Most Essential Concepts of JS

Authors :
Satyapriya Mishra ,
Bhanu Maniktahla

Most Essential Concepts of JavaScript

ABOUT THE BOOK

This eBook is an open source book written for beginners to intermediate JavaScript developers to take their understanding of the language to the next level. This book is freely distributable for education purposes.

The book does not cover the end to end explanation of the concepts of the language, but it tries to touch upon the most important and the quirkiest parts of JavaScript language. The concepts explained in this book are not in any particular order of representation, but they are presented as independent articles/sections to facilitate the user of going through topics of their choice. The examples given in this book are very general and simple in nature so that developers of all experience levels can take benefit from the document. The book is also written keeping in mind the challenges developers face during interviews and complex topics are covered with very simple explanations so that readers don’t have to memorize concepts, but they will derive pleasure by understanding them.

ABOUT THE AUTHORS

Satyapriya Mishra

Satyapriya Mishra is Software Consultant, Technology Trainer, Public Speaker, Author and Blogger with years of experience in building enterprise software applications and mentoring individuals as well as teams. He is very passionate about sharing his knowledge to others and he is often found teaching, mentoring and guiding young professionals to achieve their career objectives. He is the founder of http://code2stepup.com/ where he discusses technical stuff.
He can be reached at http://imsatya.com if you have any interesting project to discuss.

Bhanu Maniktahla

Bhanu is a full stack Software developer, Trainer and Blogger who regularly shares her insights on the most popular digital platforms. She has been training young engineers for a couple of years now, enabling the young minds for a tough corporate journey ahead in their career.
She can be easily found on the internet through her digital footprints shared via blogs and social media channels.

TABLE OF CONTENTS

Most Important Array Methods

Array is a very special concept in any programming language. All the major programming languages support the array data structure. Before going into the technicalities of the array from a programming point of view, let's try to understand what an array is from a common man's perspective.

Have you ever seen a school assembly? It is pretty obvious, you have seen it and in childhood all of us have been part of that boring event. In the assembly hall, students of the same class or section sit together in one row, isn't it. That is a bare minimum real-life example of an array. Another example could be the color pencil box which all of us have used for coloring pictures. In that pretty little box, the items share something in common. They are all used for giving colors, right. So, in the perspective of programming, we can say array is a collection of items which are related to each other.
The arrangement of the items inside an array leads to a particular structure because of which arrays are called a special type of data structure. The term data structure refers to an arrangement of data in such a way that setting, and retrieval of items is very simple, and they follow a certain pattern.

An array essentially can just contain simple data elements like numbers and strings or in more complex scenarios, they can contain complex objects. In javaScript particularly, arrays are hashtable based and they not necessarily are placed in contiguous memory locations. JavaScript arrays are index based and we can set and fetch items based on their indices. JavaScript essentially does not support the concept of associative arrays which means it does not fetch or set array items through keys or simple terms strings.

The simplest way to initialize an array in javaScript is:

```
var arr = [];
```

If you want to store some items, say some numbers in the array, we can initialize it as follows.

```
var arr = [1,2,3,4,5];
```

In case you want to store some objects in the array, it will be like the below code snippet. In this case the array is also known as Array of Objects.

```
var arr = [
    { name: 'Satya', age: 28 },
    { name: 'Shruti', age: 24 },
    { name: 'Swikruti', age: 23 }
];
```

JavaScript arrays are zero indexed. What does that mean and why is it so. Let's see in a while. Zero indexed simply means that the index of the first element of an JS array starts from 0 just like C and C++.There are other programming COBOL, Fortran, R, Julia etc. which do not follow this convention. This happens for JS because of the design of the programming language. When we say we want to access the first element of the array, by design, the first element resides at a place which is referenced by the array variable itself. So here the index is not pointing to the memory position of the element, but rather it points to the offset of the element from the initial memory location pointed to by the variable. Clear, Huh. Now let's explore some important properties and methods of array data structure in JavaScript.

length:
The length property gives the count of the items present in the array. It is a very important property as it is very helpful while iterating over an array. The index of the last item in the array is 1 less than the length.

```
var arr = [1, 2, 3, 4, 5];
console.log(arr.length); // 5
```

To get the last item we can do like;

```
arr[arr.length - 1] // 5
```

For iterating over the array, we can run a for loop through the array with the length property coming in as a condition.

```
        for (var i = 0; i < arr.length; i++) {

            console.log(arr[i]);

        }
```

push ():
push method is used to add a new array element at the end of the array. Pretty straightforward. Let’s see the syntax.

```
    var arr = [1,2,3,4,5];
    arr.push(6);
    console.log(arr); // [1, 2, 3, 4, 5, 6]
```

pop ():
pop method is used to remove the last item from an array. We can even empty an array with this method if we run a loop on the array for a count equal to the length of the array and pop each element from it. The pop() method returns the popped value.

```
        var arr = [1, 2, 3, 4, 5];
        var len = arr.length;
        for (var i = 0; i < len; i++) {
            arr.pop();
        }
```

shift ():

The shift () method is used to remove an element from the beginning of an array. It returns the removed element after the operation.

```
var arr = [1, 2, 3, 4, 5];
arr.shift();
```

unshift ():
The unshift method is used to add a new item at the beginning of the array. It returns the length of the final array.

```
var arr = [1, 2, 3, 4, 5];
arr.unshift(0);
```

delete ():
The delete method is used to delete an array entry as follows.

```
var arr = [1, 2, 3, 4, 5];
delete arr[2];
console.log(arr); // [1, 2, empty, 4, 5]
```

It returns a Boolean true or false depending upon if the deletion operation was successfully conducted or not. The deleted item shows as empty in the array and the length of the array remains unaffected. That happens because though there is no item present in the particular memory location, still it gets counted in the length of the array.

splice ():
The splice method removes array items and replaces new items if provided. The syntax of the splice() method is as follows.

Splice (starting index to remove item, number of items to remove, replacement items);

The third parameter is optional.

```
var arr = [1, 2, 3, 4, 5];
arr.splice(1, 2);
console.log(arr); //  [1, 4, 5]
```

With replacement items by providing the third parameter, we can remove the item first and replace it with new items.

```
var arr = [1, 2, 3, 4, 5];
arr.splice(1, 2, 9, 10);
console.log(arr); // [1, 9, 10, 4, 5]
```

If given a negative index, the splice operation is conducted from the end of the array.
This method returns a new array containing the removed items.

Slice ():
This method returns a new array containing the specified items except the last index provided.

```
arr = [1, 2, 3, 4, 5];
arr.slice(1, 3); // [2, 3]
```

concat ():
The concat method merges more than two arrays or values into a single array and returns the new array. The new array is serialized according to the original arrays.

```
var arr1 = [1, 2];
var arr2 = [3, 4];
arr1.concat(arr2); // [1,2,3,4]
```

The order of the concat operation matters and the sequence changes as demonstrated in the following code snippet.

```
var arr1 = [1, 2];
var arr2 = [3, 4];
arr2.concat(arr1); // [3,4,1,2]
```

The concat method also takes individual values as arguments like below.

```
var arr1 = [1, 2];
var arr2 = [3, 4];
arr1.concat(arr2, 5, 6); // [1,2,3,4,5,6]
```

forEach():

The forEach method is used to iterate over an array. It is native to the Array prototype. The advantages of forEach above for loop are

a) it has a clear syntax
b) the forEach method keeps the variables from the callback function bound to the local scope
c) the forEach is less error prone

An example of the forEach method is given below.

```
arr.forEach((item, index) => {
    console.log(`${item} is at index ${index}`);
});

// 1 is at index 0
// 2 is at index 1
// 3 is at index 2
// 4 is at index 3
// 5 is at index 4
```

indexOf ():
This method returns the position of the first occurrence of the item in the array. If the provided item is not part of the array, then it returns -1. Due to this feature, this method is used to check if an item is present in the array.
First let's check a very simple example.

```
var arr = [1, 2, 3, 4, 5];
arr.indexOf(3); //2
```

To check if an element is present in the array, we can do like

```
if (arr.indexOf(item) > -1) {
    // Code to be executed when item is present
}
```

lastIndexOf ():
The lastIndexOf method returns the last occurrence of the item in the array. It essentially searches from the end of the array.

```
var arr = [1, 2, 3, 4, 1, 6];
arr.lastIndexOf(1); // 4
```

includes ():
This method checks if an item is present in the array and returns a Boolean.

```
var arr = [1, 2, 3, 4, 5];
arr.includes(2); // true
```

find ():
The find method is used to find a particular item in an array. This method is very useful particularly when you are dealing with an array of objects. We pass a condition for this method and it selects the first element that matches the condition. If no item matches the condition, then it gives undefined.

```
var students = [
    { roll: 1, name: "John" },
    { roll: 2, name: "Pete" },
    { roll: 3, name: "Mary" }
];
var student = students.find(student => student.roll == 2);
console.log(student.name); // {roll: 2, name: "Pete"}
```

findIndex ():
This method returns the index of the found element instead of the entire object and if no item matches the condition, then it returns -1.

```
var students = [
    { roll: 1, name: "John" },
    { roll: 2, name: "Pete" },
    { roll: 3, name: "Mary" }
];
var student = students.find(student => student.roll == 2);
console.log(student.name); // 1
```

map ():
The map () function is used for array transformation. It takes another custom function as an argument and on each array item applies that function. The function provided as an argument can be a conventional vanillaJS function or it may be an arrow function.

```
var arr = [1, 2, 3, 4, 5];
arr.map(function (item) {
    console.log(item + 1);
});
```

Sort ():
The sort method as the name suggests is used to sort an array. A simple issue with the sort method is that it sorts the items by their corresponding string ASCII values thereby giving inaccurate results sometimes.

To overcome this problem, we pass a callback function as an argument which compares elements and gives the exact result. A simple use of the sort method is as follows.

```
arr = [3, 2, 3, 4, 2, 2, 7];
arr.sort(function (a, b) {
    return a - b;
}); // ascending order sorting
```

We can also do a descending order of sorting just by reversing the return value of the callback function. An example of a descending order sorting can be as follows.

```
arr.sort(function (a, b) {
    return b - a;
});// descending order sorting
```

Reverse ():
This method is used to reverse an array. It is a very straightforward method.

```
var arr = [1, 2, 3, 4, 5];
arr.reverse();
```

reduce ():

This method is used to reduce the array into a single value.

```
function sum(total, num) {
    return total + num;
}
var arr = [1, 2, 3, 4, 5];
numbers.reduce(sum); // 15
```

isArray ():
This method returns a Boolean stating if the provided variable is an array or not. If the value is an array it returns a true, else it returns false.

```
var arr = [1, 2, 3, 4, 5];
Array.isArray(arr); // true
Array.isArray({}); // false
```

These are pretty much the mostly needed array methods. Let’s move on to the next section which is about Object methods.

Most Important Object Methods

Objects in JavaScript are of prime importance as every literal in JavaScript has an object connection. As we know it, null is also an object. To understand objects better we can take a real life example where we are all surrounded by objects and each of the objects have different characteristics to differentiate between themselves. These characteristics are known as methods and properties. For example, a car object has 4 wheels which is its property and it can take some actions to move when we drive it, becoming a method. Since JavaScript as a language is completely object based, it is imperative for every JavaScript developer to know the inbuilt methods and properties of objects by heart. That is the objective of this section.

Till we are dealing with ES5, we Java Scripters don't have any concept of class, which only got released in ES6. So, in this part we will follow the POJO (Plain Old JavaScript Objects) wherein one object creates another object. We will explore the different possibilities of use cases of object methods and properties and see how we can use them better for our benefit.

