JAVASCRIPT IMPORTANT INTERVIEW QUESTIONS

1. How do you detect primitive or non-primitive value types in Javascript?
2. Explain the key features introduced in Javascript ES6
3. What are the differences between var, const & let in JavaScript?
4. What are arrow functions in Javascript?
5. What is hoisting in Javascript?
6. What is Strict Mode in Javascript?
7. What is NaN?
8. Is javascript a statically typed or a dynamically typed language?
9. What is difference between Null and Undefined
10. What is DOM?
11. What is BOM?
12. Explain about this keyword in Javascript with an example.
13. What is scope in Javascript?
14. What is closure in Javascript?
15. Explain call(), apply() and bind() methods in Javascript
16. What are prototypes in Javascript?
17. What are callback functions in Javascript and what is callback hell?
18. What is Temporal Dead Zone in Javascript?
19. What are promises in Javascript?
20. Explain rest parameter in Javascrip
21. What are generator functions in Javascript?
22. What is the difference between function declarations and function expressions
23. What is the difference between setTimeout, setImmediate and process.nextTick
24. Which symbol is used for comments in JavaScript?
25. What is an event bubbling in JavaScript?
26. Which is faster in JavaScript and ASP script?
27. What is negative infinity?
28. Is it possible to break JavaScript Code into several lines?
29. How many ways an HTML element can be accessed in JavaScript code?
30. What are undeclared and undefined variables?
31. What are undeclared and undefined variables?
32. What is the difference between innerHTML & innerText?
33. How to delete property-specific values?
34. hat is a prompt box?
35. What is the difference between ViewState and SessionState?
36. Does JavaScript support automatic type conversion?
37. What are all the looping structures in JavaScript ?
38. How can the style/class of an element be changed?
39. What is the ‘Strict’ mode in JavaScript and how can it be enabled?
40. How to convert the string of any base to integer in JavaScript?
41. Explain how to detect the operating system on the client machine?
42. What are the types of Pop up boxes available in JavaScript?
43. What is the difference between an alert box and a confirmation box?
44. What is the disadvantage of using innerHTML in JavaScript?

45. What is the use of void(0) ?
46. Why do we use the word “debugger” in javascript?
47. Difference between “ == “ and “ === “ operators.
48. Explain Implicit Type Coercion in javascript.
49. Explain passed by value and passed by reference.
50. Explain Higher Order Functions in javascript.
51. What is the difference between exec () and test () methods in javascript?
52. What are the types of errors in javascript?
53. What is recursion in a programming language?
54. What is the use of a constructor function in javascript?
55. Which method is used to retrieve a character from a certain index?
56. In JavaScript, how many different methods can you make an object?
57. What are classes in javascript?
58. Difference between Async/Await and Generators usage to achieve the same functionality.
59. What are the primitive data types in JavaScript?
60. What is the role of deferred scripts in JavaScript?
61. What has to be done in order to put Lexical Scoping into practice?
62. New features in ES6 version.
63. Is javascript a statically typed or a dynamically typed language?
64. What is closure in JavaScript?
65. What is Critical Rendering Path?
66. What are basic JavaScript array methods?
67. What is the rest parameter and spread operator?
68. What are observables?
69. What is microtask in JavaScript?
70. What Pure Functions in JavaScript?
71. What are the various statements in error handling?
72. What do you mean by strict mode in javascript and characteristics of javascript strict-mode?
73. What are the differences between cookie, local storage and session storage?
74. Difference between Debouncing and Throttling.

Programming Questions Link Is In Bio

1.**How do you detect primitive or non primitive value types in Javascript?**

In JavaScript, values are generally categorized as either primitive or non-primitive (also known as reference types). Primitive values include:

1. Number: Represents numeric values.
2. String: Represents textual data.
3. Boolean: Represents true or false.
4. Undefined: Represents an uninitialized variable or absence of a value.
5. Null: Represents the intentional absence of any object value.
6. Symbol: Represents a unique identifier.

Non-primitive values are objects, which include arrays, functions, and custom objects.

We can detect primitive or non primitive in Javascript in the following ways:

1.Using the typeof operator:

This operator returns a string indicating the type of a value.

Primitive types will return their corresponding strings (e.g., "number", "string", "boolean").

Non-primitive types will typically return "object" or "function"

```
Javascript
// Using the typeof operator:
let num = 10;
let str = "Hello";
let bool = true;
let obj = {};
let func = function() {};
console.log(typeof num); // Output: "number"
console.log(typeof str); // Output: "string"
console.log(typeof bool); // Output: "boolean"
console.log(typeof obj); // Output: "object"
console.log(typeof func); // Output: "function"
```

important note: typeof null returns "object" even though it's a primitive value.

2.Using the Object() constructor:

1. This constructor creates a new object wrapper for a value.
2. If a value is primitive, it will be equal to its object-wrapped version.
3. If a value is non-primitive, it won't be equal to its object-wrapped version.
4.

```
Javascript
 // Using the Object() constructor:
console.log(num === Object(num)); // Output: true (primitive)
console.log(obj === Object(obj)); // Output: false (non-primitive)
```

2.**Explain the key features introduced in Javascript ES6**

In ES6, JavaScript introduced these key features:

Arrow Functions:

Concise syntax for anonymous functions with lexical scoping.

Template Literals:

Enables multiline strings and variable inclusion for improved readability.

Destructuring Assignment:

Simplifies extraction of values from arrays or objects.

Enhanced Object Literals:

Introduces shorthand notation for defining object methods and dynamic property names.

Promises:

Streamlines asynchronous programming with a cleaner, structured approach

3.What are the differences between var, const & let in JavaScript?

Attribute	var	let	const
Scope	Functional scope	Block scope	Block scope
Update/ Re-declaration	Can be updated and re-declared within thescope	Can be updated but cannot be re declared within the scope	Cannot be updated or re declared within the scope
Declaration without Initialization	Can be declared without being initialized	Can be declared without being initialized	Cannot be declared without being initialized
Access without Initialization	Accessible without initialization (default: undefined)	Inaccessible without initialization (throws 'ReferenceError')	Inaccessible without initialization (throws 'ReferenceError')
Hoisting	Hoisted and initialized with a 'default' value	Hoisted but not initialized (error if accessed before declaration/ initialization)	Hoisted but not initialized (error if accessed before declaration/ initialization)

4.What are arrow functions in Javascript?

