**Question 1**

**for (var i = 0; i < 3; i++) {**

**setTimeout(function() {**

**alert(i);**

**}, 1000 + i);**

**}**

A) Number 3 alerted 3 times

b) Number 0 alerted three times

c) Number 1,2,3 alerted

d)undefined

Detail

**When the loop runs, it schedules three separate set Timeout functions with delays of 1000 ms (1 second), 1001 ms, and 1002 ms. These timeouts are set almost simultaneously because the loop doesn't wait for each timeout to complete before moving on to the next iteration.**

**However, due to the closure behavior in JavaScript, the i variable inside the set Timeout callback function will be referencing the final value of i (which is 3) when the timeouts execute. As a result, when the timeouts fire after the delays, they will all alert the value 3.**

**So, the correct answer would be:**

**b) Number 3 alerted three times**

**Q2 :(function () {**

**var a = b = 5;**

**})();**

**console.log(b);**

**5**

**Details:**

1. var a = b = 5;

• Inside the immediately invoked function expression (IIFE), you're declaring a variable a and assigning it the value of b, which is also assigned the value 5. Note that b is not declared with the var keyword, so it becomes a global variable.

2. console.log(b);

• Outside the IIFE, you're trying to log the value of the variable b.

However, there's a scoping issue here:

• The variable b is not explicitly declared with var, let, or const inside the IIFE, so it becomes a global variable.

• The IIFE's scope is limited to the function itself, and the a variable is scoped within the IIFE.

So, when you try to log the value of b outside the IIFE, it will work and log the value 5. This is because b is accessible in the global scope due to not having been declared with a specific scope keyword

**Q3. Output of the Followings?**

1

console.log (3 + "3");

2️⃣

console.log (3++ "3");

3️⃣

```js

console.log ([2] === [2]);

Here are the outputs for each of the given JavaScript console.log statements:

console.log (3 + "3");

The + operator is used for both addition and string concatenation in JavaScript. When one operand is a string, JavaScript will treat the other operand as a string and perform concatenation.

The number 3 is converted to a string, and then the strings "3" and "3" are concatenated.

Output: 33

console.log (3++ "3");

This code will result in a syntax error because ++ is used in an invalid way. The ++ operator is used for incrementing numeric values, but it cannot be used in this manner.

This line of code will likely produce a parsing error or another type of error due to the invalid use of the ++ operator.

console.log ([2] === [2]);

In JavaScript, objects are compared by reference, not by their content. When you compare two arrays using the === operator, it checks if they refer to the exact same object in memory, which is not the case here.

The two arrays [2] and [2] might have the same content, but they are separate instances in memory, so the comparison will result in false.

Output: false

**Q4: what is "pyramid of doom" in java script?**

when multiple nested callbacks are used, it will lead to the state pyramid of doom.

**Q5: what is callback in java script?**

In JavaScript, a "callback" refers to a function that is passed as an argument to another function and is intended to be executed or "called back" at a later point in time, often after a specific event or asynchronous operation has occurred. Callbacks are a fundamental concept in JavaScript, especially when dealing with asynchronous operations like fetching data from a server, handling user interactions, or dealing with timers.

Callbacks are commonly used to ensure that certain code executes only after a specific task is complete. For example, when making an AJAX request to a server to fetch data, you might provide a callback function that should be executed once the data is received from the server.

function fetch Data (URL, callback) {

// Simulating an asynchronous operation (e.g., AJAX request)

set Timeout (() => {

const data = {id: 1, name: "John Doe”};

callback(data); // Call the provided callback with the fetched data

}, 1000);

}

function handle Data(data) {

console.log ("Received data:", data);

}

// Usage: Pass the `handle Data` function as a callback

fetch Data ("https://example.com/data", handle Data);

```

In this example, the `fetch Data` function takes a URL and a callback function as arguments. It simulates an asynchronous operation using `set Timeout` and then calls the provided callback with the fetched data.

Callbacks are a building block for managing asynchronous operations in JavaScript, but they can lead to callback hell (also known as "pyramid of doom") when multiple nested callbacks are used, which can make the code hard to read and maintain. To mitigate this, modern JavaScript has introduced techniques like Promises and async/await to handle asynchronous code in a more organized and readable manner.

**Q6: Difference between let, Var, const in JavaScript?**

|  |  |  |
| --- | --- | --- |
| let | const | var |
| `let` is used to declare block-scoped variables | `const` is used to declare block-scoped variables, like `let`. | `var` is the oldest way to declare variables in JavaScript. |
| `let` are mutable | Variables declared with `const` are immutable | Variables declared with `var` are function-scoped (or globally scoped if declared outside of any function). |
| `let` variables are hoisted to the top of their block (but not initialized) and are not accessible before the declaration | `const` variables must be initialized when declared and cannot be reassigned to a new value | `var` variables are hoisted to the top of their function or global context and are initialized with the value `undefined`. |
| Example:  ```JavaScript  let x = 10;  x = 20; // Value can be changed | `const` does not prevent changes to the properties of objects or arrays, only the reassignment of the variable itself. | `var` variables can be reassigned and re-declared within the same scope. |
|  | Example:  ```JavaScript  const y = 5;  // y = 10; // This would result in an error  ``` | var z = 15;  z = 25; // Value can be changed  var z = 30; // Redefined in the same scope  ``` |

**Q7 Datatypes in Java Script?**

|  |  |  |
| --- | --- | --- |
| Primitive Data Types |  |  |
|  | String | let name = "John"; |
|  | Number | let age = 25; let price = 99.99; |
|  | Boolean | let is Active = true; let is Student = false; |
|  | Undefined | let undefined Variable; |
|  | Null | let empty Value = null; |
|  | Symbol | const unique Key = Symbol("description"); |
| Complex Data Types |  |  |
|  | Object | let person = {name: "Alice", age: 30 }; |
|  | Array | let numbers = [1, 2, 3, 4, 5]; |
|  | Function | function greet(name) {   return "Hello, " + name + "!";  } |
|  | Date | let current Date = new Date(); |
|  | RegExp | let pattern = /abc/; |

**Q8.Promises in JavaScript:**

Promises in JavaScript provide a structured way to handle asynchronous operations, like fetching data, using .then() to handle success and .catch() to handle errors.

|  |
| --- |
| const fetchData = () => {  return new Promise((resolve, reject) => {  setTimeout(() => {  resolve("Data fetched successfully");  }, 1000);  });  };  fetchData()  .then(result => console.log(result))  .catch(error => console.error(error)); |

**Q9. Hoisting in JavaScript:**

Hoisting in JavaScript moves variable and function declarations to the top of their scope during compilation, allowing variables to be used before they're declared, resulting in undefined in this case.

|  |
| --- |
| Javascript  console.log(x); // Outputs: undefined  var x = 5; |