|  |
| --- |
| **Question 35. Explain the concept of a closure in JavaScript and provide an example.**  Ans. A closure is a JavaScript feature that allows a function to remember and access its outer (enclosing) function's variables and parameters even after the outer function has finished executing. Closures are created whenever a function is defined within another function. Here's an example:  Javascript  function outer() {  const outerVar = 'I am from outer function';  function inner() {  console.log(outerVar); // Accesses outerVar from the outer function  }  return inner;  }  const closureFunc = outer();  closureFunc(); // Outputs: ""I am from outer function""  **Question 36. How can you iterate over the properties of an object in JavaScript?**  Ans. You can iterate over the properties of an object in JavaScript using various methods, such as for...in loops or methods like Object.keys(), Object.values(), and Object.entries(). Here's an example using for...in:  Javascript  const person = {  name: 'Alice',  age: 30,  city: 'New York'  };  for (const key in person) {  if (person.hasOwnProperty(key)) {  console.log(`${key}: ${person[key]}`);  }  }  **Question 37. What is memoization, and how can it be implemented in JavaScript?**  Ans. Memoization is an optimization technique used to cache the results of expensive function calls and return the cached result when the same inputs occur again. It can improve the performance of functions that are called with the same arguments multiple times. You can implement memoization in JavaScript using an object to store the results. Here's a simple example:  Javascript  function memoize(fn) {  const cache = {};  return function (...args) {  const key = JSON.stringify(args);  if (cache[key]) {  return cache[key];  } else {  const result = fn(...args);  cache[key] = result;  return result;  }  };  }  // Example usage  const expensiveFunction = memoize(function (n) {  console.log(`Calculating for ${n}`);  return n \* 2;  });  console.log(expensiveFunction(5)); // Output: Calculating for 5, 10  console.log(expensiveFunction(5)); // Output: 10 (cached)  **Question 38. Describe the purpose of the "arguments" object in JavaScript functions.**  Ans. The "arguments" object is a special object available within all JavaScript functions. It contains an array-like list of the arguments passed to the function, regardless of the number of named parameters defined in the function's signature. It allows you to work with a variable number of arguments. However, it is not a true array but an "array-like" object and lacks array methods. Modern JavaScript encourages using the rest parameters syntax (...args) instead of the "arguments" object for better flexibility and compatibility with array methods.  **Question 39. How can you compare two JavaScript objects for equality?**  Ans. To compare two JavaScript objects for equality (i.e., having the same properties and values), you can't use == or === directly because they compare references, not object contents. You need to implement a custom comparison function that recursively checks the properties and values of the objects. Here's a simple example:  Javascript  function deepEqual(obj1, obj2) {  if (obj1 === obj2) {  return true;  }  if (typeof obj1 !== 'object' || typeof obj2 !== 'object' || obj1 === null || obj2 === null) {  return false;  }  const keys1 = Object.keys(obj1);  const keys2 = Object.keys(obj2);  if (keys1.length !== keys2.length) {  return false;  }  for (const key of keys1) {  if (!keys2.includes(key) || !deepEqual(obj1[key], obj2[key])) {  return false;  }  }  return true;  }  // Example usage  const objA = { x: 1, y: { z: 2 } };  const objB = { x: 1, y: { z: 2 } };  console.log(deepEqual(objA, objB)); // Output: true  **Question 40. What is a JavaScript generator, and how does it work?**  Ans. A JavaScript generator is a special type of function that allows you to pause and resume its execution. It's defined using the function\* syntax and contains one or more yield statements. Generators are used to generate a sequence of values lazily and iteratively. When you call a generator function, it returns an iterator object that can be used to control the execution of the generator. You can pause the generator using yield and resume it using the iterator's .next() method. Here's a simple example:  Javascript  function\* countToThree() {  yield 1;  yield 2;  yield 3;  }  const iterator = countToThree();  console.log(iterator.next().value); // Output: 1  console.log(iterator.next().value); // Output: 2  console.log(iterator.next().value); // Output: 3  **Hard**  **Question 1. What is the difference between the "call," "apply," and "bind" methods in JavaScript?**  Ans. 1.call() and apply() are methods that allow you to invoke a function with a specific this value and arguments. The main difference is how you pass arguments:  -call(thisArg, arg1, arg2, ...) accepts arguments individual  -apply(thisArg, [arg1, arg2, ...]) accepts arguments as an array.  2.bind() creates a new function with a fixed this value and, optionally, pre-specified arguments. It doesn't invoke the function immediately but returns a new function that can be called later.  **Question 2. How does JavaScript handle memory management, and what is garbage collection?