1. JavaScript Prototype

Let me explain the various way of creating objects in JavaScript. One of the ways to create objects in JavaScript is the constructor function. Consider the below constructor function:

```
function Bike(model,color){
   this.model = model,
   this.color = color,

this.getDetails = function(){
     return this.model+' bike is '+this.color;
   }
}
var bikeObj1 = new Bike('BMW','BLACK');
var bikeObj2 = new Bike('BMW','WHITE');

console.log(bikeObj1.getDetails()); //output: BMW bike is BLACK
console.log(bikeObj2.getDetails()); //output: BMW bike is WHITE
```

In the above example, we are creating two objects, bikeObj1,bikeObj2 using a constructor. In JavaScript, every object has its own methods and properties. In our example, two objects have two instances of the constructor function `getDetails()`. It doesn't make sense having the copy of `getDetails` doing the same thing.

Instead of using the copy of instances, we are going to use prototype property of a constructor function.

Prototype

When an object is created in JavaScript, JavaScript engine adds a `__proto__` property to the newly created object which is called `dunder proto`. `dunder proto` or `__proto__` points to the prototype object of the constructor function.

```
function Bike(model,color){
  this.model = model,
  this.color = color
}

Bike.prototype.getDetails = function(){
 return this.model+’ bike is ‘+this.color;
}
var bikeObj1 = new Bike(‘BMW’,’Black’);
```

```
console.log(bikeObj1.getDetails());
```

The `bikeObj1` object, which is created using the Bike constructor function, has a `dunder proto` or `__proto__` property which points to the prototype object of the constructor function `Bike`.

In below code, both `bikeObj1` it's `dunder proto` *or* `__proto__` property and `Bike.prototype` property is equal. Let's check if they point at the same location using `===` operator.

```
console.log(bikeObj1.__proto__ === Bike.prototype );

//output : true
```

TK [Not sure what the author intends to say here] So using prototype property, how many objects are created functions are loaded into memory only once and we can override functions if required.

2. JavaScript(ES6) Class

JavaScript classes, introduced in ES6, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax **does not** introduce a new object-oriented inheritance model to JavaScript. In early ES5 using function expression.

existing Prototype based inheritance:

```
function Bike(model,color) {
    this.model = model;
    this.color = color;
}

Bike.prototype.getInfo = function() {
    return this.color + ' ' + this.model+ ' bike';
};
```

Defining classes:

Classes are in fact "special functions", and just as you can define function expressionsand function declarations, the class syntax has two components: class expressions and class declarations.

ES6 class:

```
class Bike{
  constructor(color, model) {
    this.color= color;
    this.model= model;
  }
}
```

Benefits of Using class

- Convenient, self-contained syntax.
- A single, canonical way to emulate classes in JavaScript. Prior to ES6, there were several competing implementations in popular libraries.
- More familiar to people from a class-based language background.

3. IIFE

What is an IIFE in JavaScript?

IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function ()
{// logic here })
();
```

Your first encounter may seem quite confusing but actually, the pattern is simple. The pattern is immediately invoked function expression.

JavaScript functions can be created either through a function declaration or a function expression. A function declaration is the “normal” way of creating a named function.

A function created in the context of an expression is also a function expression. The key thing about JavaScript expressions is that they return values.

In both cases above the return value of the expression is the function.
That means that if we want to invoke the function expression right away we just need to tackle a couple of parentheses on the end. Which brings us back to the first bit of code we looked at:

```
(function ()
{ var foo = “hello”;
console.log(foo);
 })
();
console.log(foo); //Error: foo is not defined
```

The primary reason to use an IIFE is to obtain data privacy. Because JavaScript’s var scopes variables to their containing function, any variables declared within the IIFE cannot be accessed by the outside world.

4. JavaScript Closures

What is Closure?

> *A* closure *is the combination of a function and the lexical environment within which that function was declared.*

A closure is an inner function that has access to the outer (enclosing) function’s variables — scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

Let see a closure example below:

```
function User(name){

  var displayName = function(greeting){
   console.log(greeting+' '+name);
  }
return displayName;
}

var myFunc = User('Raj');

myFunc('Welcome '); //Output: Welcome Raj
myFunc('Hello '); //output: Hello Raj
```

In this code, We have an outer function `User()` which returns an inner function as `displayName()`,

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

5. The Module Pattern

In JavaScript, a module is a small unit of independent, reusable code. Modules are the foundation of many JavaScript design patterns and are critically necessary when building any non-trivial JavaScript-based application.

JavaScript module export as the value rather than define a type, as JavaScript JavaScript module can export an object, Modules that export a string containing an HTML template or a CSS stylesheet are also common.

JavaScript doesn’t have private keyword but we can achieve private methods and properties using closures.

