**JS Topics**

**1. Scope in JavaScript:**

JavaScript has 3 types of scope:

* Block Scope
* Function Scope
* Global Scope

**Block Scope:**

Before ES6 2015, JS had only Global and Function Scope

ES6 introduced two important new JS keywords: let and const

These two keywords provide Block Scope in JS.

Variable declared inside a {} block cannot be accessed from outside the block:

exceptions being variable defined with var keyword

{

let x = 2;

var y = 3; // This will be available outside of block

z = 4; // This will be available outside the scope

}

// x cannot be used outside of the block

**Function Scope:**

Anything defined with in a function gets function scoped it won’t be available outside the function whatsoever.

Function myFunction() {

var carName = “Volvo”; // This will not be available outside of function

var carName2 = “BMW”; // This will not be available outside of function

var carName3 = “Toyota” // This will not be available outside of function

}

**Global Scope:**

Anything declared and defined globally is global scoped and will be available anywhere in the program.

**2. IIFE:**

Immediately invoked function expression.

As name suggests these functions invoke themselves:

Eg:

(function () {

/\* \*/

})()

(() => {

/\* \*/

})()

**3. Hoisting:**

Hoisting as the name suggests hoists (takes up) the declaration of functions, variables or classes to the top of their scope, prior to execution of the code.

Hoisting allows functions to be safely used in code or before they are declared.

Functions declared with “**function**” keyword gets hoisted.

Hoisting works with **variables** defined with var keywordtoo but it only hoists declarations not initializations. Until that point in the execution is reached the variable has its default initialization as “undefined”.

Hoisting works with variables defined with **let & const** variables as well but, unlike var, are not initialized with a default value. An exception will be thrown if a variable declared with let or const is read before it is initialized.

Class defined using a class declaration are hoisted, which means that JS has a reference to the class. However, the class is not initialized by default, so any code that uses it before the line in which it is executed will throw a ReferenceError.

**4. Closures:**

A closure is the combination of a function bundled together with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

**Practical use of closures:**

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data.

This has obvious parallels to OOP, where objects allow you to associate data with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

**5. Callbacks:**

A callback is a function passed as an argument to another function.

Callbacks really shine in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

**Asynchronous Functions:**

Asynchronous functions are those functions which do not return something right away and might take some time in doing that

**6. Callback Hell:**

When we want certain asynchronous functions to execute in a certain manner then usually programmer nests callbacks inside of callbacks which results in callback hell. So basically, nesting too many callbacks result in callback hell.

**7. Promises:**

A **Promise** is a JavaScript object that links producing code and consuming code.

“Producing code” is code that can take some time to execute.

“Consuming code” is code that must wait for the result.

It always will have one of the 3 values:

* Pending
* Fulfilled
* Rejected

**8. Async and await**

Async await is used to work with promises in asynchronous functions. It is basically syntactic sugar for promises. It is just a wrapper to restyle code and make promises easier to read and use.

Putting ‘async’ before a function means simply one thing that it is going to return a promise. Other values are wrapped in a resolved promise automatically.

Putting await keyword makes JavaScript wait until the promise settles and returns its result.

**9. Diff b/w == & ===**

== will do the type-Conversion before equating the value of 2 variables that means ‘2’ and 2 will be equal with ==. === is also known as strictly equal it will equate both the value and their types and will return the respective boolean value

**10. Diff b/w undefined, undeclared & null**

Null is pointing to nothing in memory.

Undefined is a variable that has not been assigned any value.

Undeclared is a variable that has not been properly declared using const, var or let.

**11. NaN**

The Global NaN property is a value representing Not a Number, you can use isNaN() method to check if a variable is not a number and return Nan

**12. Rest & spread operator**

Rest and spread operators are depicted with three dots (…) but they behave a bit differently.

**Rest Operator**: Helps us in collecting rest of the values into an array. For ex: Suppose I have a function which accepts a user’s details such as firstName, lastName, Gender, Age & DOB. And we want to separate out all the values other than firstName, lastName we can write something like function firstName, lastName, ...otherInfo and now this otherInfo will be the array of all the otherInformations which was fed into the function

**Spread Operator**: Helps us in spreading/expanding iterables into individual elements.

**13. Prototype**

Every object in JavaScript has a built-in property, which is called its prototype. The prototype is itself an object, so the prototype will have its own prototype, making what’s called a prototype chain. The chain ends when we reach a prototype that has null for its own prototype.

**14. Currying**

Currying simply means evaluating functions with multiple arguments and decomposing them into a sequence of functions with a single argument.

In other terms, currying is when a function – instead of taking all arguments at one time – takes the first one and returns a new function, which takes the third one, etc. until all arguments are completed.

**15. ES6 Features**

I. **let and const keywords.**

**ii. Arrow functions.**

**Iii. Multi-line strings:** Users can create multi-linestrings by using back-ticks(`).

**iv. Default parameters:** In ES6, users can provide the default values right in the signature of the functions. But, in ES5, OR || operator had to be used.