For the uninitiated, we can create objects in different ways, out of which the below three are of prime importance.

i) By literals:

```
var obj = {};
```

ii)By constructor:

Here we are creating new objects through the Object constructor. This constructor can be any valid constructor function. We can directly call it or we can provide another object as a boilerplate for the newly created object.

```
var obj1 = new Object();
var obj2 = new Object({ name: 'john', age: 28 });
```

Alternatively, we can also create specific objects of a particular type as follows.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

var john = new Person('john', 28);
```

iii) By the Object.create method:

```
var obj = Object.create({});
```

Now let's explore some of the most important object methods available in the JavaScript language.

Assign ():
The Object.assign method copies the properties from one object to another object. This is very handy when you want to copy all the properties of one object to another object. For this method the target object is passed as the first argument and the source object is passed as the second parameter.

```
var obj1 = { fName: 'foo', lName: '' };
var obj2 = { lName: 'bar', age: 32 };
var target = Object.assign(obj1, obj2);
console.log(obj1) //{fName: "foo", lName: "bar", age: 32}
```

Here we can observe that while copying, the common properties in the target object get overridden.

create():

The Object.create method as we saw in the object creation methods, helps to create an object of the provided object type.

```
var obj = Object.create({ name: 'john', age: 28 });
console.log(obj);
```

This will create a new object called obj with the name and age properties in the prototype.

DefineProperty ():

The Object.defineProperty method is used to create a new property in an existing object.

```
var obj = {};
Object.defineProperty(obj, "name", {
    value: 'JOHN'
});
console.log(obj); // {name: 'JOHN'}
```

defineProperties ():

The Object.defineProperties method creates new properties within an object and it can create multiple properties at one go.

```
var obj = {};
Object.defineProperties(obj,
    {
        name: {
            value: 'Satya'
        },
        age: { value: 23 }

    });
console.log(obj); //{name: "Satya", age: 23}
```

entries ():

The Object. Entries method returns the properties of an object in the same order as they appear if we run a loop through the object. This method does not bring the properties from the prototype of the object.

```
var obj = { name: 'satya', age: 20 };
Object.entries(obj); //   ["name", "satya"] ["age", 20]
```

freeze():

The Object.freeze method freezes an object and does not let any changes happen to the object. Once an object is frozen, it does not even let the property values to be changed. If the operation is happening under strict mode, then an attempt to alter the object will give rise to an error. So, be careful of that.

```
var obj = { name: 'John' };
Object.freeze(obj);
obj.name = 'Doe'; // try to change the object
console.log(obj); // {name: 'John'}
```

getOwnPropertyDescriptor ():

The Object.getOwnPropertyDescriptor method returns the description of a particular property from a given object.

```
var obj = { name: 'JOHN' };
Object.getOwnPropertyDescriptor(obj, 'name');
// {value: "JOHN", writable: true, enumerable: true, configurable: true}
```

getOwnPropertyNames ():

The Object.getOwnPropertyNames method returns an array containing all the property names (both enumerable and non-enumerable) belonging to the object.

```
var obj = { name: 'John', age: 28 };
Object.getOwnPropertyNames(obj); // ["name", "age"]
```

getPrototypeOf ():

The Object.getPrototypeOf method returns the prototype of an object. This method is very useful for asserting the lineage of an object. A simple example demonstrating the use of this method is given below.

```
        var obj = { name: 'John', age: 28 };
        Object.getPrototypeOf(obj); // returns the prototype of the
provided object
```

Is ():

The Object.is method is used to compare two values and it returns a Boolean expressing if the provided values are the same or not. If they are the same, it returns true, else it returns a false. This method is different from the == operator as it does not apply type coercion for the operands. And it is different from the === operator as well since it returns true for the expression NaN === NaN. A simple demonstration of the is () method is given below.

```
        Object.is(NaN, NaN); // true
```

preventExtensions ():

The Object.preventExtensions method is used to prevent addition of any new properties to an existing object. Once this method is applied on an object, then we can't add new properties to it. In the following example we create a new object with just one property called name. Then we apply the preventExtensions () method to the object and then try to add a new property called age. Next, we console the object and see that the age property is not added to the object.

```
        var obj = { name: 'JOHN' };
        Object.preventExtensions(obj);
        obj.age = 28;
        console.log(obj); // {name: 'JOHN'}
```

isExtensible ():

The Object.isExtensible method is used to check if there is a possibility to add new properties to an existing object. If the method returns true, then new properties can be added to the given object else no new properties can be added to it. A small demonstration of the method is given below.

```
var obj = { name: 'JOHN' };
Object.preventExtensions(obj);
Object.isExtensible(obj); // false
```

isFrozen ():

The Object.isFrozen method determines if an object is frozen or not. It returns a Boolean. For empty objects upon which the preventExtensions method is applied, the isFrozen () method returns true. This object is known as vacuously frozen if that object has at least one property and the preventExtensions() method is applied upon it. The isFrozen () method returns false as even though new properties can't be added to it, still the existing properties are writable and configurable.

Some sample code to demonstrate the usage of the isFrozen() method is given below.

```
var obj = { name: 'JOHN' };
Object.freeze(obj);
console.log(Object.isFrozen(obj)); // true

var obj = {};
Object.preventExtensions(obj);
console.log(Object.isFrozen(obj)); // true
```

isSealed ():

The Object.isSealed method returns a Boolean indicating whether an object is sealed or not.

```
var obj = {
    name: 'JOHN'
};
console.log(Object.isSealed(obj)); //false
Object.seal(obj);
console.log(Object.isSealed(obj)); // true
```

keys ():

The Object.keys () method returns the list of keys available in an object. The return value is of type array.

```
var obj = {
    name: 'JOHN',
    age: 21
};

Object.keys(obj); // ["name", "age"]
```

Seal ():

The Object.seal () method seals an object. In this context the word seal essentially means the following points.

- no new properties can be added to the object
- making all existing properties as non-configurable
- existing properties can be overridden, if they are not sealed already

```
var obj = {
    name: 'JOHN',
    age: 21
};

Object.seal(obj);
obj.address = 'India'; // try to add a new property to the object
console.log(obj); // {name: "JOHN", age: 21}
delete obj.name; // try to delete a property from the object
console.log(obj); // {name: "JOHN", age: 21}
```

values ():

The Object. Values method returns an array containing the property values of the object.

```
var obj = {
    name: 'JOHN',
    age: 21,
    foo: function () {
        console.log('Hello World');
    }
};
console.log(Object.values(obj)); // ['JOHN', 21, function]
```

__defineGetter__ ():

This method sets a getter function on an object property such that when that property is looked up to, it calls a getter function.

For demonstrating this method, let's create a new object with the Object.create method with a very simple prototype. Then we create a new property called details with the __defineGetter__ method and attach a function to this property. Next, we try to access the details property such that the getter function returns a string.

```
var person = { name: 'JOHN' };
var obj = Object.create(person);
obj.__defineGetter__('details', function () {
    return `My name is ${this.name}`;
});
obj.details; // "My name is JOHN"
```

constructor:

The constructor property returns a reference to the object constructor function that created the given object.

```
var obj = { name: 'JOHN' };
obj.constructor; //ƒ Object() { [native code] }
```

hasOwnProperty ():

The hasOwnProperty method returns a Boolean indicating whether the given object contains a particular property of its own. It does not take into account
the properties inherited by the object.

```
var obj = { name: 'JOHN', age: 28 };
obj.hasOwnProperty('name'); // true
```

isPrototypeOf ():

This method helps to ascertain if an object exists in the prototype chain of another object. We can see the usage with a simple example.

```
        var person = { name: 'JOHN' };
        var obj = Object.create(person);
        person.isPrototypeOf(obj); // true
        Object.prototype.isPrototypeOf(person); // true since Object is
the parent class of all objects
        Object.prototype.isPrototypeOf(obj); // true
```

propertyIsEnumerable ():

This method returns a Boolean indicating if a particular property is enumerable or not.

```
        var obj = { name: 'JOHN', age: 28 };
        obj.propertyIsEnumerable('name'); // true
```

toLocaleString ():

The toLocaleString method returns the string representation of an object which is demonstrated below.

```
        var obj = { name: 'JOHN' };
        obj.toLocaleString(); // "[object object]"

        var d = new Date();
        d.toLocaleString() // converts the date to a string representation
```

toString ():

The toString method converts an object to a string value.

```
var obj = { name: 'john', age: 28 };
console.log(obj.toString()); // [object Object]
```

The signature of the returned value of this method [object, type] where type represents the type of the object.

valueOf ():

The valueOf method returns the primitive value of an object. JavaScript inherently applies this method on all objects to get their value.

```
var obj = { name: 'JOHN' };
obj.valueOf(); // {name:  'JOHN'}
```

Destructuring

Real time objects in JavaScript can be very complex and accessing the keys of those objects are very cumbersome. Imagine a situation where we have a very complex object which is coming from the backend by virtue of a successful API call. A fragment of the object may be something like this.

```
myObj = {
    userDetails: {
        businessDetails: {
            name: 'XYZ Company',
            address: {
                line1: 'First Line',
                line2: 'Second Line',
                state: 'California',
                City: 'XYZ City',
                zip: ''
            }
        },
        personalDetails: {
            name: 'John Doe',
            address: {
                line1: 'First Line',
                line2: 'Second Line',
                state: 'California',
                City: 'XYZ City',
                zip: ''
            },
            phone: {
                countryCode: '+1',
                number: '999999999',
                extn: '1234'
            }
        }
    }
};
```

Now you have a form where you need to auto populate these values. Let's take the example of binding the fields from the address key inside the business Details object. So, for each form field, using vanillaJS you would do something like this.

myObj.userDetails.businessDetails.address.line1 and *myObj.userDetails.businessDetails.address.line2* and so on. By now you must have got the idea that for each value you have to do the same traversal in the object tree, which is very cumbersome. Isn't it.

No worries anymore. The ECMAScript team has come up with a brilliant solution for this problem. With ES2015, they have introduced a concept called Destructuring which helps to unpack complex entities like the above objects or arrays into separate variables. Let's see the different possibilities with destructuring.

Destructuring with Arrays:

- ❖ Simple assignment through array

```
var arr = [1, 2, 3, 4, 5];
var [a, b, c, d, e] = arr;
console.log(a, b, c, d, e); // 1,2,3,4,5
```

- ❖ Extracting simple values

```
var arr = [1, 2, 3, 4, 5];
const [p, q] = arr;
console.log(p, q); // 1, 2
```

- Ignoring some specific values

```
var arr = [1, 2, 3];
[x, , z] = arr;
console.log(x, z); // 1 3
```

- Destructuring multiple values through a spread operator

```
var arr = [1, 2, 3, 4, 5];
var [a, b, ...others] = arr;
console.log(others); // [3,4,5]
```

- Setting default value if the corresponding object in array undefined

```
var arr = ['Satya'];
let [name = 'hello', age = 27] = arr;
console.log(name, age); // Satya 27
```

- Parsing return value of function

```
function foo() {
    let arr = [1, 2, 3, 4, 5];
    return arr;
}
[a, ...rest] = foo();
console.log(a, rest); // 1, [2,3,4,5]
```

Destructuring with Objects:

- Simple unpacking with same variable name

```
var obj = { a: 1, b: 2 };
var { a, b } = obj;
console.log(a, b); // 1 2
```

- Unpacking with different variable name

```
var obj = { x: 1, y: 2 };
var { x: foo, y: bar } = obj;
console.log(foo, bar); // 1 2
```

- Assigning default values to properties

```
var obj = { x: 10 };
var { x = 5, y = 10 } = obj;
console.log(x, y); // 10 10
```

- Providing default parameters to a function

```
function userDetails({ name = 'Satya', age = 20 }) {
    console.log(name, age);
}
var user1 = { name: 'John' };
var user2 = { name: 'Doe', age: 28 };
```

```
        userDetails(user1); // John 20,since there was no age param, it
took the default value
        userDetails(user2); // Doe 28
```

❖ Multi-level destructuring

```
        var obj = {
            fullname: 'John Doe',
            age: 28,
            addresses: [{
                type: 'Primary',
                landmark: 'Bangalore',
                houseNo: 15
            }]
        };
        var {
            fullname: fullName,
            addresses: [{
                type: addressType
            }]
        } = obj;
 console.log(`My name is ${fullName} and address is ${addressType}`);
 // My name is John Doe and address is Primary
```

❖ Rest operator in object destructuring