Arrow functions are a concise way to write anonymous function expressions in JavaScript. They were introduced in ECMAScript 6 (ES6) and are especially useful for short, single-expression functions.

Here's the basic syntax for an arrow function:

```
Javascript

const add = (a, b) => {

return a + b;

};
```

In this example, the arrow function takes add two parameters (a and b) and returns their sum. The => syntax is used to define the function, and the body of the function is enclosed in curly braces {} . If there's only one expression in the function body, you can omit the curly braces and the return keyword:

```
Javascript

const add = (a, b) => {

return a + b;

 }
```

Here is an example showing how both traditional function expression and arrow function to illustrate the difference in handle the this keyword.

Traditional Function Expression:

```
Javascript

// Define an object

let obj1 = {

value: 42,

 valueOfThis: function() {

return this.value; // 'this' refers to the object calling the function (obj1)

} };


// Call the method

console.log(obj1.valueOfThis()); // Output: 42
```

In this example, obj1.valueOfThis() returns the vlaue property of obj1, as this inside the function refers to the object obj1 .

Arrow Function:

```
Javascript

 // Define another object

let obj2 =

 { value: 84,

 valueOfThis: () => {

return this.value; // 'this' does not refer to obj2; it inherits from the parent scope (window in this case)

} };

// Call the method

console.log(obj2.valueOfThis()); // Output: undefined or an error (depending on the environment)
```

In the arrow function within obj2, this does not refer to obj2 . Instead, it inherits its value from the parent scope, which is the global object (window in a browser environment). Consequently, obj2.valueOfThis() returns undefined or may even throw an error, as this.value is not defined in the global scope.

5.What is hoisting in Javascript?

In JavaScript, hoisting is a phenomenon where variable and function declarations are conceptually moved to the top of their respective scopes, even if they're written later in the code. This behaviour applies to both global and local scopes.

Here are some examples to illustrate hoisting:

Example 1: Variable Hoisting

```
Javascript

console.log(myMessage); // Outputs "undefined", not an error

var myMessage = "Greetings!";
```

While myMessage appears declared after its use, it's hoisted to the top of the scope, allowing its reference (but not its initial value) before the actual declaration line.

Example 2: Function Hoisting

```
Javascript

sayHello(); // Outputs “Hello, world!”

function sayHello() {

console.log("Hello, world!"); }
```

Even though sayHello is defined after its call, JavaScript acts as if it were declared at the beginning of the scope, enabling its execution.

Example 3: Hoisting within Local Scopes

```
Javascript

function performTask() {

 result = 100; // Hoisted within the function

 console.log(result); // Outputs 100

var result;

 }
```

performTask();

Hoisting also occurs within local scopes, like functions. Here, result is hoisted to the top of the perdormTask function, allowing its use before its explicit declaration.

Key Points:

Only declarations are hoisted, not initializations. The example with console.log(x) ; demonstrates this, as x is declared but not initialised before its use, resulting in undefined.

Strict mode enforces declaration: Using “use strict”; at the beginning of your code prevents using variables before they're declared, helping avoid potential hoisting-related issues.

6.What is Strict Mode in Javascript?

Strict Mode is a feature that allows you to place a program, or a function, in a “strict” operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal expression "use strict"instructs the browser to use the javascript code in the Strict mode.

Strict mode helps in writing "secure" JavaScript by notifying "bad syntax" into real errors.

The strict mode is declared by adding "use strict"; to the beginning of a script or a function. If declared at the beginning of a script, it has global scope.

Example:

```
Javascript

'use strict';

 x = 15; // ReferenceError: x is not defined

function strict_function() {

'use strict';

x = 'Test message';
```

```
console.log(x); }

strict_function(); // ReferenceError: x is not defined
```

7.What is NaN?

The NaN property in JavaScript represents a value that is "Not-a Number," indicating an illegal or undefined numeric value. When checking the type of NaN using the typeof operator, it returns "Number."

To determine if a value is NaN, the isNaN() function is employed. It converts the given value to a Number type and then checks if it equals NaN.

Example:

isNaN("Hello"); // Returns true, as "Hello" cannot be converted to a valid number

isNaN(NaN); // Returns true, as NaN is, by definition, Not-a Number

isNaN("123ABC"); // Returns true, as "123ABC" cannot be converted to a valid number

isNaN(undefined); // Returns true, as undefined cannot be converted to a valid number

isNaN(456); // Returns false, as 456 is a valid numeric value

isNaN(true); // Returns false, as true is converted to 1, a valid number

isNaN(false); // Returns false, as false is converted to 0, a valid number

isNaN(null); // Returns false, as null is converted to 0, a valid number

8.Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, variable types are determined at runtime, allowing a variable to hold values of any type without explicit type declarations. This flexibility can make coding more convenient but may also lead to runtime errors if types are not handled appropriately.

JavaScript, being dynamically typed, allows variables to change types during execution and accommodates a wide range of data types without explicit type annotations.

9.What is difference between Null and Undefined

Feature	Null	Undefined
Type	Object	Undefined
Definition	Assignment value indicating no object	Variable declared but not yet assigned a value
Nature	Primitive value representing null/empty	Primitive value used when a variable is unassigned

Representat ion	Absence of a value for a variable	Indicates the absence of the variable itself
Conversion in Operations	Converted to zero (0)	Converted to NaN during primitive operations

10.What is DOM?

DOM stands for Document Object Model, serving as a programming interface for web documents.

1. Tree Structure: It represents the document as a tree, with the document object at the top and elements, attributes, and text forming the branches.
2. Objects: Every document component (element, attribute, text) is an object in the DOM, allowing dynamic manipulation through programming languages like JavaScript.
3. Dynamic Interaction: Enables real-time updates and interactions on web pages by modifying content and structure in response to user actions.
4. Programming Interface: Provides a standardized way to interact with a web document, accessible and modifiable using scripts.
5. Cross-platform and Language-Agnostic: Not bound to a specific language and works across various web browsers, ensuring a consistent approach to document manipulation.
6. Browser Implementation: While browsers have their own DOM implementations, they follow standards set by the World Wide Web Consortium (W3C), ensuring uniformity in document representation and manipulation

11.What is BOM?

BOM (Browser Object Model) is a programming interface extending beyond DOM, providing control over browser-related features.