```
var myModule = (function() {
    'use strict';

    var _privateProperty = 'Hello World';

    function _privateMethod() {
        console.log(_privateProperty);
    }

    return {
        publicMethod: function() {
            _privateMethod();
        }
    };
}());

myModule.publicMethod();                    // outputs 'Hello World'
console.log(myModule._privateProperty);     // is undefined
protected by the module closure
myModule._privateMethod();                  // is TypeError protected
by the module closure
```

these modules can have exported to the other JS files using the export keyword,

```
//myMOdule.js file
```

```
export default myModule;
```

modules can import to another JS file

```
//second.js file
import myModule from ‘./myModule’;
```

Why do we need to use Module?

There are a lot of benefits to using modules in favor of a sprawling,

some are,

1. maintainability
2. reusability
3. Namespacing

6. Hoisting:

In JavaScript what its hoisting means,

> *Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.*

It's actually a definition for hoisting,

In simple explanation for Hoisting with code,

```
console.log(Hoist);
var Hoist = ’The variable Has been hoisted’;
//output : undefined//
```

actually, JavaScript has hoisted the variable declaration. This is what the code above looks like to the interpreter

```
var Hoist;
console.log(Hoist);
```

```
Hoist = ’The variable Has been hoisted’;
```

JavaScript only hoists declarations, not initialization.

Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

so this also match for a variable of function level scope also hoisted

Next thing you need to know about Hoisting,

1. let, var, const keyword in JavaScript (ES6)
2. Hoisting Functions
3. Hoisting classes

You can elaborately learn about Hoisting with this site and also I had refereed for Hoisting *Understating-Hoisting-in-JavaScript*

7.understanding Scope:

A simple definition for a scope in JavaScript:

> *Scope is the accessibility of variables, functions, and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.*

As per the above definition of Scope, So, the point in limiting the visibility of variables and not having everything available everywhere in your code.

the scope is defined in the main two ways,

- Global Scope
- Local Scope

```
var greeting='Welcome to blog';
(function(){
  console.log(greeting); //Output: Welcome to blog
})();
```

consider above code greeting variable should be global scope, it can access inside the function,

```
(function(){

  var greeting = 'Welcome to blog';
  console.log(greeting); //Output: Welcome to blog
})();

console.log(greeting); //Output:Reference-Error greeting not defined
```

In the above code for local scope,

In scope level variables in JavaScript ES6 has updated hoisting variable let, var, const type check with that, In order to learn the scope, you need to understand hoisting also.

JavaScript scoping is quite the topic and there is much more to be learned you can refer *this site for more*.

8. Currying:

Explaining currying about with examples in JavaScript

Currying is a technique of evaluating the function with *multiple arguments*, into a sequence of function with a single argument.

In other words, when a function, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth until all arguments have been fulfilled.

consider below example code:

```
var add =   function (a){
                 return function(b){
                       return function(c){
                              return a+b+c;
                              }
                        }
                  }
console.log(add(2)(3)(4)); //output 9
console.log(add(3)(4)(5)); //output 12
```

this currying achieving through closures, so above program variables a,b private properties of the parent function

Why Useful Currying?

Mainly It helps to create a higher order function. It is extremely helpful in event handling.

How to convert an existing function to curried version?

The curry function does not exist in native JavaScript. But library like *lodash* makes it easier to convert a function to curried one.

9. Memoization:

Memoization is a programming technique which attempts to increase a function's performance by caching its previously computed results. Because JavaScript objects behave like associative arrays, they are ideal candidates to act as caches. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. However, if the data is not cached, then the function is executed, and the result is added to the cache.

The function is an integral part of the programming, they help to modularity and reusable to our code, as per above definition memoization is an optimizing our code

```
const memoizedAdd = () => {
        let cache = {};
        return (value) => {
            if (value in cache) {
                console.log('Fetching from cache');
                return cache[value];
            } else {
                console.log('Calculating result');
                let result = value + 10;
                cache[value] = result;
                return result;
            }
        }
    }

