Javascript Interview Questions

1. Why Javascript is a Dynamic Language?

Javascript is a Dynamic / Runtime language means that the data types of the variables can change during the runtime.

Example:

```
Var x = 0;
x++;
typeof(x)        // number
x = "Text";
typeof(x)        // String
x = true;
typeof(x)        // boolean
```

2- How Javascript determines datatypes?

Javascript determines the values by looking at it at the runtime, or depending on the value assigned.
We can check datatype using typeof(x).

3- What is typeof function?

Function that is used to determine datatypes in Javascript.

4. How to check datatypes in JS?

By using typeof() function.

5 What are the different datatypes in JS?
There are 8 datatypes and we can categorise them in:
1- Primitive
2- Objects

Primitive
1- String
2- number
3- null
4- undefined
5- boolean
6- big int → longer than limit of number ($2^{53}-1$). 1234n example
7- Symbol → unique & immutable primitive datatype.

Object
1- new Object();

6- Explain Undefined.
Undefined means the variable has been declared but no value is assigned to it.
Example:

```
var x = 10;
var y = "ABC";
var z;        // undefined
```

7- Explain Null.
Null indicates intentional absence of data. Null indicates its not zero, not empty string just absence of data.

8- Null vs undefined?

- Undefined means variable is created but value is not assigned.
- Null means we assigned value as Null which means absence of data. Usefull while using objects and you don't want some properties from it.

9- Explain Hoisting

It is a mechanism where variables and function declaration are moved to the top of the scope.

Example:

```
console.log(x);   // undefined
var x = 10;
```

This means declaration of variable x is moved to top or above console.log during runtime. It will print undefined because hoisting only moves declaration not initialization.

10- Are Javascript initialization hoisted?

Yes, but with undefined as value.

Explained in the question above ↑

11- What are global variables?

Global variables are accessible throughout the webpage or document.

12- What are the issues with Global variables?

Global variables makes application hard to debug as it can be manipulated by many functions.

13- What happens when we declare a variable without var keyword?

- Without var keyword variable becomes global. That means even if you declare it inside a function, it'll be accessible outside to all the document.

Example:

```
var x = 10;
function fun1() {
    y = 100;
    console.log(x);
}
fun1();
console.log(y);    // valid
```

14- What is Use Strict Keyword?

15- How to force developers to use var keyword?

"Use Strict" strictly checks if the variable is defined using "var" keyword or "let" keyword.

Example:

```
"use strict"
var x = 10;
    y = 5;    // error - y is not defined
```

16- How can we handle Global variable issues?

17- How can we avoid Global variables?

It is difficult to avoid global variables but we can organize it properly by doing two things:

1- Put global variable in a proper Namespace.
2- Module pattern using Closures and IIFE.

Example:

```
// use of namespace
var global = {};
global.connectionString = "Test";
global.logDir = "d:\Logs";

// use of closure
var myGlobal = (function () {

    var connectionString = "Test"
    function GetConnection () {
        return connectionString;
    }
    return {
        GetConnection
    }
})();
var str = myGlobal.GetConnection();
```

18- What are Closures?
19- Why do we need Closures?

Closures are functions inside function and it makes a normal function stateful.
The need for closures are solving global variable issues.
It creates self contained modules.
It creates self contained state.

__/__/06

Example:

```
function Simplefunction(){
    var x = 0;
    x++;
}
function Closurefunction(){
    var x = 0;
    function Increment(){
        x++;
    }
    return {
        Increment
    }
}

Simplefunction();   } Both will return x as 1;
Simplefunction();
var ref = Closurefunction();
ref.Increment();  // x = 1
ref.Increment();  // x = 2
```

Q- Explain IIFE?

Immediately invoked function expression.

It is an anonymous function which gets immediately invoked.

Example:

```
var x = 0;
(function(){
    var y = 10          // local to this function
    alert("Calling ...");
})();
```

__/__/07

21 What is the use of IIFE?
22. What is name collision in global scope?
23. IIFE vs Normal Function?

Name collision happens when same name function names and variable names are declared.