1. Window Object: Core BOM element representing the browser window, with properties and methods for browser control.
2. Navigator, Location, History, Screen Objects: Components handling browser information, URL navigation, session history, and screen details.
3. Document Object: Accessible through BOM, allowing interaction with the structure of web pages.
4. Timers: Functions like setTimeout and setInterval for scheduling code execution.
5. Client Object: Represents user device information, aiding in responsive web design.
6. Event Object: Manages events triggered by user actions or browser events.

12.Explain about this keyword in Javascript with an example.

In JavaScript, the this keyword is a special variable that is automatically defined in the scope of every function. Its value depends on how the function is invoked. The this keyword is used to refer to the object that is the current context of the function or, more simply, the object that the function is a method of.

Here are some common scenarios that affect the value of :

Global Context:

When this is used outside of any function or method, it refers to the global object (in a browser environment, it usually refers to window).

```
Javascript
console.log(this); // refers to the global object
 (e.g., window in a browser)
```

Method Invocation:

When a function is a method of an object, this refers to that object.

```
Javascript
const myObject = {
myMethod: function()
{
console.log(this); // refers to myObject
 } }; myObject.myMethod();
```

Constructor Function:

When a function is used as a constructor with the new keyword, this refers to the newly created instance of the object.

```
Javascript
function MyClass() {
this.property = 'some value';
} const myInstance = new MyClass();
console.log(myInstance.property); // 'some value
```

13.What is scope in Javascript?

In JavaScript, the term "scope" refers to the context in which variables and functions are declared and accessed. It defines the visibility and accessibility of these variables and functions within the code. Understanding scope is crucial for managing the lifecycle and behaviour of variables and functions in a program.

There are two main types of scope in JavaScript: global scope and local scope.

Global Scope:

Variables declared outside of any function or block have global scope.

Global variables are accessible throughout the entire code, including within functions.

```
Javascript
var globalVar = "I am global";
function exampleFunction() {
console.log(globalVar); // Accessible inside the function
}
exampleFunction();
console.log(globalVar); // Accessible outside the function
```

Local Scope:

Variables declared inside a function or block have local scope.

Local variables are only accessible within the function or block where they are declared.

```
Javascript
function exampleFunction() {
 var localVar = "I am local";
console.log(localVar); // Accessible inside the function
 }
exampleFunction();
// console.log(localVar); // This would result in an error because localVar is not accessible outside the function
```

Scope Chain:

The scope chain refers to the hierarchy of scopes in a program. When a variable or function is referenced, JavaScript looks for it in the current scope and then traverses up the scope chain until it finds the variable or reaches the global scope

```
Javascript
var globalVar = 42;
function mainFunction(){
var localVar1 = 777;
var innerFunction1 = function(){
console.log(localVar1); //Accesses localVar1 inside
innerFunction1, outputs 777
}
```

```
var innerFunction2 = function(){
console.log(globalVar); //Accesses globalVar inside
innerFunction2, outputs 42
}
innerFunction1();
 innerFunction2();
 }
mainFunction();
```

14.What is closure in Javascript?

In JavaScript, a closure is a function along with its lexical scope, which allows it to access variables from its outer (enclosing) scope even after that scope has finished executing. A closure allows a function to remember and access variables from the environment in which it was created, even if the function is executed in a different scope.

Here's an example to illustrate closures in JavaScript:

```
Javascript


function outerFunction() {
//outer function scope
 let outerVariable = 10;
function innerFunction() {
// inner function scope
 let innerVariable = 5;
//Accessing both inner and outer variables
console.log("Inner Variable:", innerVariable);
console.log("Outer Variable:", outerVariable);
}
// Returning the inner function, creating a closure
return innerFunction;
}
```

```
// calling outerFunction returns innerFunction which is now a clousre
 let closureFunction = outerFunction();
//Executing the colosure function
closureFunction();
```

outerFunction defines an outer variable (outerVariable) and an inner function (innerFunction).

innerFunction has access to the variables of its outer function(outerVariable)

outerFunction returns innerFunction, creating a closure

The returned closureFunction retains access to the outerVariable even after outerFunction has finished executing

Calling closureFunction() logs both the inner and outer variables to the console.

15.Explain call(), apply() and bind() methods in Javascript.

In JavaScript, the call, apply, and bind methods are used to manipulate how a function is invoked and set the value of this within the function.

call method:

The method call is used to invoke a function with a specified this value and arguments provided individually.

```
Javascript
function sayHello(greeting)
{
console.log(greeting + ' ' + this.name);
}
const person = { name: 'John' };
sayHello.call(person, 'Hello'); // Outputs: Hello John
```

Here, call is used to invoke the sayHello function with person as the this value, and 'Hello' as an argument.

apply method:

The apply method is similar to call , but it accepts arguments as an array.

```
Javascript
```

```
function sayHello(greeting) {
console.log(greeting + ' ' + this.name);
}
const person = { name: 'John' };
sayHello.apply(person, ['Hello']); // Outputs: Hello John
```

In this example,apply is used to achieve the same result as call , but the arguments are provided as an array.

bind method:

The bind method creates a new function with a specified this value and, optionally, initial arguments.

```
Javascript
function sayHello(greeting) {
 console.log(greeting + ' ' + this.name);
 }
 const person = { name: 'John' };
 const sayHelloToJohn = sayHello.bind(person);
sayHelloToJohn('Hello'); // Outputs: Hello John
```

Here, is used to create a new function (sayHelloToJohn) where this is permanently set to person. When calling , it's as if you're calling sayHello with person as this .

These methods are especially useful when dealing with functions that are part of objects or classes, and you want to explicitly set the context (this) for their execution.

16.What are prototypes in Javascript?

Every object in JavaScript has a prototype, which acts as a blueprint for shared properties and methods.

When you try to access a property or method on an object, JavaScript first checks the object itself.

If it's not found, it looks up the prototype chain, following a linked list of prototypes until it finds what it's looking for, or reaches the end (null).

Example:

```
Javascript
function Person(name) {
this.name = name;
}
// Add a method to the prototype, shared by all Person objects:
Person.prototype.greet = function() {
console.log("Hello, my name is " + this.name);
};
```

--

```
Javascript
// Create two Person objects
 const person1 = new Person("Alice");
 const person2 = new Person("Bob");
// Both objects can access the greet method from the prototype:
person1.greet(); // Output: "Hello, my name is Alice"
 person2.greet(); // Output: "Hello, my name is Bob"
```

The Preson function acts as a constructor to create objects with a name property.