```
var obj = { a: 1, b: 2, c: 3, d: 4, e: 5 };
var { a, b, ...rest } = obj;
console.log(rest); // {c: 3, d: 4, e: 5}

//destructuring in array of objects
var obj = [{
    name: 'James',
    age: 24
},
{
    name: 'John',
    age: 26
},
{
    name: 'Jean',
    age: 27
}];
var [firstObject, secondObject, thirdObject] = obj;
console.log(secondObject); // {name: 'John', age: 26}
```

This is pretty much the basic possibilities with destructuring.

Prototypes in JavaScript

Once upon a time, there was a small cute cat who had just started seeing the world. She used to wander around the lands with her mother. Then one fine day she asked "Mom, why do I meow". The mom was a JavaScript expert. Surprised, she told "YOU ARE MY ROTOTYPE". All properties you have and actions you take are coming to you by virtue of you being my prototype. The cute little cat, astonished by the answer asked again, "Mamma what is this prototype all about?", to which the mother said just go through this section and you will get to know whatever you ought to know. End of the story. Let’s get back to our developer gears.

The entire JavaScript ecosystem is built on one concept i.e. the Prototype. You may be an expert or a naive in JavaScript, but knowingly or unknowingly you have to cross your path with the prototype concept. It is everywhere. The arrays we store our items or the objects or even the functions, all are driven by prototypes. So, it is very imperative for any java scripter to get hold of this concept and take advantages of.

The bare minimum introduction to the prototype concept is that whenever we create a new object, or a new array and we console it, we see the items of the entity along with a __proto__ object. This happens thanks to prototypes. At a minimum we need to know that the __proto__ object associated with the array or object points to the prototype of the function which created this entity under the hood, in our case the __proto__ from array points to Array () and that of object points to the Object () constructor function's prototype.

Since we are developers, we just need to get hold of the prototype use cases so that we can use these things to our advantage in our next program. On a very basic level, prototype does two things for us. It helps our entities (arrays, objects, functions) to inherit from its parent and secondly it helps us to add new methods and properties to the constructor functions. Both of these things we will explore in this section.

Inheritance through prototype:

Have you ever done anything similar to this?

```
var arr = [1, 2, 3, 4, 5];
arr.push(6);
```

Just wait for a second and think. While declaring the array did you write the push () method to it. No, right. Then who gave it to the array?
The answer is simple. Any array we declare, under the hood it is being created from the Array () constructor. In this process it gets the privilege of inheriting the properties and methods from the parent constructor.

If you print the above array into the console, you can see the __proto__ object and you can easily find the push () method in that object. Now type Array. prototype onto the console window and traverse through that output object. Here also you can easily find the push () method.
So, in this entire process what happened is when you tried to invoke the push () method on the array, the compiler searched for this method in the object. Since we did not provide the method to the object directly, it looked for it in the __proto__ object. It found the method there. Had it not found the method in the __proto__ object, it would have gone through the object hierarchy and looked up for the method in the parent Array () constructor prototype.

This mechanism by which the children are linked to their parent constructor functions in the object tree is known as Prototype chaining.

Adding new methods and properties to a constructor function

As we know till ES5, we don't have the concept of classes (in depth explanation of classes is given in another section) and everything is an object in JavaScript. So, it is very evident that objects create objects in JavaScript. These objects are essentially constructor functions. Before diving deeper into that let's see a very simple constructor function in action.

```
function Person(name) {
    this.name = name;
    foo: function foo() {
        console.log(this.name);
    }
}
var john = new Person('John');
var peter = new Person('Peter');
```

Pretty simple right. But when we console the above two objects john and peter we can see a problem. Both these objects contain the foo () method even though they might not require it. This is not what we require. This is a grey spot on the memory. To fix this problem, we will add a new method called bar to this constructor function and see if there are any changes after that.

```
        Person.prototype.bar = function () {
            console.log('I am coming from the prototypically added
method!!!');
        }
```

Now when we create an instance of the Person object, the instance will not contain the bar () method in its definition by default. But, still we can access the bar () method from the new instance. This is the power of prototype.

These are the two main applications of the prototype design pattern in JavaScript.

Hoisting in JavaScript

Hoisting in English language means to pull something up. You must have been to the National Day celebration event, aren't you? On that occasion, the chief guest approaches the mast and he pulls some rope and the flag goes up, right. This is exactly what happens in JavaScript albeit with some change. In JavaScript the functions and variables are hoisted to the top of the scope. Let's see some use cases for that to understand the hoisting concept better. But before that let's gather some common knowledge about the process.

First thing first. Just keep it in mind that whenever the browser engine is reading your code and giving you the output, mostly it is a two-phase process.

1. Hoisting phase
2. Execution phase

In the hoisting phase, the browser takes all your available functions and variables, takes them up to the highest position in the available scope depending upon whether they are declared locally or globally and create space for them in the memory. Wait a moment. Declaration!!! Is it not the same as an assignment?
No, these two stages are different. JavaScript gives us the flexibility to declare and assign variables with a single statement, but the interpreter treats the statement differently.

Suppose you write like this.

```
var myVar = 10;
```

The interpreter will treat the above code in two statements like this.

```
var myVar; // declaration stage
myVar = 10; // assignment stage
```

Now that we know the overall concept of hoisting let's apply it in the JavaScript context and see what are the various scenarios where we need to be cautious while writing code.

The difference between "undefined" and "Is not defined"

These two phrases seem similar, but in JavaScript context they are completely different. While undefined is a value, 'is not defined' is a type of reference error. Let's see how.

```
console.log(var1);
var var1 = 10;
```

The above code prints undefined in the console because the variable is assigned later than the console.log statement. Essentially the interpreter follows the following steps to execute the above code.

Like we have already discussed, the declaration phase comes before the assignment phase. So, the above code looks like the below to the interpreter.

```
var var1;
console.log(var1);
var1 = 10;
```

Here what is happening is, on step1, the interpreter allocates a memory space for the variable var1. Now that the variable is not yet assigned any value, it assigns a value undefined to the memory location. So now the memory location described by var1 contains undefined. On the next line when we try to access the memory location by referring to var1, it fetches the value from the memory location and till now it contains undefined. So, the console statement prints undefined.
On the next step, the variable is assigned a value 10 and now the memory location described by var1 contains the value 10. That is the reason why, when we try to access a variable before it is assigned, gives us an undefined.

Next, we do something like this.

```
console.log(var2);
```

Here in this case the browser throws an error saying Uncaught Reference Error: var2 is not defined. Here the error clearly says that it is a reference error because since the variable is not declared, there is no memory location allocated to the var2 variable and it is not able to reference it. That is why the error comes up.

The "type of" ambiguity

The type of operator returns the type of a variable such that type of (1) returns "number" and type of (null) returns "object". Now in the context of this article, if we write the below code onto the console, the type of operator returns "undefined".

```
        console.log(typeof var1); //undefined (when variable is not 
declared or assigned)
        console.log(typeof var2); // undefined (when variable is assigned 
later)
        var var2 = 10;
        var newVar = 10;
        console.log(typeof newVar); // number (when variable is assigned 
before typeof assertion)
```

This leads us to the conclusion that in JavaScript, a variable is assigned a value of undefined if it is not assigned to any other value except undefined before referring to it.

If it is not var, then it is global

This is a very typical use case of hoisting. Let's see that by an example.

```
        foo();
        function foo() {
            var1 = 10;
            var var2 = 20;
        }
        console.log(var1); // 10
        console.log(var2); // Uncaught ReferenceError: var2 is not defined
```

What happened here. For the first variable declared inside the function, we have omitted the var keyword. So as soon as the function is called, the variable is assigned to the global namespace and hoisted there. That is why when we try to access the variable outside of the function it is becoming accessible.
Had we put the var keyword before that variable, it would have been a local variable to the function body and would be inaccessible outside of the function, which is the case for the var2 variable.
Wait a second. If you have not observed yet, here in this example we are calling the function even before its declaration. Let's save this concept for one of
the upcoming pointers about function hoisting.

let & const: the ES6 big boyz

Till ES6 was released there were only two types of scopes available in JavaScript, one being the global scope and the other one the local scope. But, with the release of ECMAScript2015, the dynamics got changed about the declarations of variables. It introduced two new keywords namely let and const. While the former is used to define a block scoped variable the latter came to be used for defining immutable values. Here we will see how the hoisting of the values change with the use of let and const keywords. Now let's see some of the possible use cases and their corresponding outputs with let and const.

```
console.log(var1); // Uncaught Reference error var1 is not defined
let var1 = 10;

let var2;
console.log(var2); // undefined
var2 = 10;

console.log(var1); // Cannot access 'var1' before initialization
const var1 = 10;
```

★ A constant declaration should always accompany an initialization.

function hoisting with declaration

The interpreter always takes the functions to the top of the execution context and that is why we can access the functions even before declaring them like below.

```
foo();
function foo() {
    alert('I am being called!!!');
}
```

function hoisting with function expressions

When we write a function as an expression, the interpreter treats it as a normal variable and hoists it to the top of the scope. Thereby when we try to call the function through an expression even before it is declared, we get an undefined value.

```
foo();
var foo = function () {
    alert('');
}
```

The above code returns undefined for obvious reasons.

class hoisting with declaration

A ES6 class is not hoisted when it is declared. We can’t access a class before it is declared. An example in this context is given below.

```
// instantiating  a class before it is declared
var john = new Person('john', 28);

class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

The above code throws an error saying Uncaught Reference Error: Person is not defined which proves that class declarations are not hoisted to the top of the scope and we can't access a class before it is declared. Let's see what happens when we write a class expression.

class hoisting with expression

When a class is written in the form of an expression, the interpreter treats it as a normally hoisted variable. Let's see this thing in the form of two examples.

```
console.log(Person1); // undefined
var Person1 = class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

The above code gives undefined as the variable is hoisted to the top of the scope before it is initialized with a class expression.

```
var john = new Person('john', 28);
console.log(john);//Person {name: "john", age: 28}
var Person = class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

In the above example even though the class expression is assigned later, we can still make a new object from the class.

Order of precedence for functions and variables
This order of precedence defines which value the execution context takes up in case of functions and variables declaration and assignment. It is governed by two simple rules.

- ❖ Variable assignment is prioritized than function declaration
- ❖ Function declaration is prioritized than variable declaration

We can understand the above two rules through two simple examples as given below.

Variable assignment is prioritized than function declaration

In this example we create a function and a variable with the same name and assign some value to the variable. Then we try to access the variable first as a reference and then as a function. We can see that the variable assignment has taken precedence over the function declaration.

```
var name = 'john';
function name() {
    alert('my name is john');
}
console.log(name); // john
name(); // Uncaught TypeError: name is not a function
```

Function declaration is prioritized than variable declaration

In this example we take a function and a variable with the same name, but we don't assign any value to the variable. Then we try to access both of these and see what happens.

```
var name;
function name() {
    alert('my name is john');
}
console.log(name); // gives the body of the function
```

This is pretty much it about the concept of hoisting in JavaScript.

Closures in JavaScript

A closure is an environment in which an inner function has access to the methods and properties of the outer enclosing function such that the inner function can boast of availability of three scopes viz its own local scope, the global scope and the scope of its parent function which encloses it.
A typical real-life example of a closure can make this concept better understood. Take the example of your father's car. You can drive it by virtue of you being the offspring of your parents. But the car is not available for anybody else who does not belong to your family. The rest of the world can't access it, right. This example is similar to the concept of closures where an inner function (the offspring) has access to the methods and properties of the parent function (the parent).

In technical terms as mentioned in the MDN documentation, A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). By this definition we can clearly decipher that there are multiple functions involved and they create a lexical environment or state around them. A closure is created anytime a function is created.