// returned function from memoizedAdd
const newAdd = memoizedAdd();
console.log(newAdd(9)); //output: 19 calculated
console.log(newAdd(9)); //output: 19 cached
```

from the above code, you can understand memoization.

10. Callback Function:

Callback function definition:

> A reference to executable code, or a piece of executable code, that is passed as an argument to other code.

From the above definition, the callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

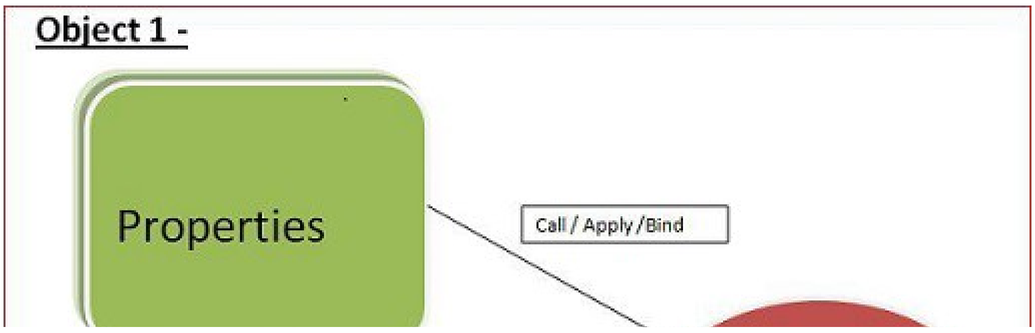
```
function greeting(name) {

console.log('Hello ' + name);
}

function processUserInput(callback) {
    //var name = prompt('Please enter your name.');
    name = 'raja';
    callback(name);
}
processUserInput(greeting); //output Hello Raja
```

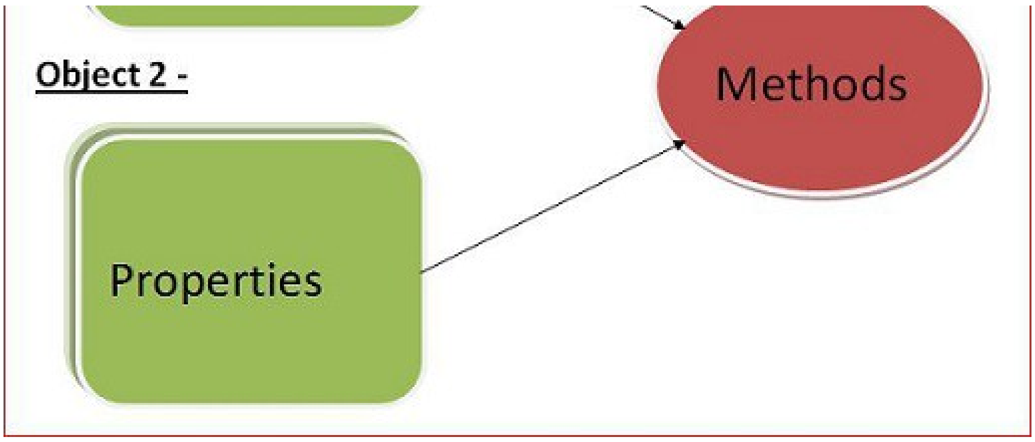
In this ab above program, function greeting passed as an argument to the processUserInput function, so I hope you now understood callback function in JavaScript.

11. The apply, call, and bind methods:

In traditionally JS have object, Properties, and methods, so each object has properties and methods.

In JavaScript, we can do some magic using call, apply, bind methods,





JavaScript Call/Apply/Bind Method

Consider the above image Object1 can have its own Properties and Object2 can have its own property, so we can write a common method for these object and use within that using call/apply/bind method. Hope you now you can understand why call/apply/bind method using.

let's look at the difference and code for the Call/Apply/Bind method

1.Call method:

consider below code, obj have the property of num, using call method we can bound obj to add function,

```
var obj={
   num : 2
}
var add = function(num2,num3,num4){
   return this.num + num2 + num3 + num4;
}
var arr=[3,4,5];

//Call method
console.log(add.call(obj,3,4,5));  //Output : 14

//Apply method
console.log(add.apply(obj,arr));   //Output : 14

//bind Method
var bound = add.bind(obj);
console.log(bound(3,4,5));         //output : 14
```

2.Apply Method

The same way Apply method also works but the only difference is using apply method the passing arguments could be an array, refer below code.

3.Bind Method:

bind method returns a method instance instead of result value, after that need to execute a bound method with arguments.

In the above Code simply explain how to use call/Apply/Bind method with an argument.

12. Polymorphism in JavaScript:

Polymorphism is one of the tenets of Object-Oriented Programming (OOP). It is the practice of designing objects to share behaviors and to be able to override shared behaviors with specific ones. Polymorphism takes advantage of inheritance in order to make this happen.

let's look at the sample code for an override a function in JavaScript

