child of the person class
Lidiya Tekle
Student {firstName: "Lidiya", lastName: "Tekle", age: 28, country: "Finland", city: "Helsinki",
Lidiya Tekle is 28. She lives in Helsinki, Finland. He knows Planning, Managing and Organizing
```

Now, the getPersonInfo method has been overridden and it identifies if the person is male or female.

## Exercises

### Exercises Level 1

1. Create an Animal class. The class will have name, age, color, legs properties and create different methods
2. Create a Dog and Cat child class from the Animal Class.

### Exercises Level 2

1. Override the method you create in Animal class

### Exercises Level 3

1. Let's 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 a class called Statistics and create all the functions which do statistical calculations as method for the Statistics class. Check the output below.

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

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,


// you output should look like this
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, 33), (8
```

# 13 Document Object Model(DOM)

HTML document is structured as a JavaScript Object. Every HTML element has a different properties which can help us to manipulate it. It is possible to get, create, append or remove HTML elements using JavaScript.

When it comes to React we do not directly manipulate the DOM instead React Virtual DOM will take care of update all necessary changes.

So do not directly manipulate the DOM if you are using react. The only place we directly touch the DOM is here at the index.html. React is a single page application because all the components will be rendered on the index.html page and there will not be any other HTML in the entire React Application. You don't have to know DOM very well to use react but recommended to know.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>

  <body>
    <!-- <div class="root"></div> -->
    <div id="root"></div>

    <script>
      // const root = document.querySelector('.root')
      // const root = document.getElementById('root')
      const root = document.querySelector('#root')
      root.innerHTML = <h1>Welcome to 30 Days Of React </h1>
    </script>
  </body>
</html>
```

Check out there result on codepen

🌕 You are amazing! You have just completed day 1 challenge and you are on your way to greatness. Now you are a JavaScript Ninja and ready to dive into React.

🎉 CONGRATULATIONS ! 🎉

<< Day 0 | Day 2 >>