The next jargon here is Lexical Scoping. Let's try to understand that. Lexical Scoping is the mechanism which determines how variable names are resolved by the parser when there are nested functions. It determines whether a variable is present in the current context/scope.
Now that we have got some idea about the concept, let's see that in action. A very simple example of a closure can be as follows.

```
function foo() {
    var name = 'john';
    function bar() {
        alert(name);
    }
    bar();
}
foo();
```

Here we have an outer function foo () and an inner function bar () and the outer function has a variable called name which we are trying to access inside the bar () function. Inside the parent function itself we are calling the inner function. Inside the bar() function we just have an alert statement which alerts the name.
Pretty simple right. Now let's do the same thing but with a slight variation where instead of calling the inner function inside the body of the outer function, we return the inner function and see the implications.

```
function foo() {
    var name = 'john';
    function bar() {
        console.log(name);
    }
    return bar;
}
var functionReference = foo();
console.log(functionReference); // Returns body of bar()
functionReference(); // john
```

Here we are returning the inner function bar () and storing it in a variable called function Reference. The console.log statement after that prints the body of the inner function as expected. When we invoke the function Reference variable again by means of a function call like function Reference () then we can see the alert message.

But wait. There is a small issue we need to understand here. We know any local variable lives only till the execution of the container function. This is a JavaScript phenomenon. Then when we call the function Reference (), we get access to the name variable. Has not the function foo () completed its execution. If it has completed the execution there should not be any name variable available in the memory. To solve this anomaly, there comes the concept of closure. In a typical closure, the inner function has access to the outer function variables, right. Now in our case even if the outer function foo () has completed execution, the interpreter keeps a reference of the variables inside it to use them later.

If we do;

```
console.dir(functionReference);
```

and go deeper into the output object, inside the scope [0] we get the reference of the name variable. The interpreter essentially picks the value of the variable from here.
So, it is not always a good practice to use closures as the garbage collection mechanism takes a hit by this approach. So, unless it is required please don't use closures.

Next, we will see a special case of closures called Currying.

Currying

Currying is a process in which we transform a function of multiple arguments into nested functions so that the inner functions are always returned until a final value is returned from the function block. Let's see a simple example of function currying. The Currying is a special application of closures. So, when you are facing your next JavaScript interview, don't forget to revise this currying concept.

```
function multiply(a) {
    return function (b) {
        return a * b;
    }
}
multiply(5)(5); // 25
```

Practical use of closures

There are very few ways in JavaScript by which we can make variables private. The closure method is one of them. Let's see how we can make a variable/function private with the help of closures.

```
function foo() {
    function privateFunction() {
        console.log('hello');
    }
    return {
        publicFunction: function () {
            privateFunction();
        }
    }
};
var x = foo();
x.publicFunction();
```

But if we try to call the private Function directly through x, it will throw an error saying Uncaught Type Error: x. private Function is not a function.

This is all the basics stuff about closures. There can be many possibilities with closures, but this was a starting point.

Garbage Collection Mechanism in JavaScript

We know what garbage is. Garbage essentially refers to things which are no longer of use. When it comes to programming, Garbage Collection means cleaning the memory spaces which don't contain useful data and then reallocating those cleared memory to some other data which is both active and useful. That is the basic process of Garbage Collection in pretty much all the programming languages. Some programming languages needs explicit interference from the developer while some other languages do this automatically. A low level programming language like C, requires the developer to free the memory by the use of methods such as malloc () and free () when the program no longer needs those variables or objects. It is a developer prerogative to free the memory and the ball is in the developer's court to decide explicitly whether to free the memory or not. But this is not always the case. For a high level programming language like JavaScript, the memory management process is automated. The browser takes care of that thing for us.

In this section we will see the following contents.

- ❖ The memory management life cycle
- ❖ The Garbage Collection algorithms
- ❖ Pitfalls for memory leakage

Let us now see the general life cycle of memory management to better understand the process.

- Whenever a variable, object or function is created, a memory space is allocated to it.
- In the next stage, the allocated memory is used by means of Read/Write operations.
- When the memory is no longer needed, release the memory space.

The last step is called the Garbage Collection mechanism. This is an automated process in JavaScript and the process is done by an entity called Garbage Collector, though we can't physically find this agent in the browser engine. The purpose of a Garbage Collector is to monitor memory allocation and determine if a specific memory is needed any longer. If it is not needed, then it reclaims the memory. But the Garbage Collection mechanism is always an approximation as the usefulness of a specific block of allocated memory is undecidable.

The Garbage Collection mechanism in JavaScript is governed by two algorithms which are listed below.

1) Reference Counting Algorithm
2) Mark and Sweep Algorithm

Reference Counting Algorithm

First of all, let's try to understand what is a reference. Given two objects, one object is said to have a reference to another object if the first object has access to the methods or properties of the second object. Now, let's move on to the algorithm.

The Reference count algorithm is very simple, and it determines the usefulness of an object by finding out if the object is being referenced by some other object or not. If the object is not referenced by any other object, then it is taken as a garbage value and collected. A simple example of this can be as follows.

```
var obj = { // first object created
    property1: { // second object created
        property2: 10
    }
}
```

Here two objects are created, and one is referenced by the property of the other object. The other object being referenced by being assigned to the obj variable. Since there are references, there is no scope of garbage collection.

```
// Now we have another to the existing objects.
var newObj = obj;
/**
* Still the objects are referenced by the newObj variable.
* So there is no chance of GC
*/
obj = 10;
```

```
var anotherObj = newObj.property1;
```

Now the object property1 has two references. One as the property of the new Obj variable and another from the another Obj variable. So, it can't be garbage collected.

```
newObj = 'Some string';
```

Now the object created under obj variable has zero reference as such. But still property1 is being referenced by the anotherObj variable. So, it can't be garbage collected.

```
anotherObj = null;
```

Now we don't have any reference to the property1 object from anywhere. Under this situation, the objects can be garbage collected.

The Reference Counting algorithm is very naive in nature and it can't be completely relied upon. For an example like below, the Reference Counting algorithm does very less.

```
function foo() {
    var obj1 = {};
    var obj2 = {};
    obj1.a = obj2;
    obj2.a = obj1;
    console.log(obj1);
    console.log(obj2);
}
foo();
```

Just traverse through the output objects. Can you see the nested properties? This is something known as a Circular Reference. In this case the Reference Counting algorithm fails miserably as all the nested objects are referenced by each other. So, there is no possibility of garbage collection using this algorithm. This leads us to our next algorithm, i.e. the Mark and Sweep Algorithm.

Mark and Sweep Algorithm

The Mark and Sweep algorithm implements the garbage collection mechanism by means of reaching out to objects. If the object is not reachable, then it is taken as a garbage and collected. The algorithm follows the principle that if an object is having zero references then it is effectively unreachable. So, it is fit to be garbage.
The process identifies the root object i.e. the window object and from there on it traverses to all other child objects and then to the children of the child objects. If there are some objects which can't be reached in this process, they are collected as garbage and memory is freed. This algorithm effectively solves the cycle reference problem which we saw in the Reference Counting algorithm.

Some of the common pitfalls for memory leakage

A developer can do a lot to prevent memory leakage in his application. Some of the very common causes of memory leakage in the application is given below.

- Excessive use of global by creating global variables or by omitting the var keyword in local scope
- Forgetting the clearing of timers like setInterval ()
- Unnecessary use of closures (closures always keep a reference of the variables from the parent function even if the parent function has completed execution).

So, we should always avoid making the above mistakes.

call, bind and apply methods

Object Oriented JavaScript is complex. Particularly the "this" value. Many times, even the expert developers fall into the trap of "this" ambiguity. We know a function in JavaScript has many parts like its name (sometimes with no name for anonymous functions), a body which defines what the function does and most importantly the "this". Since this is a vast concept, I have kept it aside for another section. For this section we just need to know that in a general context the "this" value of a function refers to the object calling the function. Now, let's dive into the code of bind, call and apply methods.

the bind () method:

The bind function creates a new function that when called calls the function in the provided context or this value. To understand this definition of the bind function let's see a very common problem which we developers face.

```
        var myVar = 10;
        var obj = {
            myVar: 20,
            getVar: function () {
                console.log(this.myVar);
            }
        };

        obj.getVar() // 20 (Local Access)
        var globalAccess = obj.getVar;
        globalAccess() // 10 (Here the this value is set to window object
by default)
```

As we can see from the above example, once we keep a reference of the inner function in a variable and try to access it later, then the "this" value is automatically getting set as the global object and it takes the property myVar from the global namespace. This is not what we want, right. We need to set the context as per our requirement and with an object of our choosing. The bind () function gives us this flexibility where we can set the "this" value manually by providing the desired object (this value) while keeping a bounded reference of the function. The above example when used with the bind () function can be rewritten as follows.

```
var myVar = 10;
var otherObj = {
    myVar: 30
};

var obj = {
    myVar: 20,
    getVar: function () {
        console.log(this.myVar);
    }
};

var bindWithObj = obj.getVar.bind(obj); //bind the obj
bindWithObj(); // 20

var bindWithOtherObj = obj.getVar.bind(otherObj);
bindWithOtherObj(); // 30
```

Another very simple example with the bind () method can be given as follows.

```
var obj = { name: 'John' };
function foo(age) {
  console.log(`${this.name || 'Anonymous'} is ${age} years old.`);
}
//"this" is undefined. So it takes  default value
foo(25) // Anonymous is 25 years old
var john = foo.bind(obj); // bind the obj object
//Here "this" value is bound to the obj object
john(30) // John is 30 years old
```

Practical uses of bind () function

- ❖ An often-used bind () is as shown in the above example where we bind a function to a particular object
- ❖ Since setTimeout () always takes the window object as "this", sometimes bind is used to bind the actual context with this method
- ❖ Bind () is used with partially applied functions
- ❖ Bind () method is used as constructor

the call () method

The call () method calls a function with a given "this" value and arguments provided individually. The call () method invokes the functions with a provided "this" and it invokes the function immediately unlike bind () method.
In simple terms if you want to call a function with a given "this" value, then you need to use the call () method.

Let's see the implementation of the call () method with a simple example.

```
        var obj = { name: 'john' };
        function foo(age, city) {
         console.log(`${this.name} is ${age} years old and resides in
${city}`);
        }
        foo.call(obj, 30, 'NewYork'); // john is 30 years old and resides
in NewYork
```

From the above example we can see that the obj got bound to the "this" value of the foo () method thanks to the call () method.

If the first argument in the call () method is not provided, by default it takes the "this" context from the global/window object as can be seen in the
following example.

```
var name = 'john';
function foo() {
    var name = 'peter';
    console.log(`Hi I am ${this.name}`); // Hi I am john
}
foo.call();
```

In the above example the ‘this’ value refers to the Window object and that is the reason it takes the name value from the global object instead of the local variable with the same name.

Practical use of the call () method

- used to invoke a function with a given “this” value
- call () is used to chain constructors
- it is used to invoke an anonymous function by supplying the correct "this" value.

the apply () method

The apply () method is used to call a function similar to the call () method and it also binds the "this" value to the function. The only difference between call () and apply () is that apply () takes the rest of the arguments as an array. Let's see it in action.

```
var obj = { name: 'john' };
function foo(age, city) {
    console.log(`${this.name} is ${age} years old  from ${city}`);
    // john is 30 years old  from NewYork
}
foo.apply(obj, [30, 'NewYork']);
```

This is pretty much it for the basics of call, bind and apply methods. Please keep it in mind that this concept is very important from your interview point of view.

Class concept in JavaScript

Little John is just going to start his fourth standard this academic year. He heard it somewhere that the population of the world is around 8 billion. He knew God created the universe and the human beings. But this enormous number struck his little mind and days in and days out he kept on thinking. How did God get to create so many human beings? It was a Sunday afternoon and he finally asked his mom, "Mom, how come God created so many human beings. Does not He feels tired creating one human after another. How does He has got so much time ???". The mother was a very reasonable lady and she said, God must have created a frame first. Then He could have created two more frames. One for the boys and one for the girls. Then using these two frames He just has to produce any number of human beings as he wanted. Simple was the explanation and little John nodded his head in agreement.