The greet method is added to the Preson.prototype , meaning it's shared by all instances created from Person.

When person1.greet() is called, JavaScript finds the greet method on the prototype, so it can be used even though it wasn't defined directly on person1.

17.What are callback functions in Javascript and what is callback hell?

In JavaScript, a callback is a function that is passed as an argument to another function and is executed after the completion of some asynchronous operation or at a specified time. Callbacks are commonly used in scenarios like handling asynchronous tasks, event handling, and other situations where the order of execution is not guaranteed.

```
 Javascript
 function customGreeting(name) {
console.log("Welcome, " + name + "! How can we assist you today?");
}
```

```
 function outerFunction(callback) {
 let name = prompt("Please enter your name.");
callback(name);
}
outerFunction(customGreeting);
```

In this example, the customGreeting function is the callback function passed to outerFunction

Callback hell (or "pyramid of doom") is a situation in which multiple nested callbacks make the code difficult to read and maintain. This often occurs when dealing with asynchronous operations, such as making multiple API calls or handling multiple events.

Here's an example of callback hell:

```
Javascript
 getUser(function(user) {
getProfile(user.id, function(profile) {
 getPosts(user.id, function(posts) {
displayUserProfile(user, profile, posts, function() {
// More nested callbacks...
 }); }); }); });
```

In this example, we have nested callbacks for getting a user, fetching their profile, retrieving their posts, and finally displaying the user profile. As more asynchronous operations are added, the code becomes more difficult to read and maintain.

To address callback hell, developers often use techniques like Promises or async/await in modern JavaScript to make code more readable and manageable.

18.What is Temporal Dead Zone in Javascript?

The Temporal Dead Zone is a phenomenon in JavaScript associated with the use of the let and const keywords, unlike the var keyword. In ECMAScript 6, attempting to access a let or const variable before it is declared within its scope results in a ReferenceError. The term "temporal dead zone" refers to the timeframe during which this occurs, spanning from the creation of the variable's binding to its actual declaration.

Let's illustrate this behaviour with an example:

```
Javascript
function exampleMethod() {
console.log(value1); // Outputs: undefined
```

```
console.log(value2); // Throws a ReferenceError

var value1 = 1;

 let value2 = 2;

}
```

In this example, attempting to access value2 before its declaration causes a ReferenceError due to the temporal dead zone, while accessing value1 results in an output of undefined.

19.What are promises in Javascript?

JavaScript Promises offer a streamlined approach to managing asynchronous operations, mitigating the callback hell problem encountered with events and traditional callback functions. Before Promises, working with callbacks often led to code that was hard to manage due to nested structures. Promises serve as a cleaner solution for handling asynchronous tasks in JavaScript.

Here's the syntax for creating a Promise:

Javascript

```
let promise = new Promise(function(resolve, reject) {

// Perform asynchronous operations

});
```

The Promise constructor takes a single callback function as its argument, which, in turn, accepts two parameters: resolve and reject. The operations inside this callback determine whether the Promise is fulfilled by calling resolve or rejected by calling .

A Promise can exist in one of four states:

fulfilled: The action related to the promise succeeded.

rejected: The action related to the promise failed.

pending: The promise is still awaiting fulfilment or rejection.

settled: The promise has been either fulfilled or rejected.

20.Explain rest parameter in Javascript

In JavaScript, the rest parameter is a feature that allows you to represent an indefinite number of arguments as an array. It is denoted by three dots (...) followed by the parameter name. The rest parameter collects all the remaining arguments passed to a function into a single array.

Here's a simple example to illustrate the concept:

```
Javascript
```

```
function sum(...numbers) {
return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

In this example, the sum function accepts any number of arguments. The rest parameter ...numbers collects all the arguments into an array called numbers. The function then uses the reduce method to sum up all the numbers in the array.

It's important to note that the rest parameter must be the last parameter in the function declaration. For example, this is valid:

```
Javascript
function example(firstParam, ...restParams) {
// code here
 }
```

But this is not:

```
Javascript
function invalidExample(...restParams, lastParam) {
// code here
}
```

21.What are generator functions in Javascript?

In JavaScript, generator functions are a special kind of function that allows you to control the execution flow and pause/resume it at certain points. Generator functions are defined using the function* syntax and use the yield keyword to produce a sequence of values. When a generator function is called, it returns an iterator called a generator.

Here's a simple example of a generator function:

```
Javascript
function* simpleGenerator() {
yield 1;
yield 2;
yield 3;
} // Creating a generator
const generator = simpleGenerator();
```

```
// Using the generator to get values
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
console.log(generator.next()); // { value: undefined, done: true }
```

In this example:

The function* simpleGenerator() syntax defines a generator function.

The keyword is used to produce values. Each time yield is encountered, the generator pauses its execution, and the yielded value is returned to the caller along with done: false. The generator can be resumed later.

The generator.next() method is used to advance the generator's execution. It returns an object with two properties: value (the yielded value) and done (a boolean indicating whether the generator has finished).

Generators are useful for lazy evaluation, asynchronous programming, and creating iterable sequences.

22.What is the difference between function declarations and function expressions?

Function Declaration:

A function declaration is a statement that defines a function and hoists it to the top of the current scope.

It starts with the function keyword, followed by the function name, parameters (enclosed in parentheses), and the function body.

Example:

Javascript

```
function add(a, b) {
return a + b;
}
```

Function declarations can be called before they are declared in the code because of hoisting.

Function Expression:

- A function expression is an assignment where a function is defined as part of an expression.
- It does not get hoisted in the same way as function declarations.

Example:

```
Javascript

var add = function(a, b) {

 return a + b;

};
```

In this example, add is a variable that holds an anonymous function.

Function declarations are hoisted, while function expressions are not hoisted in the same way. If you try to call a function expression before its definition, you'll get an error.

Function expressions are often used in cases where you need to assign a function to a variable or pass it as an argument to another function

23.What is the difference between setTimeout, setImmediate and process.nextTick?

setTimeout, setImmediate, and process.nextTick are all functions in Node.js that allow you to schedule the execution of a callback function, but they have some differences in terms of when the callback will be executed.

1.setTimeout:

Schedules the callback to be executed after a specified delay (in milliseconds).

The callback is added to the event queue, and it will be executed after the specified delay, but the exact timing is not guaranteed.

