10 advanced JavaScript questions

1. Question: What is the Event Loop in JavaScript, and how does it work?

Answer: The Event Loop is a fundamental part of JavaScript's concurrency model. It's responsible for managing the execution of asynchronous code. JavaScript is single-threaded, meaning it can only execute one task at a time, but it can perform asynchronous operations like I/O in a non-blocking way.

Explanation: The Event Loop constantly checks the call stack (the place where synchronous code is executed) and the message queue (where asynchronous tasks

are queued). If the call stack is empty, the Event Loop takes the first task from the message queue and pushes it onto the call stack for execution. This process continues, ensuring that asynchronous tasks are processed without blocking the main thread.

2. Question: Explain closures in JavaScript. Provide an example.

Answer: A closure is a function that "closes over" variables from its outer (enclosing) function scope. It retains access to those variables even after the outer function has completed.

Explanation: Here's an example:

```
function outer() {
  const outerVar = 'I am from the outer function';
  return function inner() {
    console.log(outerVar);
  };
}

const innerFunction = outer();
```

```
innerFunction(); // Outputs: "I am from the outer function"
```

In this example, the inner function closes over the outerVar variable, allowing it to access outerVar even after the outer function has finished executing.

3. Question: What is the "this" keyword in JavaScript, and how is its value determined?

Answer: The this keyword refers to the context in which a function is executed. Its value is determined dynamically at runtime based on how a function is called.

Explanation: The value of this can be:

In a function called as a method of an object, this refers to the object itself.
In a simple function call, this typically refers to the global object (e.g., window in a browser).
In a constructor function, this refers to the newly created object.
When using .call() or .apply(), you can explicitly set the value of this.

4. Question: What is a promise in JavaScript, and how does it differ from callbacks?

Answer: A promise is an object that represents the eventual completion (or failure) of an asynchronous operation. It provides a cleaner and more structured way to handle asynchronous tasks compared to traditional callback functions.

Explanation: Promises have two states: "fulfilled" (resolved) and "rejected" (failed). You can attach .then() and .catch() handlers to a promise to handle success and error cases. Promises help avoid callback hell and make asynchronous code more readable and maintainable.

5. Question: What is "hoisting" in JavaScript, and how does it work?

Answer: Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during compilation. However, only declarations are hoisted, not initializations.

Explanation: For example:

```
console.log(myVar); // Output: undefined
var myVar = 10;
```

In this code, myVar is hoisted to the top of the scope, but it's not yet assigned a value. That's why console.log(myVar) doesn't throw an error but logs undefined.

6. Question: Explain the concepts of "currying" and "partial application" in JavaScript.

Answer: Currying is the process of converting a function that takes multiple arguments into a series of functions that each take a single argument. Partial application is a similar concept where a function is partially applied to some of its arguments, producing a new function.

Explanation: Here's an example of currying:

```
const add = a => b => a + b;
const add5 = add(5);
const result = add5(10); // Result is 15
```

In this example, add is a curried function that takes two arguments. We partially apply it with 5, resulting in a new function add5 that takes a single argument, which adds 5 to it.

7. Question: What is the purpose of the "use strict" directive in JavaScript?

Answer: "use strict" is a pragma that enables strict mode, which helps catch common coding mistakes and "unsafe" actions in JavaScript. It enforces better coding practices and makes the code less error-prone.

Explanation: In strict mode, certain actions that were previously silent errors become explicit errors, like assigning values to undeclared variables or using reserved keywords. It encourages better coding practices and enhances performance in some cases.

8. Question: Explain the concept of "prototypal inheritance" in JavaScript.

Answer: In JavaScript, objects can inherit properties and methods from other objects through their prototype chain. Prototypal inheritance is a mechanism where an object can inherit from another object, serving as the basis for creating more complex objects.

Explanation: When a property or method is accessed on an object, JavaScript will first look on the object itself. If it's not found, it will traverse up the prototype chain to find it in the object's prototype, and so on. This allows for a flexible and dynamic way to create and share functionality among objects.

9. Question: What are generators in JavaScript, and how do they work?

Answer: Generators are a type of function that can be paused and resumed. They allow you to control the flow of execution manually, which is useful for asynchronous operations and creating iterators.

Explanation: Here's a simple example:

```
function* countUpTo(n) {
  let count = 1;
  while (count <= n) {
    yield count;
    count++;
  }
}

const counter = countUpTo(5);
console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
// ...
```

The yield keyword allows the function to pause and resume, making it handy for asynchronous tasks and lazy evaluation.

10. Question: What is the difference between "call" and "apply" in JavaScript?

Answer: Both `call` and `apply` are methods used to invoke functions with a specific context (the value of `this`) and a set of arguments. The key difference is in how arguments are passed to the function.

Explanation:

- `call`: Arguments are passed as a comma-separated list. For example, `func.call(context, arg1, arg2, arg3)`.
- `apply`: Arguments are passed as an array or array-like object. For example, `func.apply(context, [arg1, arg2, arg3])`.

Which one to use depends on the function and how its arguments are structured. These advanced JavaScript questions cover a range of important topics. Understanding these concepts can greatly improve your ability to work with JavaScript effectively.