We are developers, and we don't miss any chance of expressing our thoughts through code. So how can we miss this grand architecture suggested by John's mother. Let's try our hand to emulate the above structure proposed by her.

To emulate the above structure, we can take two approaches. First approach is through constructor functions which we java scripters have been doing for decades. The typical code will be to create a constructor function called Humans. Then we have to create two more constructor functions called Boy and Girl. Inside each of these functions somehow, we need to use the call () method to inherit the prototype from Humans function. Somehow it will look like this.

```
function Humans(options) {
    this.options = options;
}
function Boy(otherOptions) {
    Human.call(this, otherOptions);
}
Boy.prototype = Object.create(Human.prototype, {
    constructor: {
        value: Boy
    }
});
```

★ The above code is not a working model, but a vague emulation just for the sake of explanation.

This can work. But why not try out with the newly added class syntax which was introduced in ES2015 as a syntactic sugar over the prototype based inheritance model in JavaScript. Let's try that to solve our problem.

```
class Humans {
    constructor(hands, legs) {
        this.hands = hands;
        this.legs = legs;
    }

}

class Boy extends Humans {
    constructor(name, isHeHandsome) {
        super(2, 2);
        this.name = name;
        this.isHeHandsome = isHeHandsome;
    }
}

class Girl extends Humans {
    constructor(name, isSheBeautiful) {
        super(2, 2);
        this.name = name;
        this.isSheBeautiful = isSheBeautiful;
    }

}
```

Now that our frames/classes/blueprints/cookie-cutter is in place we can create one boy called john and a girl called Sara. Let's see how that happens.

```
        var john = new Boy('John', true);
        var sara = new Girl('Sara', true);

        console.log(john); // Boy {hands: 2, legs: 2, name: "John",
isHeHandsome: true}
        console.log(sara); // Girl {hands: 2, legs: 2, name: "Sara",
isSheBeautiful: true}
```

Finally, we created a boy and a girl. We did it. But wait. There are so many things going on in the above code. Let's not be overwhelmed by the code. We will explore all the concepts one by one. Let's do it one by one.

- What is the hack of the word class?
  Well, it is the keyword you need to use if you want to declare a class.

- What is a constructor function inside the class block?

  A constructor function is the first function that gets called when a class is instantiated (when a new object is created based on the class structure).
  It is primarily used for initializing an object with some value while creating it. We can have a class without the constructor function as well like in the below example.

```
        class Hero {

        }
        var john = new Hero();
        console.log(john); // Hero {}
```

Now if we want to set some values to the already created object we can have multiple approaches of which two are given below.

The generic way of adding properties to an object.

```
class Hero {

}
var john = new Hero();
john.name = 'john';
console.log(john); // Hero {name: "john"} (Name property added)
```

Through any other function but not constructor

```
class Hero {

    someOtherMethod(name) {
        this.name = name;
        console.log(`${this.name}`);
    }

}
var john = new Hero();
john.someOtherMethod('john');
```

Here we set the name property on the object by passing an argument while calling the someOtherMethod () different from the constructor method.

- Okay. Now what is the "extends" doing out there?

  We know that inheritance happens in JavaScript by virtue of prototypes. The class syntax is a syntactical sugar on this prototype based pattern. In class based syntax we use the "extends" keyword for inheriting a class from another class. Under the hood it is the same prototype based inheritance. One key point to note here is that if a class is using the "extends" keyword, then it has to call the super() method inside the constructor even before initializing any property. Otherwise that throws an error. A class inheritance using the extends keyword is given below.

```
class Parent {

    constructor(name) {
        this.name = name;
    }
}

class Child extends Parent {
    constructor(name) {
        super(name);

        this.name = name;
    }
}

var childObject = new Child('john');
```

- Tell me how many ways we can create a class.
  Classes, like functions can be created in two ways viz class declaration and class expression (named & unnamed). Let's see that in the following code.

  <u>class declarations</u>
  Throughout this article we have done only class declaration. Still one more example would not hurt us right.

```
class Humans {
    constructor(name, age) {

        this.name = name;
        this.age = age;
    }
}
```

class expressions

- ❖ Unnamed class expressions:
  In this we don't give a name to the class directly though we can refer to the class through another variable.

```
var Humans = class {

    constructor(name, age) {

        this.name = name;
        this.age = age;
    }
}
console.log(Humans.name); // Humans
```

- ❖ Named class expressions:
  Here we give name to the class expression as shown in the below code snippet.

```
var Humans = class Human {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
console.log(Humans.name); // Humans
```

- What exactly are the static methods in a class?

Static methods are special methods inside a class which are directly called through the class and they can't be called through the instances. An example to demonstrate the static methods can be as follows.

```
class Humans {

    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    static greet() {
        return `I am a static method!!!`;
    }
}

console.log(Humans.greet()); // I am a static method!!!
```

If we instantiate this class and try to call the greet () method from the object, it throws an error.

- Is there any concept of public and private members in a class?

Yes, we can make class members public and private.

Public members

```
class Humans {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
var john = new Humans('john', 30);
john.name; // john
```

Here name is a public property as it can be accessed from the instance.

Private members

```
class Humans {
    #name;
    #age;
    constructor(name, age) {

        this.#name = name;
        this.#age = age;
    }

}

var john = new Humans('john', 30);
john.#name; // Throws an error for accessing a private property
```

This is pretty much it about the basics of class concept.

this concept in JavaScript

This is a very confusing concept in JavaScript. Many times, it so happens that due to the ambiguity of the "this" value, programmers face nightmares as they can't figure out where the "this" value is changing and how it gets bound. This section is dedicated to reveal some open secrets regarding the "this". We will be discussing four rules which govern the value of "this". Let's check them out one by one.

functions called in the global context

The value of 'this' inside a function becomes global if the function is being called from the global context and it is not explicitly been subjected to the call (), bind () or apply () methods. An example in this regard is given below.

Non-strict mode

```
function foo() {
    return this;
}

foo() == window // true
```

Strict mode
In strict mode, this becomes undefined in the global context.

```
'use strict'

function foo() {
    console.log(this);
}

foo(); // undefined
```

Another example to find a variable inside the window object.

```
var boo = 10;
function foo() {
    console.log(this.bar);
}

foo(); // 10
```

Implicit binding

This is a very important rule which is used very often in our day to day development process. As per this rule, the 'this' value points to the object through which the function is called.

```
var obj = {

    name: 'john',

    foo: function () {
        console.log(`My name is ${this.name}`);
    }

};

obj.foo(); // My name is john
```

Explicit binding:

By using methods such as call (), apply () and bind () we can explicitly bind the value of 'this' to a function. There is a detailed discussion on these methods in a separate section. So here we will only see how explicit binding can be done to bind the 'this' value.

```
function foo() {
  console.log(`I am ${this.name} and I am ${this.age} years old`);
}

var john = { name: 'john', age: 28 };
var sara = { name: 'sara', age: 21 };


foo.call(john); // I am john and I am 28 years old
foo.call(sara); // I am sara and I am 21 years old
```

Similarly, with the bind () method we can accomplish this feat as below.

```
function foo() {
  console.log(`I am ${this.name} and I am ${this.age} years old`);
}

var john = { name: 'john', age: 28 };

var greet = foo.bind(john);

greet(); // I am john and I am 28 years old
```

new keyword binding:

When an object is created using the new keyword, the 'this' value refers to the newly created object as shown below.

```
function Person(name) {
    this.name = name;
}

var john = new Person('john');
john.name // john
```

Now that we have got a fairly good idea about the 'this' concept let's see some of the use cases of it in detail.

❖ Whenever we are trying to access the 'this' value outside of all enclosures, this refers to the window object.

❖ In strict mode if the value of 'this' is not set explicitly by means of call, bind or apply, 'this' value becomes undefined.

❖ Within a class constructor, this is a regular object. All non-static methods within the class are added to the prototype of this.

❖ In arrow functions, this retains the value of the enclosing lexical context. A detailed explanation of the Arrow Functions is available in another section.

❖ When a function is used as an event listener, the 'this' value set is the element upon which the function is invoked. This holds true even if the event handler is set inline.

This is all about the basics of the 'this' keyword.

Memoization in JavaScript

We all have been to college, right. During the start of term, we used to collect books from the college library or source it outside. But the real problem with the books was that they were too lengthy, spanning thousands of pages. So, the effective solution was to go through the entire book once and write down the most important parts in our notebooks. So, during the exam we just needed to go through our personal notebook instead of going through the entire voluminous book. This is exactly the concept behind memorization. Let's see how.

Let's keep the definition simple and the understanding strong. Memoization is an optimization technique in which the results returned by complex functions are cached and in the future if the same functions are invoked with the same arguments, we collect the results from the cache instead of doing the calculation again. Simple enough right. Yeah. But these functions which return the same result with the same arguments are given a special terminology i.e. they are called the Pure Functions.

A function is said to be pure if and only if it produces the same result for the same arguments passed no matter how many times you call that function. An example of a pure function can be as follows.

```
function foo() {
    var a = 11;
    var b = 22;
    return a + b;
}
```

The above function always returns 33, no matter how many times it is called. Memoization is based on these pure functions as they are predictable in their state. An implementation of the memorization optimization varies situation to situation and requirement to requirement. But there are two concepts which are always necessary to implement this. The concepts are as follows.

1) A closure has to be present

2) The underlying function should be a higher order function (A function which takes another function as argument or returns another function is called a higher order function)

A simple use case for the Memoization technique can be that of a function which gives the Fibonacci sequence at a particular position. It takes a number as an argument and returns the Fibonacci series value at that position. For the uninitiated, a Fibonacci series is a mathematical series of numbers whose immediate value is the sum of the values of its immediate two predecessors. So, a Fibonacci series looks something like this.

1,1,2,3,5,8,13,21…

Since the series consists mostly of static values, we will take it as an example and try to implement the Memoization technique for it. For this first we will implement a simple memo function which will take care of the cache thing. Next, we will write a function for calculating the Fibonacci series and pass it as an argument to the previously declared memo () function. To understand things, we will console in between. The functions are written below. Just keep in mind that for implementing Memoization the structure of the memo () function may vary for different cases. The code given below is a general implementation.

```
 function memo(func) {
    var cache = {};
    return function () {
        var key = JSON.stringify(arguments);
        if (cache[key]) {
            console.log(cache);
            return cache[key];
        } else {
            val = func.apply(null, arguments);
            cache[key] = val;
            return val;
        }
    }
}
var fib = memo(function (n) {
    if (n < 2) {
        return 1;
    } else {
        console.log("loading...");
        return fib(n - 2) + fib(n - 1);
    }
});
```

```
fib(10);
// loading...
// loading...
// loading...
// loading...
// loading...
// loading...

// { "0": 0 }: 1
// { "0": 1 }: 1
// { "0": 2 }: 2
// { "0": 3 }: 3
// { "0": 4 }: 5
// { "0": 5 }: 8
// { "0": 6 }: 13
// { "0": 7 }: 21
// { "0": 8 }: 34
// { "0": 9 }: 55
// { "0": 10 }: 89
```

Next when we do

```
fib(5);
```

We get the answer in lesser recursions. Next we do

```
fib(10);
```

We get the result in one function call. That is the power of Memoization. It is now taking from the memory instead of calculating the value. This is pretty much the basic concepts of Memoization.