**  Ans. JavaScript uses automatic memory management, which includes garbage collection. Here's how it works:  -When variables go out of scope or are no longer referenced, they become eligible for garbage collection.  -The garbage collector identifies these unreferenced objects and frees up memory by deallocating them.  -JavaScript engines use various garbage collection algorithms to optimize memory usage.  -Developers can help by being mindful of memory usage, avoiding memory leaks, and using tools for memory profiling.  **Question 3. Explain the concepts of "thunk" and "saga" in the context of JavaScript Redux.**  Ans. -Thunk: A thunk is a function that encapsulates an action to be dispatched in Redux. It delays the action's execution until a later time or condition is met. Thunks are often used for asynchronous actions, enabling side effects and API calls.  -Saga: A saga is a middleware library for Redux that provides a way to manage side effects, such as handling asynchronous actions, in a more structured and testable manner. Sagas use generator functions to describe the flow of actions and can listen for specific actions to trigger side effects."  **Question 4. Describe the differences between the ES6 "class" and "constructor function" approaches for creating objects.**  Ans. -Class: In ES6, the class syntax provides a more structured and readable way to create constructor functions and define methods. It supports constructor functions, inheritance, and getters/setters.  -Constructor Function: Traditional constructor functions use the function keyword and require defining the constructor and adding methods to the prototype manually. They have been used prior to ES6 and are less verbose.  **Question 5. What is the Event Emitters pattern in JavaScript, and how is it used?**  Ans. The Event Emitters pattern is a design pattern in JavaScript used for implementing custom event handling. It allows objects to subscribe to and emit custom events, facilitating communication between different parts of an application. Node.js's EventEmitter is a built-in example of this pattern. To use it, you create an event emitter object, define custom events, and emit events when specific actions or conditions occur. Subscribers (listeners) can then react to these events by attaching event handlers.  **Question 6. How can you implement a singleton pattern in JavaScript?**  Ans. The singleton pattern ensures that a class has only one instance, which can be accessed globally. You can implement it in JavaScript using a module pattern or by defining a constructor function with closure. Here's an example using a constructor function:  Javascript  function Singleton() {  if (!Singleton.instance) {  Singleton.instance = this;  }  return Singleton.instance;  }  const instance1 = new Singleton();  const instance2 = new Singleton();  console.log(instance1 === instance2); // Output: true (both instances are the same)  **Question 7. Explain the concept of memoization and provide a use case for it.**  Ans. Memoization is an optimization technique where the results of expensive function calls are cached based on their input arguments. When the same inputs are provided again, the cached result is returned instead of re-computing it. A common use case is optimizing recursive functions or functions with expensive computations, such as Fibonacci calculations.  **Question 8. What are web workers in JavaScript, and how can they be used to improve performance?**  Ans. Web workers are a feature in JavaScript that allows you to run JavaScript code in the background, separate from the main browser thread. They can be used to perform tasks concurrently, improving performance by offloading CPU-intensive work. Web workers are especially useful for tasks like data processing, image manipulation, or other operations that can be parallelized.  **Question 9. What is the "prototype chain" in JavaScript, and how does it relate to inheritance?**  Ans. The prototype chain is a fundamental concept in JavaScript's object-oriented programming model. It represents the way objects inherit properties and methods from their prototypes. When you access a property or method on an object, JavaScript searches for it on the object itself and, if not found, looks up the prototype chain. This chain allows objects to inherit from other objects, enabling prototype-based inheritance.  **Question 10. Describe the differences between "shallow copy" and "deep copy" in JavaScript objects.**  Ans. -Shallow Copy: A shallow copy of an object creates a new object with a new reference but copies only the top-level properties. Nested objects or arrays within the original object are still referenced, not duplicated. Shallow copies are created using methods like Object.assign() or the spread operator.  -Deep Copy: A deep copy creates a new object and recursively duplicates all properties and nested objects, ensuring that the entire structure is independent. It's a true clone of the original object. Achieving a deep copy often requires custom recursive functions or libraries like Lodash's \_.cloneDeep().  **Question 11. How can you implement a publish-subscribe pattern in JavaScript?