```
Javascript

setTimeout(() => {

console.log('This will be executed after 1000 milliseconds');

}, 1000);
```

2. setImmediate:

Schedules the callback to be executed in the next iteration of the event loop.

It's often used when you want the callback to be executed immediately after the current event loop cycle

```
Javascript

setImmediate(() => {

console.log('This will be executed in the next iteration of the event loop');

 });
```

3.process.nextTick:

Executes the callback after the current event loop cycle, but before the event loop continues processing other I/O events.

It is often used when you want to execute a callback after the current operation but before I/O events.

```
Javascript

process.nextTick(() => {

console.log('This will be executed in the next event loop cycle');

});
```

24.Which symbol is used for comments in JavaScript?

Comments prevent the execution of statements. Comments are ignored while the compiler executes the code.

There are two type of symbols to represent comments in JavaScript:

1) Double slash: It is known as a single-line comment.

// Single line comment

2) Slash with Asterisk: It is known as a multi-line comment.

/* Multi-line comments ... */

25.What is an event bubbling in JavaScript?

Ans : Consider a situation an element is present inside another element and both of them handle an event. When an event occurs in bubbling, the innermost element handles the event first, then the outer, and so on.

26.Which is faster in JavaScript and ASP script?

Ans : JavaScript is faster compared to ASP Script. JavaScript is a client-side scripting language and does not depend on the server to execute.

The ASP script is a server-side scripting language always dependable on the server.

27.What is negative infinity?

Ans : The negative infinity is a constant value represents the lowest available value. It means that no other number is lesser than this value.

It can be generate using a self-made function or by an arithmetic operation. JavaScript shows the NEGATIVE_INFINITY value as -Infinity.

28.Is it possible to break JavaScript Code into several lines?

Ans : Yes, it is possible to break the JavaScript code into several lines in a string statement.

It can be broken by using the backslash '\'.

For example:

document.write("A Online Computer Science Portal\ for Topper World")

29.How many ways an HTML element can be accessed in JavaScript code?

Ans : There are four possible ways to access HTML elements in JavaScript which are:

1) getElementById() Method: It is used to get the element by its id name.

2) getElementsByClass() Method: It is used to get all the elements that have the given classname.

3) getElementsByTagName() Method: It is used to get all the elements that have the given tag name.

4) querySelector() Method: This function takes CSS style selector and returns the first selected element.

30.What are undeclared and undefined variables?

Ans :

➢ Undefined: It occurs when a variable is declare not not assign any value. Undefined is not a keyword.

➢ Undeclared: It occurs when we try to access any variable which is not initialize or declare earlier using the var or const keyword.

If we use 'typeof' operator to get the value of an undeclare variable, we will face the runtime error with the return value as " undefined". The scope of the undeclare variables is always global.

32.What is the difference between innerHTML & innerText?

Ans : The innerText property sets or returns the text content as plain text of the specified node, and all its descendants whereas the innerHTML property sets or returns the plain text or HTML contents in the elements.

Unlike innerText, inner HTML lets you work with HTML rich text and doesn’t automatically encode and decode text.

33.How to delete property-specific values?

Ans : The delete keyword deletes the whole property and all the values at once like

```
// Define an object
 var person = {
name: 'John',
age: 30,
city: 'New York'
 };
console.log('Before deletion:', person);
// Delete the 'city' property
delete person.city;
console.log('After deletion:', person);
```

Output:

```
Before deletion: { name: 'John', age: 30, city: 'New York' }
 After deletion: { name: 'John', age: 30 }
```

34.What is a prompt box?

Ans : The prompt box is a dialog box with an optional message prompting the user to input some text. It is often used if the user wants to input a value before entering a page. It returns a string containing the text entered by the user, or null.

35.What is the difference between ViewState and SessionState?

Ans : ➢ ViewState: It is specific to a single page in a session. ➢ SessionState: It is user specific that can access all the data on the web pages.

36.Does JavaScript support automatic type conversion?

Ans : Yes, JavaScript supports automatic type conversion.

37.What are all the looping structures in JavaScript ?

Ans :

1) while loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

2)for loop: A for loop provides a concise way of writing the loop structure. Unlike a while loop, for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

3) do while: A do-while loop is similar to while loop with the only difference that it checks the condition after executing the statements, and therefore is an example of Exit Control Loop.

38.How can the style/class of an element be changed?

Ans : To change the style/class of an element there are two possible ways. We use document.getElementByID method

```
document.getElementById("myText").style.fontSize = "16px;
```

```
document.getElementById("myText").className = "class";
```

39. What is the 'Strict' mode in JavaScript and how can it be enabled?

Ans : Strict Mode is a new feature in ECMAScript 5 that allows you to place a program or a function in a "strict" operating context. This strict context prevents certain actions from being taken and throws more exceptions. The statement "use strict" instructs the browser to use the Strict mode, which is a reduced and safer feature set of JavaScript.

40.How to convert the string of any base to integer in JavaScript?

Ans : In JavaScript, parseInt() function is used to convert the string to an integer. This function returns an integer of base which is specified in second argument of parseInt() function. The parseInt() function returns Nan (not a number) when the string doesn't contain number.

41. Explain how to detect the operating system on the client machine?

Ans : To detect the operating system on the client machine, one can simply use navigator.appVersion or navigator.userAgent property. The Navigator appVersion property is

a read-only property and it returns the string that represents the version information of the browser.

42.What are the types of Pop up boxes available in JavaScript?

There are three types of pop boxes available in JavaScript.

1. Alert

2. Confirm

3. Prompt

43. What is the difference between an alert box and a confirmation box?

Ans : An alert box will display only one button which is the OK button. It is used to inform the user about the agreement has to agree.

But a Confirmation box displays two buttons OK and cancel, where the user can decide to agree or not.

44. What is the disadvantage of using innerHTML in JavaScript?

Ans : There are lots of disadvantages of using the innerHTML in JavaScript as the content will replace everywhere. If you use += like "innerHTML = innerHTML + 'html'" still the old content is replaced by HTML. It preserves event handlers attached to any DOM elements.

45.What is the use of void(0) ?

Ans : The void(0) is used to call another method without refreshing the page during the calling time parameter "zero" will be passed.

46. Why do we use the word "debugger" in javascript?

The debugger for the browser must be activated in order to debug the code. Built-in debuggers may be switched on and off, requiring the user to report faults. The remaining section of the code should stop execution before moving on to the next line while debugging.

47.Difference between " == " and " === " operators

Both are comparison operators. The difference between both the operators is that "==" is used to compare values whereas, " === " is used to compare both values and types.

Example:

```
var x = 2;

var y = "2";

(x == y) // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"
```

48.Explain Implicit Type Coercion in javascript.

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

String coercion

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

```
var x = 3;

var y = "3";

 x + y // Returns "33"
```

Example 2:

```
var x = 24;

var y = "Hello";

x + y // Returns "24Hello";
```

Note - ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string:

```
var name = "Vivek";

 var surname = " Bisht";

 name + surname // Returns "Vivek Bisht"
```

Let’s understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types (one being a number type and the other being a string type), it converts the number type to the string type and then performs the operation. Since a er conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

Note - Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place.