Polymorphism in JavaScript

Polymorphism is a wonderful concept. But before drowning ourselves in it, let's take a real-life analogy.
Just think about your father. He is a father to you and your siblings. He is the husband of your mother. He is a brother to your aunt and a son to your grandparents. He is one person after all, but he plays different roles to different persons. This is a vague example of polymorphism in real life.
So, for any programming language that supports the Object Oriented Programming concepts Polymorphism is a paradigm that provides a way to perform a single action in different forms. In the context of JavaScript, we can say polymorphism provides an ability to call the same method on different JavaScript objects. It is a practice of designing code to share behavior between objects and to override the default behavior when particular behavior for specific cases is required. The polymorphism concept entirely relies on the inheritance from its parent class.
Let's see this concept in the form of examples.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`I am ${this.name} and I am ${this.age}`);
    }
}

class Engineer extends Person {
    constructor(name, age) {
        super(name, age);
        this.name = name;
        this.age = age;
    }
}
```

```
var john = new Engineer('john', 30);
john.greet(); // I am john and I am 30
```

Here we can see that the john object is an instance of the Engineer class and it has access to the greet () method which comes from the parent Person class. Now in the next code snippet we will see how it can override the greet () method from the parent.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`I am ${this.name} and I am ${this.age} `);
    }
}

class Engineer extends Person {
    constructor(name, age) {
        super(name, age);
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`this is ${this.name} and I am ${this.age}`);
    }

}
var john = new Engineer('john', 30);
john.greet(); // this is john and I am  30
```

Here we can see that the greet () method is overridden and the object calls the greet () method from the Employee class even though we have a greet () method coming in from the parent Person class.

This is all about the basics of Polymorphism in JavaScript.

Map Data Structure in JavaScript

A Map is a newly introduced data structure which holds data in the form of key value pairs. The Map remembers the original order of insertion of the items to it and any type of item can serve as a key or value to a Map. When we iterate through a Map the items are iterated in the same sequence as they are inserted.

Though Map and Objects are very similar in their operations, still there are some striking differences between them. Some of the key differences between Map and Objects are given below.

❏ There are no accidental keys in Map. All the keys are to be inserted by code. But for Objects there is always a prototype object inserted into it with some default values which may collide with the entries of the Object itself.

❏ A Map's key can take any value including functions, objects, strings and even NaN. The keys in Objects must be either of type string or Symbol.

❏ The keys in a Map are ordered according to the sequence of insertion. But in Objects this may not be always true.

❏ There is a size property in Map which returns the number of entries. For Objects no such property exists.

❏ A Map can be directly iterable. For Objects one way to iterate is by getting all the keys and then iterate over it. Another possibility is the Symbol.iterator.

❏ Processes which need insertion and removal very often should prefer Map than Objects.

Enough of information. Now, let's get into the code and make our hands dirty. The code snippets given below are scenario based and, so they are given as points for better memorization of the syntax.

- Creating an empty Map object.

```
        /**
```

```
    * A new Map is created through the new operator.
    * The Map() is the constructor object  used for this operation.
    */
    var map = new Map();
    console.log(map); // Map(0) {}
```

- Adding a new item to an existing Map

```
    var map = new Map();
    map.set('foo', 'bar');
    console.log(map); // Map(1) {"foo" => "bar"}
```

- To check if an entry exists in a Map

```
    var map = new Map();
    map.set('foo', 'bar');
    map.has('foo'); // true
```

- Get the count of entries

```
    var map = new Map();
    map.set('foo', 'bar');
    map.size; // 1
```

- Remove all items from Map

```
    var map = new Map();
    map.set('foo', 'bar');
    map.size; // 1
    map.clear();
    map.size; // 0
```

- Delete a particular item from Map, returns true if item exists && deleted. False if item not present or not deleted.

```
var map = new Map();
map.set('foo', 'bar');
map.delete('foo');
map.size; // 0
```

- get all the entries of a Map

```
var map = new Map();
map.set('foo', 'bar');
map.set('bar', 'baz');
map.entries(); // MapIterator {"foo" => "bar", "bar" => "baz"}
```

- Retrieve an item from the Map

```
var map = new Map();
map.set('foo', 'bar');
map.get('foo'); // 'bar'
```

If the key is not present in the Map, then it returns an undefined.

- Get all the keys of all items in a Map

```
var map = new Map();
map.set('foo', 'bar');
map.keys(); // MapIterator {"foo"}
```

The keys () methods returns an iterator containing all the leys from all the items present in the Map.

- Get all the values present in the Map

```
var map = new Map();
map.set('foo', 'bar');
map.values(); // MapIterator {"bar"}
```

The values () method returns the values of all the entries available in the Map.

- Iterating through a Map
  We can iterate through the Map entries through the for...of loop.

```
var map = new Map();
map.set('foo', 'bar');
map.set('bar', 'baz');
for (let [key, value] of map) {
    console.log(key + ' => ' + value);
}

// foo => bar
// bar => baz
```

We can also use the forEach loop to iterate through a Map like the following example given below.

```
map.forEach(function (value, key) {
    console.log(key + ' => ' + value)
})
```

```
    foo => bar
    bar => baz
```

- Converting an associative array into Map
  An associative array is an array which has key and value relationship between its items. A simple example of this kind of array is given below.

```
var arr = [['key1', 'value1'], ['key2', 'value2']];
```

Now we convert the above associative array into a Map by passing the array into the Map() constructor as shown below.

```
var map = new Map(arr);
console.log(map);
// Map(2) {"key1" => "value1", "key2" => "value2"}
```

- Converting a Map to an Array

  We can convert a Map to an array by using the Array. from() method like below code snippet.

```
var arr = [['key1', 'value1'], ['key2', 'value2']];
var map = new Map(arr);
var arr = Array.from(map); // We get back the array here
```

- Clone an existing Map

  We can clone an existing Map to another Map in the following way.

```
var map = new Map();
map.set('foo', 'bar');
```

```
var clonedMap = new Map(map);
console.log(cloedMap); // Map(1) {"foo" => "bar"}
```

- We can also merge multiple maps like following code snippets.

```
var map1 = new Map();
map1.set('foo', 'bar');

var map2 = new Map();
map2.set('bar', 'baz');

var mergedMap = new Map([...map1, ...map2]);

console.log(mergedMap);
// Map(2) {"foo" => "bar", "bar" => "baz"}
```

Above are some of the most important points to keep in mind while dealing with Map data structure.

Set Data Structure in JavaScript

A SET is a data structure with a collection of values. The SET is iterated in the same sequence of values as they were inserted. Another important aspect of SET is that it only contains unique values. Because of this property we use Sets to remove duplicates from arrays which we will see in a while. Let's jump into the code straightaway and see some of the key aspects of coding with Sets.

- Create a new Set

  The Set () constructor is used to create a new Set like below.

```
var set = new Set();
console.log(set); // Set(0) {}
```

  Or we can also initialize the Set while creating it in the same statement as below.

```
var set = new Set([1, 2, 3, 4, 5]);
console.log(set); // Set(5) {1, 2, 3, 4, 5}
```

- Checking if a value exists in a Set

  We can check if a value exists in a Set or not by the has () method.

```
var set = new Set([1, 2, 3, 4, 5]);
console.log(set.has(1)); // true
console.log(set.has(6)); // false
```

- Get the length of a Set

We can get the length or the number of items in a Set by the size property.

```
var set = new Set([1, 2, 3, 4, 5]);
console.log(set.size); // 5
```

- Add new elements to a Set.

  We can add new items to a Set by the add () method. It adds a new item at the end of the Set. Since the Set takes values sequentially in the way they are added we can only add new items at the end.

```
var set = new Set([1, 2, 3, 4, 5]);
set.add(6);
console.log(set); // Set(6) {1, 2, 3, 4, 5, 6}
```

- Remove all the items from a Set

  We can use the clear () method to remove all the items from a Set.

```
var set = new Set([1, 2, 3, 4, 5]);
set.add(6);
set.clear();
console.log(set); // Set(0) {}
```

- Delete a particular element from the Set

```
var set = new Set([1, 2, 3, 4, 5]);
set.delete(2);
console.log(set); // Set(4) {1, 3, 4, 5}
```

- Get all the key-value pair in Set

  It is a little bit tricky here. We don't have a key: value pair in Sets. So essentially when we execute the entries () method on a Set it gives an iterable array of [value value] pairs where value is essentially the same with value. This is done to bring uniformity with Map. A detailed guide about Maps is given in another section.

```
var set = new Set([1, 2, 3, 4, 5]);
console.log(set.entries());
// SetIterator { 1 => 1, 2 => 2, 3 => 3, 4 => 4, 5 => 5 }
```

- Looping through a Set
  We can use the forEach loop to loop through a Set like below code example.

```
var set = new Set([1, 2, 3, 4, 5]);

set.forEach(function (item) {
    console.log(item);
});
```

The above code logs all the items of the Set onto the console.

We can also use the for...of loop to iterate through a Set like below code snippet.

```
var set = new Set([1, 2, 3, 4, 5]);
for (let item of set) {
    console.log(item);
    }
```

- Get all the values from a Set

  We use the values () method to extract all the values from a Set.

```
var set = new Set([1, 2, 3, 4, 5]);
console.log(set.values()); // SetIterator {1, 2, 3, 4, 5}
```

- Creating a Set from an array

```
var arr = [1, 2, 3, 4, 5];
var set = new Set(arr);
console.log(set); // Set(5) {1, 2, 3, 4, 5}
```

- Removing duplicate values from an array

```
var arr = [1, 2, 2, 2, 3, 4, 4, 5, 5, 5];
var set = new Set(arr);
console.log(set); // Set(5) {1, 2, 3, 4, 5}
```

This is pretty much the basics of Set operations and it can be a good starting point for your future encounters with Set data structure.

Promises in JavaScript

There were a father and a son living in some remote part of the country. The father daily used to send the son to fetch some food. Since it was a far-off place the market was not always open. Some days it was open and some days it was not. If the market was open the son would bring some nice pizzas and the father-son duo would feast on that. If the market is closed, then the son would come back empty handed and they would cook at home. But for sure the son would come back to home regardless of the status of the market. So, there are three situations in this story. First is that the market would be opened, and the son would bring pizzas. Second is that the market will be closed, and they will cook. Third is the interim duration when the son is gone to the market and the father has no clue as to whether the market is open or close. But it so happens that under the hood the son before going to the market promises his father that whether or not the market is opened, he will be back for sure. This is a promise he makes to his father and he always keeps the promise.

JavaScript promises more or less follow the same pattern. They may be successful or failure, but eventually they come back. Now let's go to the inner aspects of a JavaScript promise.

A promise is a placeholder for a future value when we are dealing with asynchronous JavaScript. Promises are always asynchronous. Wait. What is this asynchronous thing? As we know JavaScript is a single threaded language and it executes the code line by line, from top to bottom. Imagine a situation where you need to make an API call to get some results from a server. Then in the normal flow what would happen is that once the interpreter comes across this line and a request is sent, then it would stop there idle and wait for the response. In that process the next block of code does not get executed until the response from the previous call has arrived. This is a typical case for synchronous operations where the interpreter gets blocked. This is Blocking issue. Now let's see the same process with a different perspective. In this case the interpreter reaches the code and makes the request. But it does not wait for the response or does not get blocked.
It goes to the next block of code and executes it and then moves onto the next block as well. Now when the response comes back from the API call the interpreter gets a notification about it and it goes back inside the code block which made the API call and does what needs to be done when the data from the server is available. Once this operation is done the interpreter goes back to the normal execution flow and start from where it left previously. This style of execution is known as asynchronous mode. Next, we will write some code to see how this asynchronous thing works and how to handle it. So, without further ado let's get started.

The simplest way of writing a Promise is as follows.

```
var promise = new Promise(function (resolve, reject) { });
console.log(promise); // Promise {<pending>}
```

Well, so many things are happening here. Let's discuss each of them one by one.
What is the new Promise () thing! Well, it is the way to create a new promise object through the 'new' keyword. We are using the Promise () constructor for this. It is that simple. And this constructor takes a function as an argument. It is in the sole discretion of the developer to use a normal function or an arrow function. It's a matter of personal choice.
Next is the parameters being passed to the constructor. In relation to the above story, if the son comes back with the pizza, then his purpose of going to the market was achieved right. Same way if the operation with the Promise is successful, then the first parameter resolve is invoked. But if the Promise is failed then the second parameter reject is invoked. As simple as that. If we remove the parameters, then it is still a Promise, but it will not be that useful. From the above discussion we can easily conclude that a Promise can have three states. When it is successful, it is said to be resolved, when it is failed, it is called rejected and when it is in the interim period of the operation it is in the pending state.

