Asynchronous Programming, Callbacks, Promises, Async/Await, AJAX & Fetch API

Asynchronous Programming & Callbacks

1. What is Asynchronous Programming?

JavaScript is single-threaded, meaning it executes one operation at a time on a single main thread. Asynchronous programming allows non-blocking execution where time-consuming operations (like network requests, timers, or file reading) run in the background, enabling the main thread to continue processing other user interactions without freezing the application.

2. What are Callbacks?

A callback is a function passed as an argument to another function, which is then invoked inside the outer function once an asynchronous operation completes. While foundational, excessive nesting of callbacks leads to 'Callback Hell' (or the Pyramid of Doom), making code unreadable, brittle, and difficult to debug.

3. Program: Callback Demonstration & Callback Hell

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Callback Demo</title>
</head>
<body>
  <script>
    function stepOne(callback) {
      setTimeout(() => {
        console.log('Step 1 complete');
        callback();
      }, 1000);
    }

    function stepTwo(callback) {
      setTimeout(() => {
        console.log('Step 2 complete');
        callback();
      }, 1000);
    }

    // Nested callbacks (Callback Hell)
    stepOne(() => {
```

```
        stepTwo(() => {
          console.log('All steps completed successfully.');
        });
      });
    </script>
</body>
</html>
```

PART 2: Promises

1. What is a Promise?

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It provides a cleaner, more robust alternative to callbacks, eliminating callback hell through chaining via .then(), .catch(), and .finally().

2. The Three States of a Promise

- Pending: Initial state, neither fulfilled nor rejected.
- Fulfilled (Resolved): The operation completed successfully, and the promise has a value.
- Rejected: The operation failed, and the promise has a reason (error).

3. Program: Creating and Consuming Promises

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Promise Demo</title>
</head>
<body>
  <script>
    const checkInventory = (item) => {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          if (item === 'Laptop') {
            resolve({ item: 'Laptop', price: 1200, status: 'In Stock' });
          } else {
            reject(new Error('Item is out of stock.'));
          }
        }, 1000);
      });
    };

    checkInventory('Laptop')
      .then(result => {
        console.log('Success:', result);
      })
```

```
        .catch(error => {
          console.error('Error:', error.message);
        })
        .finally(() => {
          console.log('Inventory check finalized.');
        });
  </script>
</body>
</html>
```

PART 3: Async/Await

1. What is Async/Await?

Introduced in ES2017, async/await is syntactic sugar built on top of Promises. The 'async' keyword transforms a function into a asynchronous function that always returns a promise. The 'await' keyword pauses the execution of the async function until the promise is settled, making asynchronous code look and behave like synchronous code.

2. Error Handling with Try...Catch

Because async/await code reads synchronously, runtime and rejection errors are caught cleanly using standard try...catch blocks, greatly simplifying complex asynchronous control flows.

3. Program: Async/Await Implementation

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Async/Await Demo</title>
</head>
<body>
  <script>
    const fetchUserData = () => {
      return new Promise((resolve) => {
        setTimeout(() => resolve({ id: 1, name: 'Asha Sharma' }), 1000);
      });
    };

    async function getUserProfile() {
      try {
        console.log('Fetching user profile...');
        const user = await fetchUserData();
        console.log('User retrieved:', user);
      } catch (error) {
        console.error('Failed to fetch user:', error);
```

```
        } finally {
            console.log('Process complete.');
        }
    }

    getUserProfile();
  </script>
</body>
</html>
```

PART 4: AJAX & Fetch API

1. What is AJAX?

AJAX (Asynchronous JavaScript and XML) is a technique for creating fast and dynamic web pages by exchanging data asynchronously with a web server behind the scenes, allowing parts of a web page to update without reloading the entire page. Traditionally implemented using the XMLHttpRequest object.

2. What is the Fetch API?

The Fetch API is a modern, promise-based interface implemented in ES6+ for fetching resources across the network. It provides a more powerful, flexible, and cleaner replacement for XMLHttpRequest.

3. Program: Fetch API GET and POST Requests

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Fetch API Demo</title>
</head>
<body>
  <button id="loadBtn">Fetch Data</button>
  <div id="result"></div>
  <script>
    document.getElementById('loadBtn').addEventListener('click', async () => {
      try {
        // GET Request
        const response = await fetch('https://jsonplaceholder.typicode.com/users/1');
        if (!response.ok) throw new Error('Network response was not ok');
        const data = await response.json();
        document.getElementById('result').innerHTML = `<h3>User:
${data.name}</h3><p>Email: ${data.email}</p>`;
      } catch (error) {
        console.error('Fetch error:', error);
      }
```

```
      });
    </script>
</body>
</html>
```