# JS_Beginning_to_Mastery_Part1_2

April 21, 2023

### 0.0.1 Object destructuring

[1]:
```
const band = {
    bandName: "The Beatles",
    members: 4,
    genre: "Rock",
    famousSong: "Yesterday",
    famousAlbum: "Abbey Road"
};

let {bandName, members, ...restProps} = band;
console.log(bandName);
console.log(restProps);
```

```
The Beatles
{ genre: 'Rock', famousSong: 'Yesterday', famousAlbum: 'Abbey Road' }
```

### 0.0.2 Objects inside arrays

[2]:
```
const users = [
    {userId: 1, name: 'John', age: 25},
    {userId: 2, name: 'Mary', age: 30},
    {userId: 3, name: 'Peter', age: 28}
];
```

[3]:
```
for(let user of users){
    console.log(user.name);
}
```

```
John
Mary
Peter
```

### 0.0.3 Nested destructuring

[4]:
```
const users2 = [
    {"userId": 1, "name": "John", "age": 25, "address": {"city": "New York",␣
 ↪"state": "NY"}},
```

```
    {"userId": 2, "name": "Mary", "age": 30, "address": {"city": "Boston",␣
  ↪"state": "MA"}},
    {"userId": 3, "name": "Peter", "age": 28, "address": {"city": "Chicago",␣
  ↪"state": "IL"}}
];
```

[5]:
```
const [Myuser1, Myuser2, Myuser3] = users2;
console.log(Myuser2)
```

```
{
  userId: 2,
  name: 'Mary',
  age: 30,
  address: { city: 'Boston', state: 'MA' }
}
```

But suppose, we need only the name of user1 and age of user3, we can use nested destructuring

[6]:
```
const[{name}, , {age}] = users2;
```

[7]:
```
console.log(name)
```

```
John
```

[8]:
```
console.log(age);
```

```
28
```

We can also change the name of the variables

[9]:
```
const[{name: myName}, , {age:myAge}] = users2;
console.log(myName);
console.log(myAge);
```

```
John
28
```

Suppose, we need the userId and name of user1 and age of user 3

[10]:
```
const[{name: myName2, userId}, , {age: myAge2}] = users2;
console.log(myName2);
console.log(userId);
console.log(myAge2);
```

```
John
1
28
```

Returns the index at which the target is found if the target is not found, return -1

[11]:
```
function findTarget(array, target){
    for(let i = 0; i<array.length; i++){
        if(array[i]===target){
            return i;
        }
    }
    return -1;
}
const myArray = [1,3,8,90]
const ans = findTarget(myArray, 4);
```

[12]:
```
console.log(ans);
```

```
-1
```

[13]:
```
console.log(findTarget(myArray, 3));
```

```
1
```

#### 0.0.4 Arrow function – =>

[14]:
```
// conventional function -- multiple arguments
const sumThreeNumbers = function(a, b, c){
    return a + b + c;
}
```

[15]:
```
// arrow function -- multiple arguments
const sumThreeNumbers2 = (a, b, c) => {
    return a + b + c;
};
```

[16]:
```
console.log(sumThreeNumbers2(1,2,3));
```

```
6
```

[17]:
```
// arrow function -- single argument
const even = num => num % 2 === 0;
console.log(even(5));
```

```
false
```

[18]:
```
// arrow function -- single argument -- continued
const square = (num) => {
    return num * num
};
console.log(square(5));
```

```
25
```

[19]:
```
// arrow function -- single argument -- continued
const square2 = num => num * num; // don't use return if it's a single line or␣
 ↪there is no parenthesis
console.log(square2(5));
```

25

[20]:
```
// arrow function
const firstChar = anyString => anyString[0];
console.log(firstChar("hello"));
```

h

[21]:
```
// arrow function -- no arguments
const sayHello = () => "hello";
console.log(sayHello());
```

hello

[24]:
```
// arrow function -- no arguments -- continued
const sayHello3 = () => {
    return "hello";
};
console.log(sayHello3());
```

hello

[25]:
```
// without arrow function
function findTarget(array, target){
    for(let i = 0; i<array.length; i++){
        if(array[i]===target){
            return i;
        }
    }
    return -1;
}
```

[26]:
```
// arrow function
const findTarget2 = (array, target) => {
    for(let i=0; i <array.length; i++){
        if(array[i]==target){
            return i;
        }
    }
    return -1;
};
```

[29]:
```
const myArray4 = [1,3,8,90];
const ans4 = findTarget(myArray4, 4);
```

```
console.log(ans4);
```

```
-1
```

[30]:
```
const ans5= findTarget2(myArray4, 4);
console.log(ans5);
```

```
-1
```

[31]:
```
const ans6= findTarget2(myArray4, 8);
console.log(ans6);
```

```
2
```

[32]:
```
const ans7= findTarget2(myArray4, 8);
console.log(ans7);
```

```
2
```

### 0.0.5 Hoisting

**Hoisting for variables**

### 0.0.6 var – variable can be accessed before declared

**undefined will be returned if hoisted** Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their respective scope before code execution. This means that regardless of where variables and functions are declared in the code, they are interpreted as if they were declared at the beginning of their scope.