Now let's extend our example to the next step.

```
var promise = new Promise((resolve, reject) => {
    var x = 10;
    if (x === 10) {
        resolve('Promise is resolved');
    }
});

promise.then((message) => {
    console.log(message);
});
```

Here everything remains the same except for the fact that we are calling the resolve method with a string argument and we have an extra then () block. Like we discussed above when the promise is successful, we call the resolve method with some argument depending on the use case. Here we are just passing a string for this purpose. Now, in the then () block we are capturing the result of the successful promise. So essentially whenever a Promise is successful, the then () method is called. In our case we are just capturing the message passed from the resolve method and printing it onto the console.

Now in the next step we will see how to handle a failed promise.

```
var promise = new Promise((resolve, reject) => {
    var x = 10;
    if (x === 33) {
        resolve('Promise is resolved');
    } else {
        reject('Promise is rejected');
    }
});

promise.then((message) => {
    console.log(message);
})
    .catch((errMessage) => {
        console.log(errMessage);
    })
```

Here we have just mocked a scenario when the Promise is failed, and we pass a message through the reject method. We capture the message inside the catch () block.

Now let's see a promise with an asynchronous operation with the setTimeout () method which is an a-sync window method.

```
var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Promise Resolved!!!');
    }, 2000);
});
promise.then((message) => {
    console.log(message);
```

```
        }).catch((errMessage) => {
                console.log(errMessage);
        })
```

Now let's see a very important concept called Promise chaining.

In a promise, when the then() block is executed, it returns a new promise. We can capitalize on this fact and return a new promise from the then() block and then capture the returned promise in a subsequent then() block.

We can do this process multiple times and this is known as promise chaining. Please note that having multiple then() blocks for the same promise does not entitle to be called as a promise chain. Let's see one example of Promise chaining.

```
        new Promise((resolve, reject) => {

            resolve(10);
        })
            .then((result) => {
                return result * 2;
            })
            .then((result) => {
                console.log(result * 3);
            });
```

As we can see here that the first then() block returns a new promise and it is captured in the next then() block which prints the result after multiplying by three. This is an example of promise chaining.

What if you have multiple promises and you want to operate on them after all of them have been successful. In this case Promise.all () method is the way out.

```
var prom1 = new Promise((resolve, reject) => {
    resolve(1);
});


var prom2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Promise Resolved');
    }, 2000);
});

Promise.all([prom1, prom2])
    .then((values) => {
        console.log(values);
    });
```

Here we have two promises whose references we pass into the Promise.all () method in the form of an array. In the then () block all the resolved values are printed simply.

But there are a couple of things to notice in the Promise.all () method. First thing is that the Promise.all () method waits till all the provided promises are resolved. Secondly there should not be any rejected promise. Once it encounters a rejected promise it throws an error instantly.

This is pretty much the basics about promises.

Higher Order Functions and Callbacks

A function is a doer. Functions are the first among the equals and they are called as the first class citizens in JavaScript. The reasons for this special tag and treatment are as follows.

- They can be declared
- They can be written as expressions
- They can be passed as arguments to other functions
- They can be returned from other functions
- Functions can be passed around as objects and they have definite value

From the above-mentioned points, point (iii) and (iv) are enablers for Higher Order functions. Higher Order Functions is a Mathematical concept but for the sake of simplicity we will just tread the JavaScript path. A function which takes another function as an argument and a function which returns another function as return value is a higher order function. This capability opens a whole new world of opportunity for us. Let's see how we can take advantage of these capabilities.

Functions taking another function as argument

We as JavaScript developers have passed functions as arguments to other functions multiple times knowingly or unknowingly. A small example of this can easily clear the air around this.

```
setTimeout(function () {
    console.log('I am a higher order function');
}, 0);
```

Here we are passing an anonymous function as an argument to another function i.e. the setTimeout (). If we want to further decouple the inner function, we can do something like below.

```
    function foo() {
        console.log('I am a higher order function');
    }
    setTimeout(foo, 0);
```

This process of passing functions to other functions as a value, opens up a lot of opportunities for us. With this approach we can define one standalone function and use it multiple times and when the function is executed it will do as per the context of the callee. Another simple example can prove this fact easily.

```
<a href="#" id="anchor">Click Anchor</a>
<button id="btn">Click Button</button>
<script>
    var anchor = document.getElementById('anchor');
    anchor.addEventListener('click', foo);
    var btn = document.getElementById('btn');
    btn.addEventListener('click', foo);
    function foo() {
        console.log(this.id);
    }
</script>
```

As we run this code and click on the button and the anchor element, we can see the respective ids being printed on the console. Here we are passing the same function to both the event listeners and are able to get a reference to the element which was clicked. This is the power of higher order functions being passed as references.

Functions returning another function

Have you heard about closures or function currying??? You must have at some point of time encountered this if you have been programming in JavaScript for some time now. Anyway, let's see an example of this type of function.

```
    function multiply(a) {
        return function (b) {
            return a * b;
        }
    }
    multiply(3)(3);
```

A complete section named closures about these types of functions is provided in this book. You can check that out to find more about closures.

Callback functions

A callback function is a function which is provided as an argument to another function and it gets executed once some operation/event occurs. The callbacks are an essential part when it comes to event listeners and when dealing with a-sync operations. A simple example of the callbacks is given below.

```
<button id="btn">Click Button</button>

<script>

    var btn = document.getElementById('btn');
    btn.addEventListener('click', foo);

    function foo() {
        console.log(this.id);
    }

</script>
```

This is all about higher order functions and callbacks.

Value vs Reference Assignments

This section is going to be very short, really short. But this is a very important concept and you will often encounter this concept in interviews as well as while developing applications. Let's finish it quickly as grab all the important concepts on the way.

We have 9 data types in JavaScript. Those are listed below.

- Boolean
- Number
- String
- Null
- Undefined
- Symbols
- BigInt
- Object (Objects, Arrays, Functions are included in the Object data type)

Out of all the above data types, the Object data types are reference types and all others are value types. Shortly, we will come to know about the value type and the reference types. Let's start with the value types.

Value type assignments

The value type assignment means whenever we assign a value to the variable, the variable identifier holds the value assigned. If we do like this;

```
var x = 10;
```

the identifier x holds a value of 10. It is as simple as that. Now when we reassign;

```
x = 20;
```

Now the variable x holds the value of 20. Next when we do

```
var y = 20;
var z = y;
y = 30;
console.log(z);
```

Here even if we reassign the value of y after assigning it to z, the value of z does not change after that and still holds 20. This same thing will apply for other data types like Boolean, Number etc.

Reference type assignments

At some point of time in your academics or development career, you must have come across the concept of Pointers in C programming. A Pointer essentially contains the address of the memory location where the value is stored instead of the value of the variable itself. This is the case with Reference type assignments. In JavaScript the Object data types are reference types. Let's see to that.

```
var arr = [1, 2, 3, 4];
var clonedArr = arr;
arr.push(5);
console.log(arr); // [1, 2, 3, 4, 5]
console.log(clonedArr); // [1, 2, 3, 4, 5]
```

Here when we declare the arr, it points to the memory location, not to the values [1,2,3,4]. Then we clone the array into clonedArr variable and it contains the value of the arr variable i.e. the reference to the memory location. Next when we do a push operation, there is a change in the content of the memory location, but not the value held by the array. So now the memory location contains [1,2,3,4,5]. So, when we console the arr variable, it prints the contents inside the memory location i.e. [1,2,3,4,5]. But since the clonedArr also refers to the same memory location, it also prints the same content. This concept is also true when we are dealing with objects and functions. This is the gist of assignment by reference.

Coding Standards and Best Practices in JavaScript

Most of the time writing working code is not enough. The industry demands good code. Like any other programming language, JavaScript also has certain best practices which a developer has to follow. Ultimately it is the coding standards that differentiates a novice from a ninja. Otherwise most of us developers write working code, isn't it? In this section we will go through some of the best practices which we should adhere to while writing JavaScript code.

➢ Don't write code for yourself. Write it for the next developer who may have to work on your code in future. Please name your variables and functions such that they make sense. Don't try to make them very explicit by using articles or jargons, but name them in such a way that they clearly say what they are doing. For the sake of showing off your technical prowess, please never use any sort of extravagant names to your entities. They will do more harm than good to your reputation. Please don't use x, y, z, p, q etc. as variable names. Because maybe these letters make sense to you now when you are working on this particular code block, but they will make very less sense when you come back to this block again after a week or so. So always give them names which are expressive. But, also take care of the fact that you don't give names like sentences just to make them overly expressive.

➢ While writing code please avoid the global space. Don't pollute it with your variables and function. This area is not garbage collected and so the variables remain forever in the memory. Which may slow down your application or lead to memory leakage. A detailed section about garbage collection mechanism is provided in this book. Please refer to see how the browser handles garbage collection and the necessity of avoiding global will be evident from that. Always wrap your variables and functions inside an object or inside a module to prevent them from spilling onto the global namespace.

➢ Always write your code in strict mode. That way you may find a lot of issues pertaining to possible mistakes.

➢ While writing code always try to use a linter so that your entire application follows the same linting rules. TSLint or any other tool is a good point to start with.

➢ Please don't clutter your code with comments. In my career I have seen several developers who write big essays like comments before each function block which they think will help the other developers to understand what the function block is doing. The intention is good. But is it really necessary? I happened to see some developers writing big comments because they find it fancy because most of the IDEs available today show the comments in a different color and font. But is it a good approach? Definitely not. Please don't write comments unless it is very much necessary. What you are writing is code and it is already a masterpiece. Please don't ruin it by writing unnecessary comments. Rather, write your code in such a way that it is very understandable. Name the entities in such a way that they are self-explanatory. In fact, writing voluminous comments is now old fashioned and it is no more a badge for your technical foresightedness.

➢ Try to write shorthand notations when it makes sense like the example given below.

```
var arr = [];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
```

Instead we can directly write it in one statement like

```
var arr = [1, 2, 3, 4];
```

It is the same thing, is not it. Many times, we do things like below to objects.

```
var obj = {};
obj.name = 'john';
obj.age = 29; //  etc etc..
```

But we can write it in the below fashion.

```
var obj = { name: 'john', age: 29 };
```

Many times, we do injustice to the if statements like below.

```
var isTrue = true;
var num1 = 10;
var num2 = 20;
var sum = 0;
if (isTrue == true) {
    sum = num1 + num2;
} else {
    sum = num2 - num1;
}
```

This code looks super simple, but we can make it elegant like below by using the ternary operator.

```
var isTrue = true;
var num1 = 10;
var num2 = 20;
var sum = 0;

sum = isTrue ? num1 + num2 : num2 - num1;
```

See. Writing some standard code is that easy.

➢ Follow the SRP. SRP stands for Single Responsibility Principle. It essentially means that one code block should do one and only one task. That way your code becomes more readable, more testable and more maintainable. The benefits of all the worlds are at your disposal. From now on don't ever think of writing very complex and big functions. Rather write simple and sober functions so that if any bug comes, it can be detected easily.

- ➢ Progressive enhancement is the key for a better user experience. The browsers are many and you never know if the user has disabled scripts on his system. So, the best bet in this case would be to first give the user the bare minimum things without JavaScript and then you enhance the functionality of your application progressively.

- ➢ Try to avoid nested code. Nested code comes with a huge price. There will always be a possibility of callback hell, some rat holes for bugs, a racing condition or an unmaintainable and untestable code base. Why to fall into all these traps. Rather divide all the nested things into smaller functions and just give a reference to them if in the worst case you just can't avoid them.

- ➢ Use some sort of internationalization and configurable settings so that when the static texts change due to a change request you have to change only in one file rather than altering the texts scattered around the whole application.