```
var employee = new Employee('raja');

//override function
//this method going to execute

Employee.prototype.getDetails = function() {
    return this.name.toUpperCase();
}

console.log(employee.getDetails()); //outPut: RAJA

//object and prototype function
function Employee(name) {
    this.name = name;
}

Employee.prototype.getDetails = function() {
    return this.name;
}
```

In the above simple program prototype-based method for an Employee constructor function has to override by another prototype function as return the Name as uppercase.

So we can override a function in different Scope, and also possible for method overloading, JS doesn’t have method overloading native but still, we can achieve.

Still few concepts are in oops, that are all are Method overloading, Abstraction, Inheritance in JavaScript.

13. Asynchronous Js :

In JavaScript Synchronous and asynchronous are code execution Pattern,

In JavaScript Code Execution done By two separate ways:

1. Browser JS Engine (popular V8 JS Engine)
2. NodeJs V8 Engine

Browser JS Engine parse Html file and executes JavaScript by three patterns,

1. synchronous
2. Asynchronous
3. defer

```
index.html

<script src='index.js'>           //default Synchronous

<script async src='index.js'>      //parse as Asynchronously

<script defer src='index.js'>      //parse as deferred
```

while browser JS Engine parsing HTML file if <Script > tag occurs means there will be blocking, so how its get execute JavaScript Code for that above three patterns.

1. If synchronous <script > tag occurs, JS engine will download the code and execute that code and after that only parsing the below HTML code, generally Synchronous is a blocking script execution.
2. If Asynchronous <script async> tag occurs, while downloading the code JS engine will parse the HTML and once If JS code gets downloaded pause the parsing and back into the JS Code Execution, generally Asynchronous is a Non-blocking script execution.

3. If Asynchronous <script defer> tag occurs, JS Engine will parse the all HTML code and after that only executes the JS Code,

NodeJS V8 Engine:

NodeJS V8 engine executes its JavaScript Code as single threaded based on Event loop? so need to know more about

Event Loop and NodeJS

Synchronous and Asynchronous Paradigm

14. Understand Promises :

Promise definition:

> ***The `Promise` object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.***

A promise represents the result of the asynchronous function. Promises can be used to avoid chaining of callbacks. In JavaScript,

so whenever JavaScript Code Execute as Asynchronously, need to handle an operation as one of the ways would be using promises.

Promises, have been around quite a while and are defined by a spec called Promise/A+. ES6 has adopted this spec for its Promise implementation; but there are other Promise libraries out there such as Q, Bluebird, RSVP, and others that adhere to this spec and offer other features on top of it.

A `Promise` is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

```
var promise1 = new Promise(function(resolve, reject) {
    isDbOperationCompleted = false;
    if (isDbOperationCompleted) {
        resolve('Completed');
    } else {
        reject('Not completed');
    }
});

promise1.then(function(result) {
    console.log(result); //Output : Completed
}).catch(function(error) {
    console.log(error); //if isDbOperationCompleted=FALSE
    //Output : Not Completed
})
```

consider above code for a sample promise assume like doing DB Operation as asynchronously, In that promise Object arguments as two function resolve and reject,

whenever we execute that promise using .then() and .catch() as callback function In order to get resolve or reject values

To learn more about *Promise*.

15. Async & Await:

Babel now supporting async/await out of the box, and ES2016 (or ES7) just around the corner, **async & await basically just syntactic sugar on top of Promises**, these two keywords alone should make writing asynchronous code in Node much more bearable.

In JavaScript Asynchronous pattern handled in various versions,

ES5 -> Callback

ES6 -> Promise

ES7 -> async & await

However, what a lot of people may have missed is that the entire foundation for async/await is **promises**. In fact, every *async* function you write will return a promise, and every single thing you *await* will ordinarily be a promise.

```
async function getUserDetail() {
    try {
        let users = await getUsers();
        return users[0].name;
    } catch (err) {
        return {
            name: 'default user'
        };
    }
}
```

Above code for a simple async & await,

Good, async/await is a great syntactic improvement for both nodejs and browser programmer. Comparing to Promises, it is a shortcut to reach the same destination. It helps the developer to implement functional programming in JavaScript, and increases the code readability, making JavaScript more enjoyable.

In order to understand about Async&Await, you must need to understand promises.

In Async & Await lot more stuff there, Just play around it.