Example of Name collision:

```
function Init() {
    var x = 10;
}

var Init = 20;  // Init function become variable
Init();         // Error, Init is not a function
```

Here same name of function and variable is given.

Because IIFE does not have name, so there is no way you can get name collision.

```
(function () {
    // init
})();
```

A normal function has a name while IIFe does not have name.

Normal function could encounter name collision but IIFE will never face such issue.

Normal function can be declared once and called anytime but IIFE is immediately invoked.

24- What are design Patterns?

25- Which is the most used design pattern?

Design patterns are time tested architecture solutions.

So, to create a single instance we can use Singleton pattern, if the object creational process is complex, we can use factory pattern and so.

Module design or module revealing pattern is most used design pattern.

26- What is module pattern and revealing module pattern?

Module pattern has 2 big advantages:

- Self-contained independent components.
- Provides Encapsulation and Abstraction.

Module design pattern is a combination of

IIFE + Closures.

↑ namespace

↑ Encapsulation

Example:

```
var custNS = (function(){
    function customer(){ }
    function customerInvoices(){ }
}
return{
    customer
}
})();
```

var cust1 = new custNs.Customer();

27 Various ways of creating JS objects?

There are 4 ways -

1. Literal

```
var pat = {"name": " ", "address": " "};
pat.Admit = function () { }
```

2. Object.create() // creates instance on current object

```
var patnew = Object.create(pat);
patnew.age = 10;
```

3. Constructor

```
function Patient(){
    this.name = " ";
    this.address = " ";
    this.Admit = function (){
    }
}
var pat1 = new Patient();
```

4. Class ES6

```
class PatientClass {
    constructor (name, address){
        this.name = " ";
        this.address = " ";
    }
}
var p = new PatientClass();
```

28. How to achieve inheritance in Javascript?
29. What is prototype in Javascript?
30. Explain Prototype chaining?

Javascript uses object inheritance or prototypical inheritance. Inheritance is done using Prototype object.

Inheritance is object based not class based in Javascript.

Example :

```
function Employee(){
    this.Name = " ";
    this.DoWork = function(){
        alert("Working");
    }
    this.AddAttendance = function(){
        alert("Attence added");
    }
}

function Manager(){
    this.Cabin = " ";
    this.DoWork = function(){
        alert("Manages team");
    }
}

var emp = new Employee();
Manager.prototype = emp;
```

```
var man = new Manager();
man.Name = "Shiv";
man.AddAttendance();
man.DoWork();
```

Every Javascript Object has .prototype property inside it. We're just adding Employee features inside prototype property of Manager objects.

More on prototype

- Every Javascript object has a Prototype object.
- It is an inbuilt object provided by Javascript.
- It works as linked list and search for properties inside if it's not found outside in object.

Prototype chaining

Prototype chaining is a process where the property/ methods are first checked in the current object. If not found then it checks in the prototype object, if does not find in that it try checking the prototypes prototype object, until it gets the prototype object as null.

31 Let Keyword

- Introduced in ES6.
- let Keyword helps to create immediate block level local scope.

Example :

```
function Test() {

    function
    let x = 10;
    if (x = 10) {
    let x = 2;
        console.log(x);     // 2
    }
    console.log(x);         // 10
}
```

32- Are let variable hoisted?

Yes, they are hoisted but not initialized. So, if you try to access the variables you will get an uninitialized error.

33 Explain Temporal Dead Zone?

TDZ is a period or it's a state of a variable where variables are named in memory but they are not initialized with any value.

Example:

```
// This is TDZ
console.log(x); // Error
// This is TDZ
// This is TDZ
  console.log(x); // Error you're in TDZ
let x= 10; // End of TDZ
function Test(){
    // TDZ
    console.log(y); // Error
    // TDZ
    // TDZ
    let y=10; // End of TDZ
}
    Test();
```

Variable has name in memory due to hoisting but not the value.

34. Let vs Var

	Var	let
Scope	Scope to immediate function body.	Scope to the immediate enclosing block.
Initialization Value	Initialize with undefined	initialized with nothing.

