|  |
| --- |
| **1. Question: What is the difference between let, const, and var in JavaScript?**   * Answer:   + **let** and **const** were introduced in ECMAScript 6 (ES6) and have block-level scope, while **var** has function-level scope.   + Variables declared with **const** cannot be reassigned, whereas variables declared with **let** and **var** can be.   + **const** is used for constants, and it must be assigned a value when declared.   **2. Question: Explain the concept of closures in JavaScript.**   * Answer:   + Closures occur when a function is defined inside another function and has access to the outer function's variables.   + The inner function "closes over" the outer function's scope, retaining access to its variables even after the outer function has finished executing.   **3. Question: What is the event loop in JavaScript, and how does it work?**   * Answer:   + The event loop is a crucial concept for handling asynchronous operations in JavaScript.   + It continuously checks the message queue for tasks to execute. When the call stack is empty, the event loop takes the first task from the queue and pushes it onto the stack for execution.   **4. Question: How does prototypal inheritance work in JavaScript?**   * Answer:   + JavaScript uses prototypal inheritance, where objects can inherit properties and methods from other objects.   + Each object has a prototype, and if a property or method is not found on the object itself, the JavaScript engine looks for it in the object's prototype chain until it reaches the **Object** prototype.   **5. Question: Explain the concept of promises and how they differ from callbacks.**   * Answer:   + Promises are objects representing the eventual completion or failure of an asynchronous operation.   + Unlike callbacks, which can lead to callback hell (nested and hard-to-read code), promises provide a cleaner way to handle asynchronous operations and allow chaining with **.then()** and **.catch()**.   **6. Question: What is the purpose of the bind method in JavaScript?**   * Answer:   + The **bind** method is used to create a new function with a specified **this** value, and optionally, a set of initial arguments.   + It allows you to set the context in which a function will be called, particularly useful when dealing with event handlers or callbacks.   **7. Question: How does the this keyword work in JavaScript?**   * Answer:   + The value of **this** depends on how a function is called.   + In a regular function, **this** refers to the global object (e.g., **window** in a browser). In a method, it refers to the object the method was called on. Arrow functions, however, inherit **this** from the enclosing scope.   **8. Question: Explain the difference between == and === in JavaScript.**   * Answer:   + **==** is the equality operator and performs type coercion if the operands are of different types.   + **===** is the strict equality operator and does not perform type coercion. It checks both value and type equality.   **9. Question: What is the purpose of the async and await keywords in JavaScript?**   * Answer:   + **async** is used to declare a function as asynchronous, allowing the use of the **await** keyword inside it.   + **await** is used to pause the execution of an **async** function until a promise is settled, either resolving with a value or rejecting with a reason.   **10. Question: How does hoisting work in JavaScript?**   * Answer:   + Hoisting is a behavior in which variable and function declarations are moved to the top of their containing scope during the compilation phase.   + Variables declared with **var** are hoisted and initialized with **undefined**, while functions are hoisted with their entire definition.   + **let** and **const** declarations are also hoisted but are not initialized until the actual code execution reaches them.   **1. Code:**  console.log(2 + '2');  **Output Explanation:** This will output '22'. The **+** operator is overloaded in JavaScript, and when one of the operands is a string, it performs string concatenation.  **2. Code:**  const arr = [1, 2, 3];  arr[10] = 10;  console.log(arr);  **Output Explanation:** This will output **[ 1, 2, 3, <7 empty items>, 10 ]**. JavaScript arrays are sparse, and assigning a value to an index larger than the array's current length creates empty slots.  **3. Code:**  function greet()  { console.log(name);  var name = 'John'; }  greet();  **Output Explanation:** This will output **undefined**. In JavaScript, variable declarations (using **var**) are hoisted, but not their initializations. Therefore, **name** is hoisted to the top of the function, but its value is not defined until after the **console.log** statement.  **4. Code:**  let x = 10;  if (true)  { let x = 20; }  console.log(x);  **Output Explanation:** This will output **10**. The **let** keyword has block-level scope, so the inner **x** is a different variable than the outer **x**.  **5. Code:**  function multiply(x, y = 2) { return x \* y; }  console.log(multiply(3));  **Output Explanation:** This will output **6**. The function **multiply** has a default parameter **y** set to **2**. When called with only one argument (**3**), it uses the default value for **y**.  **6. Code:**  console.log(0.1 + 0.2 === 0.3);  **Output Explanation:** This will output **false**. Due to the way floating-point numbers are represented in JavaScript, the addition may result in a small precision error.  **7. Code:**  const person = { name: 'John', age: 30 };  const { name, age } = person;  console.log(name, age);  **Output Explanation:** This will output **John 30**. Object destructuring is used to extract values from the **person** object into variables **name** and **age**.  **8. Code:**  console.log(typeof null);  **Output Explanation:** This will output **'object'**. It's a historical quirk in JavaScript. **null** is of type **'object'**, but it should ideally be its own type.  **9. Code:**  console.log(3 > 2 > 1);  **Output Explanation:** This will output **false**. The expression is evaluated from left to right, so **3 > 2** results in **true**, and then **true > 1** is evaluated, which becomes **false**.  **10. Code:**  const a = [1, 2, 3];  const b = [...a, 4, 5];  console.log(b);  **Output Explanation:** This will output **[ 1, 2, 3, 4, 5 ]**. The spread (**...**) operator is used to create a new array (**b**) by spreading the elements of array **a** and appending **4** and **5** to it.  **1. Question: What is a closure in JavaScript?**   * **Answer:**   + A closure is a combination of a function and the lexical environment within which that function was declared. It allows the function to access variables from its outer scope even after the outer function has finished executing. In simpler terms, a closure "closes over" the variables it needs, encapsulating them within the function.   **2. Question: How do closures help in maintaining state in JavaScript?**   * **Answer:**   + Closures allow functions to remember the scope in which they were created. When a function is defined inside another function, it forms a closure and retains access to the outer function's variables. This property is leveraged to maintain state across multiple function calls without the need for global variables, making closures an essential concept for implementing private variables and data encapsulation.   **3. Question: How can closures be used to create private variables in JavaScript?**   * **Answer:**   + Closures enable the creation of private variables by encapsulating them within a function's scope. These variables are not directly accessible from outside the function, providing a form of data privacy. Here's an example:   function createCounter()  {  let count = 0;  return  { increment: function()  { count++; },  getCount: function() { return count; }  }; }  const counter = createCounter();  counter.increment();  console.log(counter.getCount());  // Outputs: 1  In this example, the **count** variable is private to the **createCounter** function, and the returned object provides methods to interact with it.  **4. Question: Explain the concept of a "free variable" in the context of closures.**   * **Answer:**   + In the context of closures, a free variable is a variable referenced by a function that is not defined within that function, nor passed as a parameter. Instead, the free variable is part of the lexical environment in which the function was created. Closures allow functions to "capture" and work with these free variables, even after the outer function has completed its execution.   **5. Question: Difference between shadow DOM vs virtual DOM.**   * **Answer:**   + Shadow DOM: It operates within the context of a single web component, creating an isolated DOM tree for that component, which shields its internal structure and styles from the rest of the page. Virtual DOM: It is an abstraction over the entire document's DOM, representing a virtual copy of the full DOM tree. |