**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 nest 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.

**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 and 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 sing 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**

**16. Arrow functions**

**17. Value vs Reference**

**18. De-structuring**

**19. Array methods**

**20. Generators**

**21. Data Types**

**22. Weak Set**

**23. Weak Map**

**24. Temporal Deadzone**

**25. Memoization**