```
var x = 3;

Var y = "3";

x - y //Returns 0 since the variable y (string type) is converted to a number type
```

Boolean Coercion

Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.

Truthy values are those which will be converted (coerced) to true. Falsy values are those which will be converted to false.

All values except false, 0, 0n, -0, “”, null, undefined, and NaN are truthy values.

If statements:

```
Example:

var x = 0;

var y = 23;

if(x) { console.log(x) } // The code inside this block will not run since the value of x

if(y) { console.log(y) } // The code inside this block will run since the value of y
```

Logical operators:

Logical operators in javascript, unlike operators in other programming languages, do not return true or false. They always return one of the operands.

OR (| |) operator - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

AND (&&) operator - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

```
Example:

var x = 220;

var y = "Hello";

var z = undefined;

x | | y // Returns 220 since the first value is truthy

x | | z // Returns 220 since the first value is truthy
```

```
x && y // Returns "Hello" since both the values are truthy
y && z // Returns undefined since the second value is falsy
if( x && y ){
console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)
 }
if( x || z ){ console.log("Code runs"); // This block runs because x || y returns 220(Truthy)
}
```

Equality Coercion

Equality coercion takes place when using ' == ' operator. As we have stated before

The ' == ' operator compares values and not types.

While the above statement is a simple way to explain == operator, it's not completely true

The reality is that while using the '==' operator, coercion takes place.

The '==' operator, converts both the operands to the same type and then compares them.

Example:

```
var a = 12;
var b = "12";
a == b // Returns true
```

Coercion does not take place when using the '===' operator. Both operands are not converted to the same type in the case of '===' operator.

Example:

```
 var a = 226;
var b = "226";
a === b // Returns false
```

49.Explain passed by value and passed by reference.

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

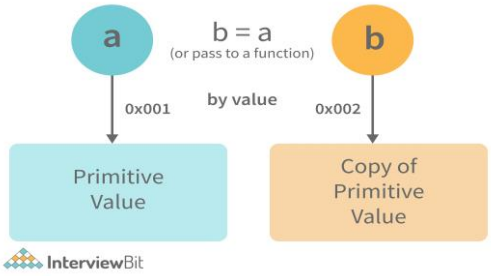
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

```
var x = 2;
```

In the above example, we created a variable x and assigned it a value of “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non-primitive data types,

Assign operator dealing with primitive types




```
var y = 234;

var z = y;
```

In the above example, the assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead, it is pointing to a new location in the memory.

var y = #8454; // y pointing to address of the value 234

var z = y;

var z = #5411; // z pointing to a completely new address of the value 234

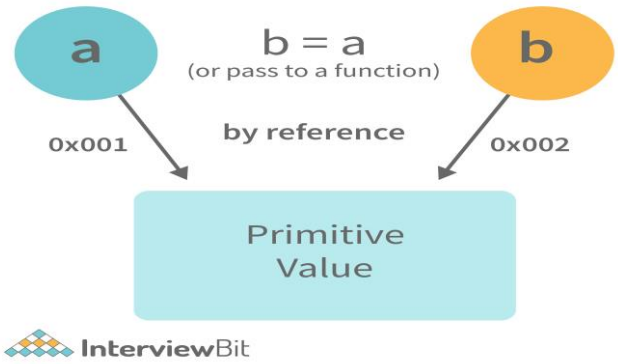
// Changing the value of y

y = 23;

console.log(z); // Returns 234,

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

Assign operator dealing with non-primitive types:




```
var obj = { name: "Vivek", surname: "Bisht" };
 var obj2 = obj;
```

In the above example, the assign operator directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2

```
var obj = #8711; // obj pointing to address of { name: "Vivek", surname: "Bisht" }
var obj2 = obj;
 var obj2 = #8711; // obj2 pointing to the same address
 // changing the value of obj1
obj1.name = "Akki";
console.log(obj2);
// Returns {name:"Akki", surname:"Bisht"}
```

From the above example, we can see that while passing non-primitive data types, the assign operator directly passes the address (reference). Therefore, non-primitive data types are always passed by reference.

50.Explain Higher Order Functions in javascript.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Higher-order functions are a result of functions being first-class citizens in javascript.

Examples of higher-order functions:

```
function higherOrder(fn)
{
fn();
```

```
}
higherOrder(function() { console.log("Hello world") });
```

```
function higherOrder2() {
return function() {
return "Do something";
} }
 var x = higherOrder2();
x() // Returns "Do something"
```

51.What is the difference between exec () and test () methods in javascript?

- test () and exec () are RegExp expression methods used in javascript.
- We'll use exec () to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
- We will use a test () to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'.

52.What are the types of errors in javascript?

There are two types of errors in javascript

1. Syntax error: Syntax errors are mistakes or spelling problems in the code that cause the program to not execute at all or to stop running halfway through. Error messages are usually supplied as well.

2. Logical error: Reasoning mistakes occur when the syntax is proper but the logic or program is incorrect. The application executes without problems in this case. However, the output findings are inaccurate. These are sometimes more difficult to correct than syntax issues since these applications do not display error signals for logic faults.

53.What is recursion in a programming language?

Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it arrives at a result.