Example:

```
function Test() {
    if(1=1) {
        let y = 10;
        var x = 10;
    }
    console.log(x);
    console.log(y);   // will give error
}

Test();
```

- When let used variables are declared in hoisting phase they do not contain any value so access will give error of access before initialized.
- In case of var hoisting declare as well as initialize the variable with undefined value.

35- What will be the output for these?

1.
```
var d = "10";
var d2 = "10";
console.log(d + d2);   // 1010
```

2.
```
var i = 10;
var i2 = 10;
console.log(i + i2);   // 20
```

3.
```
console.log(1 + 1 + "4");   // 24
```

36. What is a class in ES6?

A class in ES6 (ECMAScript 2015) is a blueprint for creating objects with properties and methods. It is syntatic sugar over Javascript existing prototype-based inheritance and doesn't introduce a new object-oriented inheritance model. Classes in ES6 make defining constructor functions and inheritance easier and more readable.

Example:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log("Hello from" + this.name);
    }
}

const person1 = new Person("Jd", 30);
person1.greet();
```

37. So with the class keyword, does it imply JS is an OOP language?

While the class keyword introduces more OOP syntax, Javascript is not purely a OOP language. JS supports multiple paradigms, including procedural, functional, and OOP programming. The class keyword is just syntactic sugar over Javascript's prototypal inheritance.

38. Difference between a class and normal function?

- A class in Javascript is typically used to define object constructors and methods. It uses the class keyword and has a constructor() to initialize object properties. Methods in classes are automatically added to the prototype.

Example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + "says hello");
  }
}
```

- A normal function is a function defined with function keyword. It can act as a constructor when invoked with new keyword, but it does not have special syntax like classes. Methods need to be manually added to the prototype.

Example:

```
function Animal(name){
    this.name = name;
}
Animal.prototype.speak = function() {
  console.log(this.name + "Says hello");
}
```

39. What is an Arrow function?

An arrow function is a concise way to write functions in ES6. It uses the ⇒ syntax and does not have its own this, arguments or prototype.

Example:

```
const add = (a,b) ⇒ a+b;
console.log(add(2,3)); // 5
```

- If one parameter, paranthesis are optional.
- If no parameter or multiple parameter, paranthesis required
- If it contains one expression then return keyword not required.
- If there are more than one statement, { } is required

One of the key feature of arrow function that they do not bind their ~~this~~ own this. Instead, they capture the this value from the surrounding context (lexical scoping). It makes them useful in call backs as they prevent this context from their outer function.

Example:

```
const obj = {
  name : "Alice",
  sayName : function(){
    setTimeout ( () ⇒ {
      console.log( this.name); // this refer to obj.
    },1000);
  }
};
```

40- Why do we need Arrow function?

- Concise Syntax.
- No binding of this
- Useful in callbacks or in situations where you don't need a separate this.

41- Difference between Normal and Arrow function.

- Arrow function have shorter syntax ⇒ , while normal functions use the function keyword.
- this binding : Arrow function inherits this value from their surrounding scope , whereas normal functions creates their own this when invoked.
- Arguments object : Arrow functions do not have their own arguments object, while normal functions do.

Example :

```
const obj = {
  arrow : () ⇒ console.log(this) // inherits this
  normal : function () { console.log(this); } // this refer to object itself
};

obj.arrow(); 'this' is not bound to obj
obj.normal(); 'this' is bound to obj
```

42- Does an Arrow function creates its own this?

No, they inherit this from the surrounding lexical context

43- Explain Synchronous execution?
Synchronous execution refers to executing code line by line, one task at a time. Each operation waits for the previous one to complete before it can proceed.

44- What is a call stack?
The call stack is a data structure that stores the execution context (functions) in Javascript. Whenever a function is called, its execution context is pushed onto the stack, and when the function completes, it is popped off the stack.

45- What is a blocking call?
A blocking call is a task that prevents further execution until it is completed. For example, reading a file synchronously blocks the execution until the file has been read completely.

46- How to avoid blocking calls?
To avoid blocking calls, use asynchronous methods (such as setTimeout, promises or async/await) or employ non-blocking I/O operations.

47- Explain Asynchronous execution?
Asynchronous execution allows multiple tasks to run independently without waiting for the previous one to complete. This enables the application to remain responsive.

48- Sync vs Async?