For example, consider the following code snippet:

[33]:
```
console.log(a);
var a = 10;
```

```
undefined
```

Even though a is declared after the `console.log()` statement, the code will still execute without throwing an error. This is because of hoisting. The code is interpreted by the JavaScript engine as follows:

```
var a;
console.log(a);
a = 10;
```

As you can see, the variable declaration `var a;` is moved to the top of the scope, so when the `console.log(a)` statement is executed, `a` has already been declared (but not yet assigned a value), so it does not throw an error.

It's important to note that only the declaration of the variable or function is hoisted, not the initialization or assignment. So, in the example above, `a` is declared at the top of its scope, but its value is not assigned until the second line of code.

Hoisting can be a helpful feature of JavaScript, but it can also lead to confusion and bugs if not used properly. It's best practice to declare all variables at the top of their respective scope to avoid unexpected behavior.

[40]:
```
console.log(y);

// variable declaration
var y = 5; // with var undefined will be returned when this variable is called␣
 ↪before this line
```

```
undefined
```

**let – variable can't be accessed before declared** With `let`, error will be thrown when this variable is called before this line

```
console.log(x); // will throw error

// variable declaration
let x = 5;
```

### Hoisting for functions

**Hoisting will work for function declaration** Hoisting also applies to functions in JavaScript. Function declarations are hoisted to the top of their respective scope, which means that they can be called before they are defined in the code.

For example, consider the following code snippet:

[34]:
```
foo();

function foo() {
  console.log("Hello, world!");
}
```

```
Hello, world!
```

Even though the `foo()` function is called before its definition, the code will still execute without throwing an error. This is because of hoisting. The code is interpreted by the JavaScript engine as follows:

[35]:
```
function foo() {
  console.log("Hello, world!");
}

foo();
```

```
Hello, world!
```

As you can see, the function declaration `function foo() {...}` is moved to the top of the scope, so when the foo() function is called, it has already been declared.

[38]:
```
// hoisting
// will work for function declaration
sayHello4();

// function declaration
function sayHello4(){
    console.log("hello");
}
```

```
hello
```

It's important to note that only function declarations are hoisted, not function expressions. Function expressions are created when a function is assigned to a variable, and they are not hoisted. For example:

```
bar(); // Throws an error

var bar = function() {
  console.log("Hello, world!");
};
```

In this example, the `bar()` function is defined as a function expression and assigned to the variable bar. Since function expressions are not hoisted, the code throws an error when `bar()` is called before its definition.

It's best practice to declare all functions at the top of their respective scope, or to use function expressions and assign them before they are called. This can help avoid unexpected behavior and errors in your code.

**Hoisting will not work for function expression**

```
hello2(); // this will not work

const hello2 = function(){
    console.log("hello");
};
```

**Hoisting will not work for arrow function**

```
hello3(); // this will not work

const hello3 = () => {
    console.log("hello");
};
```

### 0.0.7 functions inside a function

[41]:
```
function app(){
    const myFunc = () => {
        console.log("Hello from myFunc");
    };
```

```
    const addTwoNumbers = (a, b) => {
        return a + b;
    };

    const mul = (num1, num2) => num1*num2;

    console.log("inside app");

    myFunc();
    console.log(addTwoNumbers(2, 3));
    console.log(mul(5, 6));
}
app();
```

```
inside app
Hello from myFunc
5
30
```

### 0.0.8 Lexical scoping

Lexical scoping is a concept in programming languages that describes how variable scope is determined at the time of writing code, rather than when it is executed. In JavaScript, lexical scoping means that a variable's scope is determined by its location in the code, or more specifically, by where it is declared.

When a function is defined in JavaScript, it creates a new scope for the variables declared inside it. This is known as the function's closure. The variables declared inside the function are available only within the function itself, and any nested functions or closures declared inside the function.

In other words, when a function is defined, it captures the values of any variables in its enclosing scope, and those values are available to the function when it is executed, even if the variables themselves are no longer in scope.