```
function add(number) {
if (number <= 0) {
 return 0; }
 else {
```

```
return number + add(number - 1);
 } }
add(3) => 3 + add(2)
          3 + 2 + add(1)
          3 + 2 + 1 + add(0)
        3 + 2 + 1 + 0 = 6
```

Example of a recursive function:

The following function calculates the sum of all the elements in an array by using recursion:

```
function computeSum(arr){
if(arr.length === 1){
 return arr[0];
 }
else{
return arr.pop() + computeSum(arr);
 } }
 computeSum([7, 8, 9, 99]); // Returns 123
```

54.What is the use of a constructor function in javascript?

Constructor functions are used to create objects in javascript.

When do we use constructor functions?

If we want to create multiple objects having similar properties and methods, constructor functions are used.

Note- The name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.

Example:

```
 function Person(name,age,gender){
this.name = name;
this.age = age;
this.gender = gender
; }
var person1 = new Person("Vivek", 76, "male");
 console.log(person1);
```

```
var person2 = new Person("Courtney", 34, "female");
 console.log(person2);
```

In the code above, we have created a constructor function named Person. Whenever we want to create a new object of the type Person, We need to create it using the new keyword:

```
var person3 = new Person("Lilly", 17, "female");
```

The above line of code will create a new object of the type Person. Constructor functions allow us to group similar objects.

55.Which method is used to retrieve a character from a certain index?

The charAt() function of the JavaScript string finds a char element at the supplied index. The index number begins at 0 and continues up to n-1, Here n is the string length. The index value must be positive, higher than, or the same as the string length

56. In JavaScript, how many different methods can you make an object?

In JavaScript, there are several ways to declare or construct an object

1. Object.
2. using Class.
3. create Method.
4. Object Literals.
5. using Function.
6. Object Constructor.

57.What are classes in javascript?

Introduced in the ES6 version, classes are nothing but syntactic sugars for constructor functions. They provide a new way of declaring constructor functions in javascript.

58.Difference between Async/Await and Generators usage to achieve the same functionality.

- Generator functions are run by their generator yield by yield which means one output at a time, whereas Async-await functions are executed sequentially one a er another.
- Async/await provides a certain use case for Generators easier to execute. T
- he output result of the Generator function is always value: X, done: Boolean, but the return value of the Async function is always an assurance or throws an error

59.What are the primitive data types in JavaScript?

A primitive is a data type that isn't composed of other data types. It's only capable of displaying one value at a time. By definition, every primitive is a built-in data type (the compiler must be knowledgeable of them) nevertheless, not all built-in datasets are primitives. In JavaScript, there are 5 different forms of basic data. The following values are available:

1. Boolean

2. Undefined

3. Null

4. Number

5. String

60.What is the role of deferred scripts in JavaScript?

The processing of HTML code while the page loads are disabled by nature till the script hasn't halted. Your page will be affected if your network is a bit slow, or if the script is very he y. When you use Deferred, the script waits for the HTML parser to finish before executing it. This reduces the time it takes for web pages to load, allowing them to appear more quickly

61.What has to be done in order to put Lexical Scoping into practice?

To support lexical scoping, a JavaScript function object's internal state must include not just the function's code but also a reference to the current scope chain.

62.New features in ES6 version.

The new features introduced in ES6 version of JavaScript are:

Let and const keywords.

Arrow functions.

Multi-line Strings.

The destructuring assignment.

Enhanced object literals.

Promises.

63.Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during run-time in contrast to a statically typed language, where the type of a variable is checked during compile-time.

64.What is closure in JavaScript?

A closure consists of references to the surrounding state (the lexical environment) and a function that has been wrapped (contained). In other words, a closure enables inner functions to access the scope of an outside function. Closures are formed whenever a function is created in JavaScript, during function creation time

65.What is Critical Rendering Path?

The Critical Rendering Path is the sequence of steps the browser goes through to convert the HTML, CSS, and JavaScript into pixels on the screen. Optimizing the critical render path improves render performance. The critical rendering path includes the Document Object Model (DOM), CSS Object Model (CSSOM), render tree and layout

66.What are basic JavaScript array methods?

Some of the basic JavaScript methods are:

push() method : adding a new element to an array. Since JavaScript arrays are changeable objects, adding and removing elements from an array is simple. Additionally, it alters itself when we change the array's elements.

Syntax: Array.push(item1, item2 …)

pop() method : This method is used to remove elements from the end of an array.

Syntax: Array.pop()

slice() method : This method returns a new array containing a portion of the original array, based on the start and end index provided as arguments

Syntax: Array.slice (startIndex , endIndex);

map() method : The map() method in JavaScript creates an array by calling a specific function on each element present in the parent array. It is a non mutating method. Generally, the map() method is used to iterate over an array and call the function on every element of an array.

Syntax: Array.map(function(currentValue, index, arr), thisValue)

Applies a function to each element of an array and creates a new array with the results.
Example:

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(number => number * 2); // [2, 4, 6, 8, 10]
```

reduce() method : The array reduce() method in JavaScript is used to reduce the array to a single value and executes a provided function for each value of the array (from left to right) and the return value of the function is stored in an accumulator.

Syntax: Array.reduce(function(total, currentValue, currentIndex, arr), initialValue)

Applies a function against an accumulator and each element in an array (from left to right) to reduce it to a single value.

Example:

```
const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((accumulator, number) => accumulator + number, 0); // 10
```

filter():

Creates a new array containing only elements that pass a test implemented by a provided function.

Example:

```
const numbers = [1, 2, 3, 4, 5];

 const evenNumbers = numbers.filter(number => number % 2 === 0); // [2, 4]
```

67.What is the rest parameter and spread operator?

Rest parameter (…):

- It offers a better method of managing a function's parameters.
- We can write functions that accept a variable number of arguments using the rest parameter syntax.
- The remainder parameter will turn any number of inputs into an array.
- Additionally, it assists in extracting all or some of the arguments.
- Applying three dots (...) before the parameters enables the use of rest parameters.

Syntax:

```
function extractingArgs(...args){

 return args[1];

 }

// extractingArgs(8,9,1); // Returns 9
```

```
 function addAllArgs(...args){
 let sumOfArgs = 0;
let i = 0;
while(i < args.length){
sumOfArgs += args[i];
 i++;
}
return sumOfArgs;
} addAllArgs(6, 5, 7, 99); // Returns 117
addAllArgs(1, 3, 4); // Returns 8
```

Note- Rest parameter should always be used at the last parameter of a function.

Spread operator(…) : Although the spread operator's syntax is identical to that of the rest parameter, it is used to spread object literals and arrays. Spread operators are also used when a function call expects one or more arguments.