- Synchronous: executes one task at a time in sequence.
- Asynchronous: Allows multiple tasks to run without waiting, with callbacks or promises handling the completion of the asynchronous task.

49. How can we do Async calls?

1- Using promises

```
fetch('https://api.example.com/data').
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

2- Using async/await

```
const fetchData = async () => {
  try{
   const response = await fetch('https://api.example');
   Const data = await response.json();
    console.log(data);
  }catch(err){

    Console.error(err);
  }
};

fetchData();
```

3- Handling Async calls in React Components

```
const [data, setData] = useState(null);

useEffect(() => {
  const fetchData = async () => {
    try {
      const response = await fetch('https://api.example');
      const result = await response.json();
      setData(result);
    } catch (error) {
      console.error(error);
    }
  };

  fetchData();
}, []);
```

4- Using libraries like Axios

```
import axios from 'axios';
  const fetchData = async () => {
    try {
      const response = await axios.get('https://api.example');
      console.log(response.data);
    } catch (err) {
      console.error(err);
    }
  };
  fetchData();
```

5- Using React Query for Data fetching
@tanstack/react-query

```
import {useQuery} from '@tanstack/react-query';

Const MyComponent=() => {

const {data, isloading, error} = useQuery(['fetchData'], async ()=> {

  Const res = await fetch("https://api.example");
  return response.json();
});

if(isLoading) return <p> Loading </p>;

if(err) return <p> Error occured </p>;

return <div> {JSON.stringify (data)} </div>;
};
```

Q50- What is a Thread?

A thread refers to the execution context where the JavaScript code runs. JavaScript is a single-threaded language, meaning it has one thread of execution. Only one piece of JavaScript code is executed at a time, making it synchronous by default. However, it uses asynchronous mechanism to handle non-blocking operations.

Q51- Explain Multi-threading in Javascript?

- Javascript is not multi-threaded in the traditional sense. It runs in a single thread but can achieve asynchronous, non-blocking behaviour through mechanisms like callbacks, promises and async/await, alongside the event loop. While Javascript does not allow true multi-threading, web workers can be used to run tasks in parallel, but each worker runs in a separate thread and cannot directly access the main thread's variables.

Q52- Is Javascript Multi-threaded?

- No, Javascript is single-threaded

Q53- Then How Does setTimeout Run?

When setTimeout is used, it doesn't run immediately on the main thread. Instead, the following process happens -

1- The Web API (provided by the browser) handles the timer in the background.
2- After the specified time elapses, the callback function is placed in the callback queue.
3- The event loop check if the main thread (call stack) is free. Once it's free, it pulls the callback from the callback queue and pushes it to the call stack for execution.

This allows Javascript to handle setTimeout asynchronously, without blocking the main thread.

Q54. What is a Web API / Browser API?

- Web APIs (also known as Browser APIs) are APIs provided by the browser environment that allow JavaScript to interact with things outside its single-threaded runtime.

Examples:

- DOM manipulation (eg. document.querySelector)
- Timers (e.g setTimeout, setInterval)
- HTTP requests (e.g. fetch, XMLHttpRequest)
- Geolocation, LocalStorage and many others.

These APIs run it the browser, allowing asynchronous operations, which javascript interacts with.

Q55. What is an Event loop and Callback Queue?

- The event loop is a core concept in JavaScript's concurrency model. It handles asynchronous code by checking if the call stack is empty. If it is, it pushes any task from the callback queue to the call stack for execution. Here's how it works:

- The call stack contains functions that are being executed.
- The callback Queue holds asynchronous tasks (like timers or network requests) that are waiting to be executed.

-

The event loop ensures that JavaScript can handle asynchronous code even though it is single-threaded.

Q56- Event loop and callback code Example?

```
console.log("Start");

setTimeOut(() => {
console.log("Timeout callback");
}, 0);

Console.log("End")
```

1- Start printed first since it's synchronous.
2- End printed next, as the main thread executes the code sequentially.
3- The setTimeout callback is placed in the callback queue, and the event loop pushes it to the call stack after the main code finishes. Therefore "Timeout callback" is printed last even though the timeout was set to 0.