Here's an example:

[42]:
```
function outerFunction() {
  var outerVar = 'I am declared in outerFunction';

  function innerFunction() {
    console.log(outerVar);
  }

  innerFunction();
}

outerFunction(); // Output: "I am declared in outerFunction"
```

```
I am declared in outerFunction
```

In this example, `outerFunction` declares a variable called `outerVar`. `innerFunction` is defined inside `outerFunction`, which means it has access to `outerVar` because of lexical scoping. When `outerFunction` is executed, it calls `innerFunction`, which outputs the value of `outerVar` to the console.

Lexical scoping is an important concept in JavaScript, and it helps to avoid naming conflicts and unintended consequences that can occur when variables are not properly scoped.

[43]:
```
function myApp(){
    const myVar = "value1";
    function myFunc(){
        const myVar = "value 60";
        // but if we comment out the above line, then the value of myVar will␣
 ↪be "value1"
        console.log("inside myFunc", myVar);
    };
    const myFunc2 = function(){};
    const myFunc3 = () => {};
    console.log(myVar); // prints "value1"
    myFunc(); // prints "inside myFunc value 60"
}

myApp();
```

```
value1
inside myFunc value 60
```

[44]:
```
// lexical scoping
const myVar = "value 1";

function myApp2(){
    function myFunc(){
        // the value of myVar will be "value 1"
        console.log("inside myFunc", myVar);
    };
    const myFunc2 = function(){};
    const myFunc3 = () => {};
    console.log(myVar);
    myFunc();
}

myApp2();
```

```
value 1
inside myFunc value 1
```

[1]:
```
// lexical scoping
const myVar2 = "value 2";
```

```
function myApp3(){
    function myFunc(){
        const myFunc2 = () => {
            console.log("inside myFunc", myVar2); // the value of myVar will be␣
 ↪"value 2"
        };
        myFunc2();
    };
    console.log(myVar2);
    myFunc();
}

myApp3();
```

```
value 2
inside myFunc value 2
```

### 0.0.9 Block scope vs Function scope

`let` and `const` are block scoped, `var` is function scoped

In JavaScript, variables can be declared with either function scope or block scope.

Function scope means that variables declared inside a function are only accessible within that function and any nested functions or closures defined within it. Here's an example:

[1]:
```
function exampleFunction() {
  var a = 1;
  if (a === 1) {
    var b = 2;
    console.log(b); // Output: 2
  }
  console.log(b); // Output: 2
}

exampleFunction();
```

```
2
2
```

In this example, `a` is declared inside `exampleFunction` and is only accessible within that function. `b` is also declared inside `exampleFunction`, but since it uses `var`, it has function scope, which means it is accessible within the entire function, including the `if` block.

Block scope means that variables declared inside a block of code (i.e., within curly braces `{}`) are only accessible within that block and any nested blocks. Block scope is achieved using the `let` and `const` keywords, which were introduced in ES6. Here's an example:

[2]:
```
function exampleFunction() {
  let a = 1;
```

```
  if (a === 1) {
    const b = 2;
    console.log(b); // Output: 2
  }
  console.log(b); // Output: ReferenceError: b is not defined
}

exampleFunction();
```

```
2
```

```
evalmachine.<anonymous>:7
  console.log(b); // Output: ReferenceError: b is not defined
              ^


ReferenceError: b is not defined
    at exampleFunction (evalmachine.<anonymous>:7:15)
    at evalmachine.<anonymous>:10:1
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
↪21)
```

In this example, `a` is declared using `let`, which means it has block scope and is only accessible within the block where it is defined. `b` is declared using `const`, which also has block scope, but cannot be reassigned once it is defined.

To summarize, function scope means that variables declared with `var` are accessible within the entire function, while block scope means that variables declared with `let` or `const` are only accessible within the block where they are defined.

[3]:
```
{
    let a = 10;
    console.log(a); // this will work
}
// console.log(a); // will throw error
```

```
10
```

[4]:
```
{
    let a = 20;
    console.log(a); // this will work
}
```

```
20
```

[5]:
```
{
    const a2 = 10;
    console.log(a2); // this will work
}
// console.log(a2); // will throw error
```

```
10
```

[6]:
```
{
    const a2 = 20;
    console.log(a2); // this will work
}
// console.log(a2); // will throw error
```

```
20
```

[7]:
```
const a2 = 100;
console.log(a2);
```

```
100
```

[8]:
```
// for var, we can access the variable outside the block
{
    var a3 = 200;
    console.log(a3); // this will work
}
console.log(a3); // this will work too
```

```
200
200
```