Syntax:

```
function addFourNumbers(num1,num2,num3,num4){
 return num1 + num2 + num3 + num4; }
let fourNumbers = [5, 6, 7, 8];
addFourNumbers(...fourNumbers);
// Spreads [5,6,7,8] as 5,6,7,8
let array1 = [3, 4, 5, 6];
let clonedArray1 = [...array1];
// Spreads the array into 3,4,5,6
console.log(clonedArray1); // Outputs [3,4,5,6]
let obj1 = {x:'Hello', y:'Bye'};
 let clonedObj1 = {...obj1}; // Spreads and clones obj1 console.log(obj1);
let obj2 = {z:'Yes', a:'No'};
let mergedObj = {...obj1, ...obj2}; // Spreads both the objects and merges it
console.log(mergedObj);
 // Outputs {x:'Hello', y:'Bye',z:'Yes',a:'No'};
```

68.What are observables?

Observables in JavaScript are a way to handle asynchronous events. They are functions that return a stream of values, which can be used to represent data streams such as DOM events, mouse events, or HTTP requests.

Observables work by providing a way to subscribe to a stream of values, and then receiving those values as they become available. This allows you to respond to events in a more reactive way, without having to wait for the entire event stream to complete before processing it. To use observables in JavaScript, you can use the RxJS library.

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {

subscriber.next(1);

subscriber.next(2);

subscriber.next(3);

subscriber.complete();

}); observable.subscribe(value => { console.log(value); });
```

69.What is microtask in JavaScript?

A microtask is a short function which is executed after the function or program which created it exits and only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the user agent to drive the script's execution environment

70.What Pure Functions in JavaScript?

A function or section of code that always yields the same outcome when the same arguments are supplied is known as a pure function. It is independent of any state or data changes that occur while a program is running. Instead, it just relies on the arguments it is given.

Additionally, a pure function does not result in any side effects that can be seen, such as network queries, data alteration, etc.

71.What are the various statements in error handling?

Below are the list of statements used in an error handling

- try: This statement is used to test a block of code for errors
- catch: This statement is used to handle the error
- throw: This statement is used to create custom errors.
- finally: This statement is used to execute code after try and catch regardless of the result.

72. What do you mean by strict mode in javascript and characteristics of javascript strict-mode?

In ECMAScript 5, a new feature called JavaScript Strict Mode allows you to write a code or a function in a "strict" operational environment. When it comes to throwing errors, javascript is often 'not extremely severe'. However, in "Strict mode," all errors, even silent faults, will result in a throw. Debugging hence becomes more easier. Thus, the chance of a coder committing a mistake is decreased.

Characteristics of strict mode in javascript:

- Duplicate arguments are not allowed by developers.
- Use of javascript's keyword as parameter or function name is not allowed.
- The 'use strict' keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers
- Creating of global variables is not allowed.

73.What are the differences between cookie, local storage and session storage?

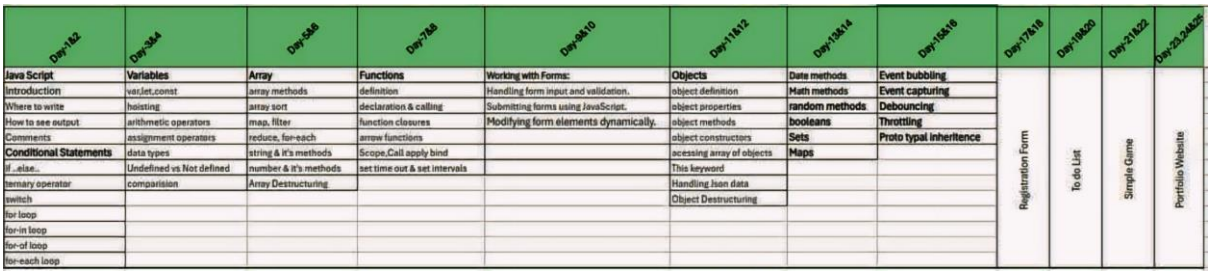
Cookie	Local storage	Session
Can be accessed on both server- side & client side	Can be accessed on client-side only	Can be accessed on client-side only
As configured using expires option	Lifetime is until deleted	Lifetime is until tab is closed
SSL is supported	SSL is not supported	SSL is not supported
Maximum size is 4 KB	Maximum size is 5 MB	Maximum size is 5 MB

74. Difference between Debouncing and Throttling.

Debouncing	Throttling
Debouncing waits for a certain time before invoking the function again.	An error has occurred in the eval() function
Ensures that the function is called only once, even if the event is triggered multiple times.	An error has occurred with a number "out of range"
Useful when you want to delay the invocation of a function until a certain period of inactivity has passed.	An error due to an illegal reference
Eg. You can debounce an async API request function that is called every time the user types in an input field.	An error due to syntax

Syntax:
function debounce(func, delay)
{
let timerId;
return function () {
const context = this;
	Syntax:
	function throttle(callback, delay = 1000) {
	let shouldWait = false;
	return (...args) => {
	if (shouldWait) return;
	callback(...args);

const args = arguments;
clearTimeout(timerId);
timerId = setTimeout(function ()
{ func.apply(context, args);
}, delay);
}; }
	shouldWait = true;
	setTimeout(() => {
	shouldWait = false;
	}, delay);
	}; }



Day-1&2	Day-3&4	Day-5&6	Day-7&8	Day-9&10	Day-11&12	Day-13&14	Day-15&16	Day-17&18	Day-19&20	Day-21&22	Day-23,24&25
Java Script	**Variables**	**Array**	**Functions**	**Working with Forms:**	**Objects**	Date methods	**Event bubbling**	Registration Form	To do List	Simple Game	Portfolio Website
Introduction	var,let,const	array methods	definition	Handling form input and validation.	object definition	**Math methods**	**Event capturing**				
Where to write	hoisting	array sort	declaration & calling	Submitting forms using JavaScript.	object properties	**random methods**	**Debouncing**				
How to see output	arithmetic operators	map, filter	function closures	Modifying form elements dynamically.	object methods	**booleans**	**Throttling**				
Comments	assignment operators	reduce, for-each	arrow functions		object constructors	**Sets**	**Proto typal inheritence**				
Conditional Statements	data types	string & it's methods	Scope,Call apply bind		acessing array of objects	**Maps**					
If ..else..	Undefined vs Not defined	number & it's methods	set time out & set intervals		This keyword						
ternary operator	comparision	Array Destructuring			Handling Json data						
switch					Object Destructuring						
for loop											
for-in loop											
for-of loop											
for-each loop											