//ES6

let calculateArea = function (height =100, width = 50){

//logic

}

//ES5

var calculateArea = function (height, width){

height = height || 100,

width = width || 50

//logic

}

**v. Template literals:**  `My name is ${firstName} ${lastName}`

**vi. Destructuring Assignment:** Array and Object destructuring =>

let fruits = [‘apple’, ‘banana’]

let [a, b] = fruits // a => apple & b=> banana

let person = {name: ‘Shubham’, age: 22}

let {name, age} = person // name=> ‘Shubham’ & age => 22

**vii. Enhanced Object literals:** If the name of the property is same you can easily and quickly create the objects inside a pair of {} using the name of the property itself.

Function getMobile(manufacturer, model, year){

return{

manufacturer,

model,

year,

}

}

**viii. Promises:**  In ES6, Promises are used for asynchronous execution. We can use promise with the arrow function as demonstrated below.

const asyncCall = new Promise((resolve, reject) => {

//do something

resolve();

}).then(() => {

console.log(“Done”)

})

**ix. Classes:**

Previously, classes never existed in JS and ES6 introduced them which makes it simpler to create objects, implement inheritance by using the “extends” keyword and also reuse the code efficiently.

**x. Modules:** Previously, there was no native support for modules in JS. ES6 introduced a new feature called modules, in which each module is represented by separate “.js” file. We can use the “import” or “export” statement in a module to import or export variables, functions, classes or any other component form/to different files and modules.

**16. Arrow functions**

Arrow functions allow us to write shorter function syntax with omitting function and return keywords and if there are no parameters we can omit the brackets too.

**17. Value vs Reference**

In JS a good rule of thumb is every primitive (numbers, strings, boolean, symbols, null & undefined) are passed as values and objects (Objects, arrays, functions) are passed as references.

**18. De-structuring**

The destructuring assignment syntax is a javaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

**19. Array methods**

.length: length property returns the length of the string.

.toString(): returns elements of an array as a string comma separated.

.join(“*\_parameter\_*”): returns elements of an array as a string “\_parameter\_” separated.

.pop(): removes and returns the last element of the array.

.push(“*\_parameter\_*”): method adds the “\_parameter\_” at the end of the array and returns the new length of the array.

.shift(): method removes the first array element, returns it and shifts all other elements to a lower index.

.unshift(“*\_parameter\_*”): method adds the “\_parameter\_” at the beginning of the array and returns the new length of the array.

.concat(“\_*another\_array\_*”): method creates a new array by merging the “\_*another\_array\_*” with the array on which the concat was ran on. **Does not make changes to the original array on its own.**

.splice(“pos”, “num\_of\_elements”, “elem”, “elem2”, ...): method can be used to add new items and remove existing items to an array and returns removed items. The splice method returns an array with the deleted items.

First parameter “pos” defines the position where new elements should be added (spliced in)

Second parameter “num\_of\_elements” defines how many elements should be removed.

Third and rest of the parameters “elem” defines the elements to be added.

.slice(“start”, “end”): method slices out a piece of an array into a new array. **slice method does not remove any elements from the source array.**

First parameter “start” defines the starting position from which the new sliced array will be created.

Second parameter “end” defines the ending position upto and not including which the new sliced array will have values uptil.

**20. Generators**

In JavaScript, generators provide a new way to work with functions and iterators. Using a generator,

1. you can stop the execution of a function from anywhere inside the function.

2. and continue executing code from a halted position.

With yield you can pause the execution

.next() method provides a way to resume the flow. Wide case usecase is to generate iterators with generators.

**21. Data Types**

Mostly used data types are SNNUBO => string, number, null, undefined, boolean, object

Others are symbols and Big int.

**22. Weak Set**

**Set:**

Set is a javaScript object that only holds unique values.

**WeakSet:**

The WeakSet object lets you store weakly held *objects* in a collection.

The main differences to the Set object are:

I) WeakSet are collection of **objects only**. They cannot contain arbitrary values of any type as Set can.

II) WeakSet is *weak,* meaning references to objects in a WeakSet are held weakly. If no other references to an object stored in the WeakSet exist, those objects can be garbage collected.

**23. Weak Map**

**Map:**

Map is a javaScript object which hold key and value pairs and remember its original insertion order.

**WeakMap:**

The WeakMap is a collection of key/value pairs whose keys must be objects, with values of any arbitrary JS type, and which does not create strong references to its keys. That is, an object’s presence as a key in a WeakMap does not prevent the object from being garbage collected. Once an object used as a key has been collected, its corresponding values in any WeakMap become candidates for garbage collection as well – as long as they aren’t strongly referred to elsewhere.

**24. Temporal Deadzone**

In case of let and const variables, Basically, Temporal Dead Zone is a zone “before your variable is declared”, i.e. where you can not access the value of these variables, it will throw an error.

Because of this Temporal Deadzone it seems let and const variables are not being hoisted but that is not the case there is a period between entering scope and being declared where they cannot be accessed. This period is the temporal dead zone(TDZ).

**25. Memoization**

In computing, memoization or memoisation is an optimization techniques used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Memoization is a way to lower a function’s time cost in exchange for space cost.