[9]:
```
// var can be accessed outside the block
// var can be accessed anywhere
{
    var firstName = "John";
    console.log(firstName);
}

{
    console.log(firstName); // it can access the previous block
}
```

```
John
John
```

### 0.0.10 Rest parameters

[10]:
```
function myFunc(a,b, ...c){
    console.log(`a is ${a}`);
    console.log(`b is ${b}`);
    console.log(`c is ${c}`);
}

myFunc(3,4,5,6,7,8,9,10);
```

```
a is 3
b is 4
c is 5,6,7,8,9,10
```

[11]:
```
function addAll(...numbers){
    console.log(numbers);
    console.log(Array.isArray(numbers));
    let total = 0;
    for(let number of numbers){
        total = total + number;
    }
    return total;
}

const ans = addAll(1,2,3,4,5,6,7,8,9,10);
console.log(ans);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
true
55
```

### 0.0.11 Parameter destructuring in JavaScript

Parameter destructuring is a feature in JavaScript that allows you to extract values from objects and arrays passed as function arguments and assign them to variables with the same name as the object's keys or array's elements. This can be done in the function signature using curly braces `{}` for objects or square brackets `[]` for arrays.

Here's an example of destructuring an object passed as a parameter:

[12]:
```
function exampleFunction({a, b}) {
  console.log(a, b);
}

const obj = {a: 1, b: 2};
exampleFunction(obj); // Output: 1 2
```

```
1 2
```

In this example, the `exampleFunction` takes an object as its parameter and destructures it using curly braces in the function signature. This creates new variables `a` and `b` and assigns them the values of the corresponding properties in the object passed as the argument.

Here's an example of destructuring an array passed as a parameter:

[13]:
```
function exampleFunction([a, b]) {
  console.log(a, b);
}

const arr = [1, 2];
exampleFunction(arr); // Output: 1 2
```

```
1 2
```

In this example, the `exampleFunction` takes an array as its parameter and destructures it using square brackets in the function signature. This creates new variables `a` and `b` and assigns them the values of the corresponding elements in the array passed as the argument.

Parameter destructuring can make your code more concise and readable, especially when dealing with complex objects or arrays passed as function arguments. However, it's important to use it judiciously and avoid excessive nesting or complexity.

[14]:
```
const person = {
    firstName: "John",
    gender: "male"
};
```

[15]:
```
// without param destructuring
function printDetails(obj){
    console.log(obj.firstName);
    console.log(obj.gender)
}

printDetails(person);
```

```
John
male
```

[16]:
```
// we can use destructuring to make the code more concise
function printDetails2({firstName, gender}){
    console.log(firstName);
    console.log(gender);
}

printDetails2(person);
```

```
John
male
```

### 0.0.12 Callback function in JavaScript

In JavaScript, a callback function is a function that is passed as an argument to another function and is called when the other function has finished executing. The purpose of a callback function is to allow asynchronous processing of data or events, without blocking the execution of other code.

Here's an example:

[ ]:
```
function fetchData(callback) {
  // Some asynchronous operation to fetch data
  const data = [1, 2, 3, 4, 5];
  callback(data);
}

function process(data) {
  console.log(data.map(x => x * 2));
}

fetchData(process); // Output: [2, 4, 6, 8, 10]
```

In this example, the `fetchData` function takes a callback function as its argument, which is called with the fetched data when it's available. The `process` function is defined separately and takes the data as its argument, processes it, and outputs the result to the console.

When `fetchData` is called, it initiates an asynchronous operation to fetch data and then calls the `callback` function with the fetched data when it's available. The `process` function is passed as the callback function, and it's executed with the fetched data as its argument.

Callbacks are widely used in JavaScript for a variety of purposes, including event handling, data fetching, and asynchronous programming. They allow for more flexible and modular code by decoupling the execution of functions from their definition.

[1]:
```
function myFunc(a){
    console.log(a);
    console.log("hello world!");
}
myFunc([1,2,3]);
myFunc("abc");
```

```
[ 1, 2, 3 ]
hello world!
abc
hello world!
```

[2]:
```
function myFunc2(){
    console.log("Inside myFunc2()");
}

function myFunc3(a){
    console.log(a);
```

```
}

myFunc3(myFunc2); // returns the function myFunc2() itself
```

```
[Function: myFunc2]
```

[3]:
```
// we can pass a function as a parameter to another function
function myFunc4(a){
    // console.log(a);
    console.log("Inside myFunc4()");
    a();
}

myFunc4(myFunc2); // callback function
```

```
Inside myFunc4()
Inside myFunc2()
```