**  Ans. The publish-subscribe pattern (pub-sub) allows objects to subscribe to events (or messages) and receive notifications when those events occur. You can implement it using custom code or libraries like EventEmitter. Subscribers register their interest in specific events, and publishers emit those events when appropriate. It's a way to decouple components and enable communication between them without direct dependencies.  **Question 12. Explain the concept of a "promise" and its states in JavaScript.**  Ans. A promise is a JavaScript object representing the eventual completion or failure of an asynchronous operation and its resulting value. Promises have three states:  -Pending: The initial state, representing an ongoing operation.  -Fulfilled (Resolved): The state when the operation is successful, and the promise holds a resolved value.  -Rejected: The state when the operation fails, and the promise holds a rejection reason.  **Question 13. What is the purpose of the JavaScript async and await keywords in asynchronous code?**  Ans. The async and await keywords are used in JavaScript to simplify working with asynchronous code. Functions marked as async return promises implicitly, and await is used within async functions to pause execution until a promise is resolved. This helps write asynchronous code that resembles synchronous code, making it more readable and easier to reason about.  **Question 14. Describe the role of the "WeakMap" and "WeakSet" data structures in JavaScript.**  Ans. WeakMap: A WeakMap is a collection of key-value pairs where keys are objects, and values can be any data type. Unlike regular maps, WeakMap keys are weakly held, meaning they don't prevent objects from being garbage collected when they're no longer in use. WeakMaps are often used for private data storage or for associating metadata with objects.  WeakSet: A WeakSet is a collection of unique objects, and like WeakMaps, it doesn't prevent objects from being garbage collected. It's typically used to store a set of objects where the reference to the objects is weak.  **Question 15. How do you implement a linked list in JavaScript?**  Ans. A linked list in JavaScript can be implemented using objects to represent nodes, where each node has a value and a reference (usually next) to the next node in the list. You also typically keep a reference to the head (the first node) of the list. Here's a basic example of a singly linked list:  Javascript  class Node {  constructor(value) {  this.value = value;  this.next = null;  }  }  class LinkedList {  constructor() {  this.head = null;  }  // Add methods like append, prepend, delete, search, etc.  }  **Question 16. What is the purpose of the "Generator" object in JavaScript, and how can it be used?**  Ans. A Generator is a special type of function in JavaScript that can be paused and resumed. It's defined using the function\* syntax and contains one or more yield statements. Generators are often used for lazy and efficient generation of values, especially in scenarios like iterating over large datasets or implementing custom iterators.  **Question 17. Explain the concept of the "event loop" in JavaScript and how it handles asynchronous tasks.**  Ans. The event loop is a central part of JavaScript's concurrency model. It's responsible for handling asynchronous tasks and callbacks. The event loop continuously checks the message queue for pending tasks. When a task is found, it's executed, and the loop continues. This allows JavaScript to perform non-blocking I/O operations and execute callbacks when events occur, ensuring responsive and efficient code execution.  **Question 18. How can you implement a deep copy of an object that contains nested objects in JavaScript?**  Ans. You can implement a deep copy of an object containing nested objects using a recursive function. Here's a basic example:  Javascript  function deepCopy(obj) {  if (obj === null || typeof obj !== 'object') {  return obj; // Return non-objects and null as is  }  if (Array.isArray(obj)) {  return obj.map(item => deepCopy(item)); // Handle arrays  }  const copiedObj = {};  for (const key in obj) {  if (obj.hasOwnProperty(key)) {  copiedObj[key] = deepCopy(obj[key]); // Recursively copy nested objects  }  }  return copiedObj;  }  // Example usage  const original = { a: 1, b: { c: 2 } };  const copied = deepCopy(original);  **Question 19. What is the "Proxy" object in JavaScript, and how can it be used for intercepting operations on objects?**  Ans. The Proxy object is a built-in feature in JavaScript that allows you to intercept and customize operations on objects. It can be used to add custom behavior, validation, or access control to objects. A Proxy is created with a target object and a handler object containing methods for intercepting various operations like property access, assignment, function invocation, etc.  **Question 20. Describe the differences between JavaScript "callbacks," "promises," and "async/await" for handling asynchronous operations.**  Ans. Callbacks: Callbacks are functions passed as arguments to other functions, typically used for asynchronous operations. They can lead to callback hell (nested callbacks) and can be challenging to read and maintain.  Promises: Promises are objects representing the eventual completion or failure of an asynchronous operation. They provide a cleaner way to handle async code with .then() and .catch(), improving readability.  