- ➢ Always try to keep the static calculations to a bare minimum level. A simple example to demonstrate this fact is given below.

```
var arr = [1, 2, 3, 4, 5];
 for (var i = 0; i < arr.length; i++) {
     console.log(arr[i]);
 }
```

But there is a small problem with the above code. For every iteration of the loop, it has to calculate the length property of the array which is rather static and does not change during the execution of the block, isn't it. We can optimize this issue as follows.

```
var arr = [1, 2, 3, 4, 5];
for (var i = 0, j = arr.length; i < j; i++) {
    console.log(arr[i]);
}
```

These kinds of simplifications we can always do in our code, right.

- ➢ Try to do DOM manipulations and access the DOM as minimum as possible. It is a costly affair. And always be careful to remove the event listeners at the end of the file. Please try to avoid placing static content through JavaScript. Rather use the power of the language to write the logic.

- ➢ If you are using external libraries in your application, try to keep the number of libraries to a bare minimum for a lean build. And also keep in mind that you always use established and stable libraries rather than some other library which is giving away some new fancy stuff.

The process of optimization is continuous, and it varies from situation to situation. A well optimized code is a delight.

Concurrency and Event Loops

We all know that JavaScript is a single threaded programming language. What does that mean? That simply means that JavaScript reads our code line by line and it does one and only thing at one point of time. In this section we will see how JavaScript handles our code under the hood. Let's get started.

Let's see a very simple function which has only three console statements and we will see how it is getting interpreted by the browser.

```
function foo() {
    coonsole.log('one');
    console.log('two');
    console.log('three');
}
foo();
```

Nothing fancy right. We have a function foo () with three console statements and we invoke the function after the declaration. Pretty straight forward. But for the interpreter it is something different than what we see. Let's get into the shoes of the interpreter.

What if I say the code above is governed by something known as a CALL STACK. Yeah, it's true. So, let's try to understand what a call stack is in very simple terms.

A call stack is a mechanism which tells the interpreter about its position. Since it is a stack data structure, it essentially follows the Last In First Out (LIFO)model.

In our case when we call the function foo (), the interpreter promptly pushes it to the call stack. Since this is the only item in the call stack the interpreter steps into the function and finds the first console statement. It pushes the first console statement onto the call stack and executes it.

In this process 'one' is printed and then this console statement gets popped out since the execution is done. On the next statement the second console is encountered, and it is pushed into the call stack. It gets executed and 'two' is printed on the console and it is popped out. Remember the foo () function is still in the call stack and it is at the bottom. Next the third console is executed after pushing it into the call stack and then it is popped out.

After this the function stops execution and it gets popped out. Now that there is nothing else to be executed inside the function and so finally it gets popped out. This is how the call stack works in normal working conditions.

But what will happen if there is an asynchronous operation like a network call, an image processing or a setTimeout () is part of the process. Let's see that through a code sample.

```
function foo() {
    console.log('one');
    setTimeout(function () {
        console.log('two');
    }, 2000);
    console.log('three');
}

foo();

// one
// three
// two
```

Now that we know the answer, let's analyze what is happening inside the browser.

We know that there are certain APIs such as setTimeout which are asynchronous, and they are not part of the JavaScript spec, rather they are part of the browser API which in other terms we call as Web APIs. To analyze the above result let's understand two more data structures available in the browser viz. the heap and the Queue. The sole role of the Heap is to store the variables in the program and they just facilitate the storage of data. The Queue data structure as we know it follows the First In First Out(FIFO) mechanism. Now let's get back to our code.

The first statement that gets executed in the above code is the invocation of the foo () function. So, the interpreter pushes the foo () function into the call stack. Then the interpreter steps into the function body and finds the first console statement, executes it, prints 'one' onto the console and then pops it.

When it encounters the setTimeout () statement, it pushes that block to the call stack. Since it is a web API and it needs to wait for 2000ms before executing the inside code, the interpreter moves this block from the call stack to the Web API for processing where the timer starts. Next the interpreter moves onto the next statement i.e. the third console statement, pushes it to the call stack, executes it, pops it out of the stack. Now, that the interpreter has reached to the end of the call stack and there are no more statements to be executed inside the function, the function foo () is removed from the call stack. So now the call stack is empty.

As soon as the setTimeout is finished processing in the Web API section and the asynchronous operation is completed, it is moved to the Queue. Now it is the only item available in the Queue. Now that the call stack is also empty, the setTimeout statement is pushed to the call stack and then the inside console statement is executed, and it prints 'two' onto the console. If the call stack would not have been empty, the setTimeout would not be pushed from the Queue to the stack. This is precisely the role of the event loop. It constantly monitors the call stack and if it is empty it pulls the first item from the Queue to the call stack.

This is the overall working mechanism of the concurrency and event loops in the browser.

How V8 Engine Works

We know that behind the power of every browser there is an engine that drives it. Below is the list of some of the major browser engines that propel some of the very popular browsers.

Browser Name	**Engine**
Google Chrome	V8
Firefox	SpiderMonkey
Edge	Chakra
Safari	JavaScriptCore

Out of the above listed browsers, the V8 engine is the most powerful browser engine available today. The internal working of the browser is so complicated that it will require millions of lines to explain everything about them. Since we as JavaScript developers need to only know the most essential parts of the V8 engine, let's save the scholarly article for some other day. Here in this article we will just explore the essentials of the engine in the form of pointers.

- ➢ The source code of V8 engine is written in C++
- ➢ It compiles and executes our JavaScript code
- ➢ It handles the memory allocation for our applications
- ➢ It handles the call stack for us
- ➢ It takes care of the Garbage collection

- ➢ The V8 engine uses the Just In Time(JIT) compilation method. In this process an unoptimized machine code is generated from the source code. This process happens before execution.

- ➢ The machine code generated by JIT is not optimized and it takes a lot of space. Further, optimization needs to be done on this code.

- ➢ In the next step Ignition optimizer comes into picture and it further processes the generated code. It basically reduces the memory overhead and makes the binaries more compact.

- ➢ In the next step, two more optimization compilers called TurboFan and CrankShaft come into picture and they provide further optimization of the generated code. Now V8 has a new Web Assembly baseline compiler in place for compiling source code.

- ➢ The garbage collection process is taken care of by a component called Orinoco. There is a complete section devoted to garbage collection in this book. You could check that if you have not done that by now.

This is the basic process in which V8 compiles your source code.

IIFE and Arrow Functions

This section is about two special variants of functions in JavaScript. One is Immediately Invoked Function Expression (IIFE) and the other one is Arrow Functions. Let's take them one by one.

Immediately Invoked Function Expression(IIFE)

First thing first. Let’s first analyze the problem.

Let's see a very simple function which adds two numbers and prints the result onto the console. This is a function which we must have seen millions of times. Let's do it once more.

```
function sum() {

    var a = 10;
    var b = 20;
    console.log(a + b);
}

sum();
```

Simple enough right. For this function to work we just need to invoke it as many times as we want, and it will print the result for us.

But in a real scenario we would just want to do it once because getting the same result multiple times makes very less sense. And another aspect of it is that our variables used inside the function are not accessible from outside of the function i.e. the variables are locally scoped.
So here we have a challenge. We need to modify our function in such a way that it gets called once and preferably no callee required and to preserve the local scope. Let's do it step by step

➔ step1
First let's wrap our function body with parentheses.

```
(function sum() {

    var a = 10;
    var b = 20;
    console.log(a + b);
})
```

Wait! It looks very similar to a function expression albeit with a name. Oh yeah. This is a named function expression. But since we don't intend to call the function by ourselves, we can try removing the name identifier in the next step.

➔ step2
Let's remove the name

```
(function () {

    var a = 10;
    var b = 20;
    console.log(a + b);
})
```

➔ step3
Now we have an unnamed function expression. But what do we call it because we need to execute the function at least once. We have been habituated to call a function by the

parenthesis. How about writing a pair of enclosing parentheses at the end of the function body and try our luck at executing it.

```
(function () {

    var a = 10;
    var b = 20;
    console.log(a + b);
})()
```

Hurray!!! It worked. Just the right logic applied. Bang on.

What will happen if we call the function just inside the closing parenthesis of the function body.

```
(function () {
    var a = 10;
    var b = 20;
    console.log(a + b);
}())
```

It also gives us the same result. So, we can write our new invention in both the ways.

Wait. This is not something new. What we have done has been there in the JavaScript ecosystem for quite some time now and fondly we call it IIFE. Ha ha ha!

So, we can safely conclude here that we achieved an Immediately Invoked Function Expression which is within parentheses and which invokes itself as soon as the interpreter reaches it. Now that we have got some brief understanding of the IIFE and what its structure is, let's now explore more about it by understanding the most striking features of this type of function.

Key takeaways from Immediately Invoked Function Expression

- An IIFE invokes itself as soon as it is defined. We don't need to explicitly call it.

- IIFE is based on a design pattern known as "Self-Executing Anonymous Function" and it has two major parts. First is the grouping operator () which creates a lexical scope around the function body and the second part is calling the function upon its body by which the engine will directly interpret the function.

- Variables declared inside IIFE are immutable.

  For your information I would like to say that immutable in this context simply means that the variables declared inside the IIFE can't be changed from outside of the function body. In Fact, they can't be accessed.

```
(function () {

    var a = 10;
    var b = 20;
    console.log(a + b);
}())
console.log(a); // throws a reference error
```

- We can pass parameters to the IIFE.
  An example to demonstrate the supply of arguments to an IIFE is given below.

```
(function (name) {

    console.log(name);
}('john'))
```

This is all pretty much the basics of the Immediately Invoked Function Expressions. Now let's switch to the Arrow Functions mode.

Arrow Functions

Multiple surveys were conducted about the most popular ES6 feature and 70% of the respondents said that Arrow functions are their favorite feature in ES6. Arrow functions have changed the dynamics of how we write functions. With the advent of the Arrow functions era, the functions in JavaScript have become more concise to read and they are now more predictable in handling 'this' value. First let's explore the various syntaxes of the Arrow functions with a comparison with the ES5 functions and then we will dig deeper into the implications of using these.

Different syntaxes of Arrow Functions:

❖ With multiple arguments:

ES5 Syntax

```
const foo = function (a, b) {
    return a * b;
}
```

ES6 Syntax

```
const foo = (a, b) => { return a * b; }
foo(3, 3);
```

❖ With only one argument

ES5 Syntax

```
const foo = function (name) {
    return name;
}
```

ES6 Syntax

In this case parentheses are not required.

```
const foo = name => name;
foo('john');
```

❖ No parameters

ES5 Syntax

```
var foo = function () {
    console.log('I am an ES5 function');
}
```

ES6 Syntax

```
var foo = () => { console.log('I am from ES6 Arrow functions'); }
```

❖ Returning Object literal

ES5 Syntax

```
function foo(id, name) {
    return {
        id: id,
        name: name
    };
}
```

ES6 Syntax

```
    var foo = (id, name) => { return { id, name } };
    foo(1, 'john');
```

The most common implication of Arrow functions

The ES5 functions define their own 'this' value based on how they are getting called. A detailed explanation regarding the rules for picking 'this' value is given in another section. Let's see an example.

```
    function Foo(age) {
        this.age = age;

        setTimeout(function () {

            console.log(this.age);
        }, 2000)
    }

    var x = new Foo(30);
```

The code above prints undefined as inside the setTimeout () the 'this' value refers to the Window object and there is no property named 'age' in the window object.

This is a very common problem in JavaScript. For years we Java Scripters have been following the below practice to overcome this issue.

```
function Foo(age) {

    var that = this;
    that.age = age;

    setTimeout(function () {

        console.log(that.age);
    }, 2000)
}

var x = new Foo(30);
```

Here we set another variable like 'that' or 'self' to keep the 'this' value bound to the current context. With ES6 Arrow Functions this problem has been done away with.

```
function Foo(age) {

    this.age = age;
    setTimeout(() => {
        console.log(this.age);
    }, 2000);
}
var x = new Foo(30);
```

This works perfectly fine and we don't need to do any workarounds for binding the 'this' value. This is the power of Arrow Functions.

This is pretty much the basics of the Immediately Invoked Function Expression and Arrow Functions.

For more interesting topics and to learn various technologies please visit http://code2stepup.com/

THANK YOU