[4]:
```
function myFunc5(callback){
    callback();
}

function myFuncTest(name){
    console.log("inside my func test");
    console.log(`Your name is ${name}`);
}

myFunc5(myFuncTest);
myFunc5(myFuncTest("John"));
```

```
inside my func test
Your name is undefined
inside my func test
Your name is John
```

```
evalmachine.<anonymous>:2
    callback();
    ^

TypeError: callback is not a function
    at myFunc5 (evalmachine.<anonymous>:2:5)
    at evalmachine.<anonymous>:11:1
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
```

```
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
↪21)
```

### 0.0.13 function returning function

[5]:
```
function myFunc(){
    function hello(){
        console.log("Hello world!");
    }
    return hello;
}

const ans = myFunc();
ans()
```

```
Hello world!
```

[6]:
```
console.log(ans());
```

```
Hello world!
undefined
```

[7]:
```
function myFunc2(){
    return function(){
        return "Hello world!";
    };
}

const ans2 = myFunc2();
console.log(ans2()); // returns "Hello world!"
```

```
Hello world!
```

### 0.0.14 Important array methods

[2]:
```
const numbers = [4, 3, 5, 6, 9];

function multiplyBy2(number, index){
    //console.log(`index is ${index} number is ${number}`);
    console.log(`${number}*2 = ${number*2}`);
}

multiplyBy2(numbers[0], 0);
multiplyBy2(numbers[1], 1);
```

```
4*2 = 8
3*2 = 6
```

[3]:
```
for(let i=0; i < numbers.length; i++){
    //console.log(i);
    multiplyBy2(numbers[i], i);
}

// using forEach method -- forEach method takes a callback function as a␣
 ↪parameter
// similar to the previous one
numbers.forEach(multiplyBy2);
```

```
4*2 = 8
3*2 = 6
5*2 = 10
6*2 = 12
9*2 = 18
4*2 = 8
3*2 = 6
5*2 = 10
6*2 = 12
9*2 = 18
```

[4]:
```
// define callback function inside the forEach method
numbers.forEach(function(number, index){
    console.log(`index is ${index} number is ${number}`);
})
```

```
index is 0 number is 4
index is 1 number is 3
index is 2 number is 5
index is 3 number is 6
index is 4 number is 9
```

[6]:
```
// define callback function inside the forEach method -- another example
numbers.forEach(function(number, index){
    console.log(`${number}*10 = ${number*10}`);
})
```

```
4*10 = 40
3*10 = 30
5*10 = 50
6*10 = 60
9*10 = 90
```

[7]:
```
// forEach with array of objects
const users = [
    { name: "John", age: 30, city: "New York" },
    { name: "Jane", age: 25, city: "Boston" },
    { name: "Jack", age: 40, city: "Miami" }
];
```

```
users.forEach(function(user){
    console.log(user.name);
})
```

```
John
Jane
Jack
```

[8]:
```
// the same above functionality using for loop
for(let i=0; i < users.length; i++){
    console.log(users[i].name);
}
```

```
John
Jane
Jack
```

[9]:
```
// forEach using arrow function
// can have both user and index
users.forEach((user, index)=>{
    console.log(user.name, index);
});
```

```
John 0
Jane 1
Jack 2
```

[10]:
```
// can have only user and not index
users.forEach((user) => {
    console.log(user.name);
})
```

```
John
Jane
Jack
```

### 0.0.15 map

In JavaScript, `map()` is a built-in function that is used to transform an array by applying a function to each of its elements. The `map()` function creates a new array with the same number of elements as the original array, but with each element transformed according to the function passed as an argument.

`map()` is similar to `forEach()`.

[1]:
```
const num = [3,4,5,6,7,8,9,10];
```

[2]:
```
// using forEach
const square = function(number){
```

```
    console.log(number*2);
}
num.forEach(square)
```

```
6
8
10
12
14
16
18
20
```

[3]:
```
// using map
const square2 = function(number){
    return number*2;
}
console.log(num.map(square2));
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[4]:
```
const squareNumber = num.map(square2); // returns an array
console.log(squareNumber);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[5]:
```
const squareNumber2 = num.map((number)=>{return number*2})
console.log(squareNumber2);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[6]:
```
const squareNumber3 = num.map((number)=> number*2)
console.log(squareNumber3);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[7]:
```
const users = [
    { name: "John", age: 30, city: "New York" },
    { name: "Jane", age: 25, city: "Boston" },
    { name: "Jack", age: 40, city: "Miami" },
    { name: "Jill", age: 35, city: "Los Angeles" },
    { name: "Joe", age: 20, city: "Chicago" }
]
```

[8]:
```
const userNames = users.map((user)=>{
    return user.name;
});
console.log(userNames);
```

```
[ 'John', 'Jane', 'Jack', 'Jill', 'Joe' ]
```

[10]:
```
const userNames3 = users.map((user)=> user.name)
console.log(userNames3);
```

```
[ 'John', 'Jane', 'Jack', 'Jill', 'Joe' ]
```

### 0.0.16 filter method

[12]:
```
const numbers6 = [1,2,3,4,5,6,7,8,9,10];

const evenNumber = numbers6.filter((number)=> number%2 == 0)
console.log(evenNumber);
```

```
[ 2, 4, 6, 8, 10 ]
```

We can also do the same without using arrow function.