async/await: async functions return promises implicitly and allow you to write async code that looks more like synchronous code. await pauses execution until a promise is resolved or rejected, simplifying error handling and making code more readable.  **Question 21. Explain the concept of "composition over inheritance" in JavaScript.**  Ans. "Composition over inheritance" is a design principle that suggests favoring object composition and building complex objects by combining simpler objects, rather than relying heavily on class inheritance hierarchies. This approach promotes flexibility, code reuse, and avoids some of the issues associated with deep inheritance hierarchies.  **Question 22. What are "maps" and "sets" in JavaScript, and how are they used?**  Ans. Maps: Maps are collections of key-value pairs, where keys can be of any data type (including objects and functions). They allow you to associate values with keys and are particularly useful for data storage and retrieval. Maps maintain the insertion order of keys, which makes them suitable for scenarios where order matters.  Sets: Sets are collections of unique values, where each value can occur only once. Sets are often used to store a collection of unique items or to eliminate duplicate values from an array.  **Question 23.How can you implement a debounce function in JavaScript to limit the frequency of a function's execution?**  Ans. A debounce function is used to delay the execution of a function until a certain amount of time has passed without further function calls. Here's a simple implementation:  Javascript  function debounce(func, delay) {  let timeoutId;  return function (...args) {  clearTimeout(timeoutId);  timeoutId = setTimeout(() => {  func(...args);  }, delay);  };  }  // Example usage  const debouncedFunction = debounce(() => {  // Your function logic here  }, 300); // Executes at most every 300 milliseconds  **Question 24. Describe the differences between "stack" and "queue" data structures in JavaScript.**  Ans. Stack: A stack is a linear data structure where elements are added and removed from the same end called the ""top."" It follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first one to be removed. Stacks are often used for function call management and undo/redo functionality.  Queue: A queue is another linear data structure where elements are added at one end (the ""rear"") and removed from the other end (the ""front""). It follows the First-In-First-Out (FIFO) principle, meaning the first element added is the first one to be removed. Queues are used for tasks like managing tasks in a print queue or handling asynchronous requests.  **Question 25. What is the "Immutable.js" library in JavaScript, and why might you use it?**  Ans. Immutable.js is a JavaScript library that provides immutable data structures, such as lists, maps, and sets. Immutable data structures cannot be changed after they are created, which helps prevent unintentional data mutations. You might use Immutable.js in scenarios where you want to ensure data integrity, enable efficient change detection in React applications, or work with persistent data structures efficiently.  **Question 26. How can you implement a "throttle" function in JavaScript to control the rate of function calls?**  Ans. A throttle function limits the rate at which a function can be called, ensuring it's not called too frequently. Here's a basic implementation:  Javascript  function throttle(func, delay) {  let lastCallTime = 0;  return function (...args) {  const now = Date.now();  if (now - lastCallTime >= delay) {  func(...args);  lastCallTime = now;  }  };  }  // Example usage  const throttledFunction = throttle(() => {  // Your function logic here  }, 300); // Calls at most once every 300 milliseconds  **Question 27. What are the differences between "prototypal inheritance" and "classical inheritance" in JavaScript?**  Ans. Prototypal Inheritance: In JavaScript, objects can inherit properties and methods directly from other objects. Prototypal inheritance is based on the prototype chain, where objects have a prototype reference and can delegate property lookups to their prototypes. It's more flexible than classical inheritance and allows objects to inherit from multiple prototypes.  Classical Inheritance: Classical inheritance is a concept from class-based languages like Java or C++. It uses class hierarchies and extends classes to create new ones. In JavaScript, classical inheritance can be emulated using constructor functions and the prototype object, but it's less flexible than prototypal inheritance.  **Question 28. Describe the "constructor pattern" and the "module pattern" in JavaScript.**  Ans. Constructor Pattern: The constructor pattern is a way to create objects in JavaScript using constructor functions. Constructor functions are called with the new keyword and create instances with shared methods and properties defined in their prototypes. It's a common pattern for defining classes-like structures in JavaScript.  Module Pattern: The module pattern is a design pattern that encapsulates code into self-contained modules. It often involves using immediately-invoked function expressions (IIFE) to create private and public members, providing a way to achieve data encapsulation and avoid polluting the global scope. Modules promote organization and reusability of code.  **Question 29. How can you handle memory leaks in JavaScript, and what tools can help with memory profiling?**  Ans. To handle memory leaks in JavaScript:  Ensure that you remove event listeners when they are no longer needed to prevent circular references.  Avoid global variables and use variable scope wisely.  Use tools like the Chrome DevTools Memory panel, Heap Snapshots, and Leak Detection to profile and identify memory issues.  Regularly test your application with memory profiling tools to catch and fix potential leaks early.  **Question 30. What is the purpose of the JavaScript Symbol data type, and how can it be used?**  Ans. Symbols are a unique and immutable data type introduced in ES6. Their main purpose is to create property keys that are guaranteed to be unique, preventing naming collisions in objects. Symbols can be used as keys in objects and are often used for creating private or hidden object properties. Example:  Javascript  const mySymbol = Symbol('description');  const obj = {  [mySymbol]: 'This is a symbol property',  };  **Question 31. Explain the differences between "WebSockets" and "HTTP polling" for real-time communication in web applications.**  Ans. WebSockets: WebSockets provide full-duplex communication over a single, long-lived connection. They are efficient for real-time applications, such as chat or gaming, as they allow instant data exchange in both directions. WebSockets are event-driven and have low latency.  HTTP Polling: HTTP polling involves repeatedly sending HTTP requests to a server at predefined intervals to check for updates. It's less efficient and has higher latency compared to WebSockets, as it requires frequent requests and responses.  **Question 32. What is "closure memory leak," and how can it be avoided in JavaScript?**  Ans. A closure memory leak occurs when a function retains references to variables or objects that are no longer needed, preventing them from being garbage collected. To avoid closure memory leaks:  Be mindful of what variables/functions are captured in closures.  Use functions like setTimeout() with care, as they can unintentionally capture variables.  Explicitly remove event listeners and references when they are no longer needed.  **Question 33. Describe the "fluent interface" design pattern in JavaScript, and provide an example.**  Ans. The fluent interface is a design pattern that allows method chaining to create expressive and readable code. Each method returns an object or value that has more methods that can be called in sequence. An example using a fictional StringBuilder class:  Javascript  class StringBuilder {  constructor() {  this.value = '';  }  append(text) {  this.value += text;  return this; // Return the instance for chaining  }  toString() {  return this.value;  }  }  const result = new StringBuilder()  .append('Hello')  .append(' ')  .append('World')  .toString(); // ""Hello World""  **Question 34. How can you implement a "curry" function in JavaScript?**  Ans. A curry function in JavaScript transforms a function that takes multiple arguments into a series of unary (single-argument) functions. Here's a basic example of a curry function:  Javascript  function curry(fn) {  return function curried(...args) {  if (args.length >= fn.length) {  return fn(...args);  } else {  return function (...moreArgs) {  return curried(...args, ...moreArgs);  };  }  };  }  // Example usage  function add(a, b, c) {  return a + b + c;  }  const curriedAdd = curry(add);  const add2 = curriedAdd(2);  const result = add2(3)(4); // 9  **Question 35. What is "event-driven programming," and how is it used in JavaScript?**  Ans. Event-driven programming is a paradigm where the flow of the program is determined by events and event handlers. In JavaScript, it's commonly used for building interactive web applications. Events can be user interactions (clicks, keypresses), HTTP requests, or custom events. JavaScript libraries and frameworks like Node.js and the browser's DOM provide APIs for handling events using event listeners and callbacks.  **Question 36. What would be the output? True or False ?**  Ans.  Javascript  const x = 0.1 ;  const y = 0.2 ;  console.log(x + y == 0.3) ;  FALSE because when you console it will give 0.300004 that's why  **Question 37. What are two-way data binding and one-way data flow, and how are they different?**  Ans. Two-way data binding means that UI fields are bound to model data dynamically such that when a UI field changes, the model data changes with it and vice-versa.  One way data flow means that the model is the single source of truth. Changes in the UI trigger messages that signal user intent to the model (or “store” in React). Only the model has access to change the app’s state. The effect is that data always flows in a single direction, which makes it easier to understand.  One way data flows are deterministic, whereas two-way binding can cause side-effects which are harder to follow and understand.  **Question 38. Write a function that generates all permutations of a given string.**  Ans.  