1. **Explain the concept of hoisting in JavaScript.**

Hoisting in JavaScript refers to the behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase. However, only the declarations are hoisted, not the initializations. For example, if you **declare a variable** **using var at the bottom of a function**, you can still **access it at the top** of that function, but its value will be **undefined** until the actual assignment is encountered.

var x; // Declaration is hoisted to the top

console.log(x); // Output: undefined

x = 5; // Assignment remains in place

1. **What are closures in JavaScript, and why are they important?**

A closure is a function that has access to its own scope, the outer function's scope, and the global scope. Closures are important because they enable data encapsulation and help in creating private variables and functions. They are frequently used in JavaScript for tasks like maintaining state, creating factory functions, and implementing callback patterns.

function outerFunction() {

  let outerVariable = 'I am from the outer function';

  function innerFunction() {

    console.log(outerVariable);

  }

  return innerFunction;

}

const closure = outerFunction();

closure(); // Output: "I am from the outer function"

In this example:

1. outerFunction is defined, and within it, there's an innerFunction.
2. innerFunction has access to the outerVariable, which is declared in the outerFunction.
3. We then call outerFunction (), which returns the innerFunction but doesn't execute it immediately.
4. We store the returned innerFunction in the variable closure.
5. When we call closure(), it executes innerFunction, and even though outerVariable is not directly defined within innerFunction, it still has access to it. This is the closure in action, where innerFunction "remembers" its outer scope, even after outerFunction has finished executing.

The output of closure() is "I am from the outer function," demonstrating how a closure allows inner functions to retain access to variables and data from their enclosing (outer) functions.

1. **What is the event loop in JavaScript, and how does it work?**

The event loop is a crucial part of JavaScript's concurrency model. It manages the execution of code, including asynchronous code, by **continuously checking the call stack and the message queue**. When the call stack is empty, the event loop takes the next message from the queue and pushes it onto the stack, allowing it to execute. This process allows JavaScript to handle non-blocking, asynchronous operations efficiently.

|  |  |
| --- | --- |
| console.log("Start");  // Asynchronous operation using setTimeout  setTimeout(function () {    console.log("Timeout 1");  }, 1000);  // Another asynchronous operation using setTimeout  setTimeout(function () {    console.log("Timeout 2");  }, 500);  console.log("End"); | Start  End  Timeout 2  Timeout 1 |

1. We start by logging "Start."
2. Two **setTimeout** functions are used to create asynchronous operations. These functions schedule the provided callback functions to run after a specified delay. However, they don't block the main thread of execution.
3. The first **setTimeout** is set for a 1000-millisecond (1-second) delay, and the second one for a 500-millisecond (0.5-second) delay.
4. We log "End."

* "Start" is logged immediately because it's a synchronous operation.
* "End" is logged next, also synchronously.

1. **What is the difference between 'null' and 'undefined' in JavaScript?**

In JavaScript, **null** is a deliberate value that represents the absence of any object value, while **undefined** is a variable that has been declared but has not been assigned a value. **null** is typically set explicitly by developers, whereas **undefined** is the default value for uninitialized variables.

1. **What is the DOM, and how do you manipulate it using JavaScript?**

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure and content of a web page, allowing developers to access and manipulate elements and content. JavaScript can be used to select elements, change their attributes, add or remove elements, and respond to user interactions, making web pages interactive and dynamic.

1. **Explain the concept of 'callback functions' and provide an example.**

A callback is a function passed as an argument to another function.

function greet(name, callback) {

  const message = `Hello, ${name}!`;

  callback(message);

}

function displayMessage(message) {

  console.log(message);

}

greet("Alice", displayMessage);

1. **What are Promises in JavaScript, and how do they help with asynchronous programming?**

Promises are a built-in JavaScript feature for handling asynchronous operations. They represent a value that may not be available yet but will be resolved in the future. Promises provide a more structured and readable way to work with async code, and they allow chaining of operations. Here's an example:

fetch('https://api.example.com/data')

  .then(response => response.json())

  .then(data => console.log(data))

  .catch(error => console.error(error));

1. **What are the states of Promises in ES6?**

* **Pending**: This refers to the initial state of every promise. It indicates that the result has not yet been computed.
* **Fulfilled**: It refers to the completion of a task.
* **Rejected**: It indicates the failure that arises during computation.

1. **What is the difference between 'let,' 'const,' and 'var' for variable declaration in JavaScript?**

**'var'** is function-scoped, **'let'** and **'const'** are block-scoped. **'var'** allows variables to be redeclared and can lead to hoisting-related issues. **'let'** allows reassignment but not redeclaration, while **'const'** is used for constants and cannot be reassigned. Block-scoped variables are not hoisted to the top of their containing function or block, and they are only accessible within that specific block.

function exampleFunction() {

  if (true) {

    var x = 10; // Function-scoped variable

  }

  console.log(x); // Accessible here

}

function exampleFunction() {

  if (true) {

    let y = 20; // Block-scoped variable

    const z = 30; // Block-scoped variable

  }

  console.log(y); // Error: y is not defined (not accessible here)

  console.log(z); // Error: z is not defined (not accessible here)

}

1. **What is the difference between a parameter and an argument?**

* **Parameter:** A parameter is a variable that is part of a function's declaration. It acts as a placeholder for a value that the function expects to receive when it is called.

function greet(name) {

  console.log(`Hello, ${name}!`);

}

* **Argument:** An argument, on the other hand, is an actual value or expression that is passed to a function when it is called. It corresponds to a parameter in the function's definition.

greet("Alice");

1. **What is the result of the spread operator array** [...'apple'] **?**

Output: ['a', 'p', 'p', 'l', 'e']

**Explanation**: A string is an iterable type, and in an array, the spread operator transfers each character of an iterable to one element. As a result, each character in a string becomes an Array element.