[13]:
```
const isEven = function(number){
    return number%2 == 0;
}
console.log(numbers6.filter(isEven))
```

```
[ 2, 4, 6, 8, 10 ]
```

### 0.0.17 reduce method

The `reduce()` method is a built-in function in JavaScript that is used to reduce an array of values to a single value. It takes in a callback function as its first argument and an optional initial value as the second argument.

The callback function is executed on each element of the array and has access to two arguments: an accumulator and the current value. The accumulator is the value that is returned and passed to the next iteration of the callback function, and it is initialized to the initial value (if provided) or the first element of the array.

The callback function can perform any operation on the accumulator and the current value, such as addition, multiplication, or concatenation. At the end of the iteration, the final value of the

accumulator is returned.

Here's an example of using the `reduce()` method to find the sum of an array of numbers:

[14]:
```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator +␣
 ↪currentValue, 0);

console.log(sum); // Output: 15
```

```
15
```

In this example, the `reduce()` method starts with an initial value of 0 and then adds each element of the `numbers` array to the accumulator. At the end of the iteration, the final value of the accumulator (which is the sum of all the numbers) is returned.

The `reduce()` method can also be used to perform other operations, such as finding the maximum or minimum value in an array, concatenating strings, or even grouping objects based on a common property.

[20]:
```
var numbers12 = [1,2,3,4,5,10];

// sum of all the numbers in the array
var sum12 = numbers12.reduce((accumulator, currentValue) => {
    return accumulator + currentValue;
});

console.log(sum12);
```

```
25
```

When `reduce()` is called without an initial value, the first iteration of the callback function uses the first element of the array as the initial value of the accumulator, and the second element as the current value. In subsequent iterations, the accumulator will be the result of the previous iteration, and the current value will be the next element in the array.

In the example above, the first iteration sets `accumulator` to 1 (the first element in the array), and `currentValue` to 2 (the second element in the array). The callback function then adds these two values together and returns the result (3), which becomes the new value of the accumulator for the next iteration.

This process continues for each element in the array until all elements have been processed, and the final value of the accumulator is returned as the result of the `reduce()` method.

So, even though an initial value is not provided, the `reduce()` method still works and returns the correct sum because it uses the first element of the array as the initial value of the accumulator. However, it's generally recommended to provide an initial value to avoid any unexpected behavior in cases where the array is empty.

[21]:
```
const userCart = [
    { name: "laptop", price: 1000, quantity: 1 },
    { name: "desktop", price: 2000, quantity: 2 },
```

```
    { name: "mobile", price: 500, quantity: 4 },
    { name: "tablet", price: 300, quantity: 3 }
];

const totalAmount = userCart.reduce((totalPrice, currentProduct) => {
    return totalPrice + currentProduct.price;
}, 0);

console.log(totalAmount);
```

3800

### 0.0.18 Sort

[22]:
```
const numbers13 = [5,9,1200,410,3000];
```

[24]:
```
const userNames13 = ['John', 'Jane', 'Jack', 'Jill', 'Joe'];
userNames13.sort();
console.log(userNames13);
```

```
[ 'Jack', 'Jane', 'Jill', 'Joe', 'John' ]
```

[25]:
```
// ascending order
numbers13.sort((a,b)=> a-b)
console.log(numbers13);
```

```
[ 5, 9, 410, 1200, 3000 ]
```

[26]:
```
// descending order
numbers13.sort((a,b)=> b-a);
console.log(numbers13);
```

```
[ 3000, 1200, 410, 9, 5 ]
```

[27]:
```
// sorting array of objects
const products = [
    {productId: 1, productName: "p1", price: 300},
    {productId: 2, productName: "p2", price: 3300},
    {productId: 3, productName: "p3", price: 1200},
    {productId: 4, productName: "p4", price: 300},
    {productId: 5, productName: "p5", price: 1500}
];

// low to high
products.sort((a, b)=>{
    return a.price - b.price;
});
console.log(products);
```

```
[
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 2, productName: 'p2', price: 3300 }
]
```