Javascript  function generatePermutations(str) {  const permutations = [];  function permute(prefix, remaining) {  if (remaining.length === 0) {  permutations.push(prefix);  return;  }  for (let i = 0; i < remaining.length; i++) {  const newPrefix = prefix + remaining[i];  const newRemaining = remaining.slice(0, i) + remaining.slice(i + 1);  permute(newPrefix, newRemaining);  }  }  permute('', str);  return permutations;  }  const inputString = 'abc';  console.log(generatePermutations(inputString));"  **Question 39. What will be the output of the following code?**  Javascript  var x = { foo : 1};  var output = (function() {  delete x.foo;  return x.foo;  })();  console.log(output);"  Ans. The result of the above code will be the output.  output. delete operator is used to delete a property from an object. In this example, x is the object with foo as a property. We are deleting foo property from a self-invoking function and in the next call (return x.foo;) trying to reference the deleted property “foo”, this will result undefined.  **Question 40. What will be the output of the following code?**  Javascript  x = 0.1;  y = 0.3;  z = x + y;  console.log(z === 0.4);  Ans. Now, here's where the issue arises:  JavaScript uses the IEEE 754 standard for representing floating-point numbers, which may introduce small rounding errors in calculations involving decimal fractions. In this case, the value of z might not be exactly 0.4 due to the way 0.1 and 0.3 are represented in binary floating-point.  So, when you run the code, you might find that z is very close to 0.4 but not exactly equal to it. Therefore, console.log(z === 0.4); may log false to the console because of this small floating-point precision issue.  To compare floating-point numbers for approximate equality, you can use a tolerance threshold, like so:  const tolerance = 0.000001; // Define a small tolerance threshold  console.log(Math.abs(z - 0.4) < tolerance); // Use Math.abs to handle negative differences  This code checks if the absolute difference between z and 0.4 is smaller than a defined tolerance threshold. If the difference is within the tolerance, it will be considered equal."  **Question 41. What will be the output of the following code?**  Javascript  const promise = new Promise((resolve, reject) => {  resolve('Resolved!');  reject('Rejected!');  });  promise  .then(data => console.log('Success:', data))  .catch(error => console.error('Error:', error));"  Ans. Expected Output: 'Success: Resolved!'  Explanation: The resolve function is called, so the then block is executed, and the catch block is not executed."  **Question 42. What will be the output of the following code?**  Javascript  console.log([] == []);  console.log([] === []);  Ans. Expected Output: False False  Explanation: Arrays are objects, and objects are compared by reference. The two empty arrays are different objects in memory, so both equality checks return false.  **Question 43. What will be the output of the following code?**  Javascript  function outer() {  let x = 10;  function inner() {  console.log(x);  }  return inner;  }  const closure = outer();  closure();  Ans. Expected Output: 10  Explanation: The inner function forms a closure over the x variable declared in the outer function, allowing it to access x even after outer has returned."  **Question 44. What will be the output of the following code?**  Javascript  const a = {};  const b = { key: 'value' };  const c = { key: 'value' };  a[b] = 123;  a[c] = 456;  console.log(a[b]);  Ans. Expected Output: 456  Explanation: In JavaScript, object keys are automatically converted to strings. So, a[b] and a[c] are equivalent to a['[object Object]']. Therefore, the value 456 overwrites the value 123 for the key '[object Object]'."  **Question 45. What will be the output of the following code?**  Javascript  const obj = {  [Symbol.toPrimitive](hint) {  if (hint === 'number') return 42;  if (hint === 'string') return 'Hello, World!';  return null;  }  };  console.log(obj + 1);  console.log(obj + ' JavaScript');  console.log(obj.toString());  Ans. Expected Output:43  'Hello, World! JavaScript'  '[object Object]'  Explanation: The object obj has a special method [Symbol.toPrimitive] which is used for type coercion. When using the + operator, JavaScript first tries to convert the object into a primitive value based on the provided hint ('number' or 'string'). If neither hint is provided, it defaults to null."  **Question 46. What will be the output of the following code?**  Javascript  const func = () => console.log('First');  (async () => {  await func();  console.log('Second');  })();  Ans. Expected Output: First  Second  Explanation: Even though await is used inside the async function, it doesn't affect the execution of func(), which is still executed synchronously.  **Question 47. What will be the output of the following code?**  Javascript  const obj = {};  Object.defineProperty(obj, 'value', {  get() {  console.log('Getting value');  return 42;  },  set(value) {  console.log('Setting value to', value);  }  });  console.log(obj.value);  obj.value = 100;  Ans. Expected Output: Getting value  42  Setting value to 100  Explanation: The get and set methods are defined using Object.defineProperty, so accessing obj.value triggers the get method, and setting obj.value triggers the set method. |