[28]:
```
// high to low
products.sort((a, b)=>{
    return b.price - a.price;
});
console.log(products);
```

```
[
  { productId: 2, productName: 'p2', price: 3300 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 }
]
```

**Create a new sorted object**

[29]:
```
// But this is changing products and we don't to do that. In
// order to avoid that, we can use slice method

// low to high
const lowToHigh = products.slice(0).sort((a, b) => {
    return a.price - b.price;
});
console.log(lowToHigh);
```

```
[
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 2, productName: 'p2', price: 3300 }
]
```

[30]:
```
// high to low
const highToLow = products.slice(0).sort((a, b) => {
    return b.price - a.price;
});
console.log(highToLow);
```

```
[
  { productId: 2, productName: 'p2', price: 3300 },
  { productId: 5, productName: 'p5', price: 1500 },
```

```
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 }
]
```

### 0.0.19 find method

The `find()` method is a built-in method in JavaScript that is used to search for the first element in an array that satisfies a specified condition. The method returns the value of the first element that passes the test implemented by the provided function.

The syntax of the `find()` method is as follows:

```
array.find(function(currentValue, index, array) {
  // code to test each element of the array
});
```

Here, `array` is the array that the method is called upon, and `currentValue`, `index`, and `array` are optional parameters that represent the current value being processed, the index of the current value, and the array being processed, respectively.

The `find()` method executes the provided function once for each element of the array until it finds an element that satisfies the provided condition. If a matching element is found, the method immediately returns its value, and if no matching element is found, the method returns `undefined`.

[31]:
```
const myArray = ["Hello", "World", "I", "am", "a", "string", "array", "how",␣
 ↪"are"];

function isLength3(string){
    return string.length === 3;
}

const ans = isLength3("dog");
console.log(ans);
```

```
true
```

[32]:
```
const ans2 = myArray.find(isLength3);
console.log(ans2); // returns the first string with length 3, here "how"
```

```
how
```

[33]:
```
const ans3 = myArray.find((string) => string.length === 3);
console.log(ans3);
```

```
how
```

[35]:
```
const users10 = [
    {userId: 1, userName: "John"},
    {userId: 2, userName: "Jane"},
    {userId: 3, userName: "Jack"},
```

```
    {userId: 4, userName: "Jill"},
    {userId: 5, userName: "Joe"},
    {userId: 6, userName: "Jenny"}
];

const myUser10 = users10.find((user)=>user.userId === 3);
console.log(myUser10);
```

```
{ userId: 3, userName: 'Jack' }
```

### 0.0.20 every method

If every element satisfies a condition, returns True

[39]:
```
const numbers14 = [2, 4, 6, 8, 10];

const ans14 = numbers14.every((number)=>number%2 === 0);
console.log(ans14);
```

```
true
```

[41]:
```
function isEven16(number){
    return number % 2 === 0;
}

const ans16 = numbers14.every(isEven);
console.log(ans16);
```

```
true
```

[43]:
```
// every method -- continued
const userCart2 = [
    {productId: 1, quantity: 2, productName: 'laptop', price: 100000},
    {productId: 2, quantity: 1, productName: 'mobile', price: 8000},
    {productId: 3, quantity: 3, productName: 'tablet', price: 30000},
    {productId: 4, quantity: 4, productName: 'watch', price: 2000}
];

const ans17 = userCart2.every((item)=>item.price<200000); // every item price␣
 ↪is less than 200000
console.log(ans17);
```

```
true
```

### 0.0.21 some method

If some elements satisfy a condition, returns True

[44]:
```

```

```
// some method -- returns true or false
const ans18 = userCart2.some((item)=>item.price>80000); // some item price is␣
 ↪greater than 80000
console.log(ans18);
```

```
true
```

### 0.0.22 fill method

Fills an array with a static value from a start index to an end index

[46]:
```
const myArray11 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
myArray11.fill(0, 2, 5); // fill with 0 from index 2 to index 4
console.log(myArray11);
```

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

[47]:
```
const myArray12 = new Array(10).fill(0); // fill with 0 from index 0 to index 9
console.log(myArray12);
```

```
[
  0, 0, 0, 0, 0,
  0, 0, 0, 0, 0
]
```

[48]:
```
const obj3 = {
    key1: "value1",
    key2: "value2"
};

const obj4 = {
    key1: "value3",
    key4: "value4",
    key5: "value5"
};

const newObject3 = {...obj3, ...obj4}; // copies obj3 and obj4 into newObject3
console.log(newObject3);
```

```
{ key1: 'value3', key2: 'value2', key4: 'value4', key5: 'value5' }
```

### 0.0.23 Maps

Map is a collection of key value pairs; duplicate keys are not allowed; store data in ordered fashion.

In Map, we can use any type as key.

**Object literal** // the below is called an object literal

object literal

- key -> string
- value -> any type

[49]:
```
const person = {
    firstName: 'John',
    age: 7,
    1: "one" // 1 will be considered as string
};

console.log(person.firstName);
console.log(person["firstName"]);
```

```
John
John
```

[50]:
```
for(let key in person){
    console.log(typeof key);
}
console.log(person[1]);
```

```
string
string
string
one
```

```
console.log(person.1); // error
```

[51]:
```
// key value pair
// for Map, we can keep any type as key
const person2 = new Map();
person2.set('firstName', 'Munna');
person2.set('age', 33);
person2.set(1, 'one');
person2.set([1,2,3], 'onetwothree');
person2.set({1: 'one'}, 'onetwothree'); // object literal as key

console.log(person2);
```

```
Map(5) {
  'firstName' => 'Munna',
  'age' => 33,
  1 => 'one',
  [ 1, 2, 3 ] => 'onetwothree',
  { '1': 'one' } => 'onetwothree'
}
```

[52]:
```
// Accessing values of Map
console.log(person2.get('age'));
console.log(person2.get(1));
console.log(person2.keys());
```

```
33
one
[Map Iterator] { 'firstName', 'age', 1, [ 1, 2, 3 ], { '1': 'one' } }
```

[53]:
```
// any type of data can be used as key

// we can iterate over keys of the Map

for(let key of person2.keys()){
    console.log(key, typeof key);
}
```

```
firstName string
age string
1 number
[ 1, 2, 3 ] object
{ '1': 'one' } object
```

In object, we couldn't use for of loop, we could use for in loop though but in Map, we can directly use for of loop

[54]:
```
// in object, we couldn't use for of loop, we could use for in loop though
// but in Map, // we can directly use for of loop

const person3 = new Map();
person3.set('firstName', 'Munna');
person3.set('age', 33);
person3.set(1, 'one');

for(let key of person3){
    console.log(Array.isArray(key));
}
```

```
true
true
true
```

**Destructuring**

[55]:
```
// destructuring an array:
let arr = [1, 2, 3, 4, 5];
let [a, b, c] = arr;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

```
1
2
3
```

[56]:
```
// destructuring an object:
let obj = { x: 1, y: 2, z: 3 };
let { x, y, z } = obj;
console.log(x); // 1
console.log(y); // 2
console.log(z); // 3
```

```
1
2
3
```

**Destructuring with map**

[57]:
```
// destructuring with Map

for(let [key, value] of person3){
    console.log(key, value);
}
```

```
firstName Munna
age 33
1 one
```

[58]:
```
// another way to create map
const person4 = new Map([
    ['firstName', 'Munna'],
    ['age', 33]
]);
console.log(person4);
```

```
Map(2) { 'firstName' => 'Munna', 'age' => 33 }
```

### 0.0.24 Add information to object but in a new object using Map

[59]:
```
const person5 = {
    id: 1,
    firstName: 'Munna'
};
const person6 = {
    id: 2,
    firstName: 'Dipti'
};
```

[60]:
```
const extraInfo = new Map();
extraInfo.set(person5, {age: 28, gender: 'Male'});
extraInfo.set(person6, {age: 18, gender: 'Female'});
```

```
console.log(extraInfo);
```

```
Map(2) {
  { id: 1, firstName: 'Munna' } => { age: 28, gender: 'Male' },
  { id: 2, firstName: 'Dipti' } => { age: 18, gender: 'Female' }
}
```

### 0.0.25 Cloning objects

[61]:
```
const obj1 = {
    key1: 'value1',
    key2: 'value2'
};
```

**clone using spread operator**

[62]:
```
// clone using spread operator
const obj2 = {...obj1};
obj1.key3 = "value3";
console.log(obj1);
console.log(obj2);
```

```
{ key1: 'value1', key2: 'value2', key3: 'value3' }
{ key1: 'value1', key2: 'value2' }
```

**cloning using Object.assign**

[65]:
```
// cloning using Object.assign
const obj5 = Object.assign({}, obj1);
obj1.key4 = "value4";
console.log(obj1);
console.log(obj5);
```

```
{ key1: 'value1', key2: 'value2', key3: 'value3', key4: 'value4' }
{ key1: 'value1', key2: 'value2', key3: 'value3' }
```

[ ]:
```

```