Q1. Define XMLHttpRequest object, and How is it used in AJAX?

The XMLHttpRequest (XHR) object is a built-in JavaScript object used to interact with web servers and retrieve data without reloading the webpage. It is fundamental to AJAX (Asynchronous JavaScript and XML), allowing web applications to send HTTP requests to the server and handle the server's response asynchronously.

AJAX uses XMLHttpRequest to:

- **Send data to the server asynchronously.**
- **Retrieve data (often in formats like JSON or XML) from the server without requiring a page reload.**
- **Dynamically update parts of a webpage based on the server's response.**

Basic Steps in Using XMLHttpRequest:

1. **Create a new XMLHttpRequest object.**
2. **Open a request using the open() method.**
3. **Send the request using the send() method.**
4. **Handle the server's response in an event handler (usually with onload or onreadystatechange).**

Q2. Write a JavaScript program to make an AJAX request using the XMLHttpRequest object to fetch data from a remote API (https://jsonplaceholder.typicode.com/users) and log a list of user names from the response to the browser console and output.

```
// Create a new XMLHttpRequest object

const xhr = new XMLHttpRequest();


// Configure the request

xhr.open('GET', 'https://jsonplaceholder.typicode.com/users', true);


// Set up the response handler

xhr.onload = function() {

  if (xhr.status === 200) {

    // Parse the JSON response

    const users = JSON.parse(xhr.responseText);


    // Log each user's name to the console

    users.forEach(user => {
```

```
    console.log(user.name);
  });
 } else {
  console.error('Request failed with status:', xhr.status);
 }
};

// Send the request
xhr.send();
```

Explanation:

- **The XMLHttpRequest object is used to send a GET request to fetch users from the specified API.**
- **The onload event handler processes the response, checking if the status code is 200 (OK).**
- **The JSON response is parsed, and each user's name is logged to the console.**

Q3. What is the same-origin policy in the context of AJAX requests, and how can you work around it?

The same-origin policy is a security measure implemented in web browsers to prevent malicious websites from making requests to a different domain without the user's knowledge. This policy restricts web pages from making requests to a domain other than the one from which the page was loaded (the "origin").

Working Around the Same-Origin Policy:

- **Cross-Origin Resource Sharing (CORS): This is a protocol that allows web servers to specify who can access their resources. If the server supports CORS, it sends specific HTTP headers (like Access-Control-Allow-Origin) to indicate which domains are allowed to access its resources.**
- **JSONP (JSON with Padding): This is an older technique that involves injecting a <script> tag to make cross-origin requests. JSONP only supports GET requests and is now considered outdated due to security concerns.**
- **Proxy Server: A server-side script can act as a proxy to make the request to another domain and then pass the data back to the client, effectively bypassing the same-origin policy.**

Q4. Compare Promises to callbacks and discuss why Promises are preferred for managing asynchronous code.

Callbacks:

- **A callback is a function passed into another function as an argument, which is then executed when the asynchronous operation completes.**

- **Problems with callbacks:**
    - **Callback Hell: Nested callbacks can become difficult to manage, leading to deep indentation and unreadable code.**
    - **Error Handling: Error handling in callbacks can be cumbersome because errors have to be passed manually through each level.**

Promises:

- **A Promise represents the result of an asynchronous operation and is either resolved (success) or rejected (failure).**
- **Advantages of Promises:**
    - **Chaining: Promises allow for method chaining using .then() and .catch(), making asynchronous code more readable and manageable.**
    - **Error Handling: Promises provide a cleaner way to handle errors through .catch() or using try/catch in async/await.**
    - **Avoid Callback Hell: Promises flatten the nested structure, allowing for easier-to-read asynchronous code.**

Example of Promise vs Callback:

- **Callback Example:**
- **function fetchData(callback) {**
- **setTimeout(() => {**
- **callback(null, "Data fetched");**
- **}, 1000);**
- **}**
-
- **fetchData((error, data) => {**
- **if (error) {**
- **console.error(error);**
- **} else {**
- **console.log(data);**
- **}**
- **});**
- **Promise Example:**
- **function fetchData() {**
- **return new Promise((resolve, reject) => {**

```
    setTimeout(() => {
      resolve("Data fetched");
    }, 1000);
  });
}

fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Q5. List and briefly explain some common Browser APIs available to JavaScript developers.

1. **DOM (Document Object Model) API: Provides methods and properties to interact with and manipulate the structure, content, and styles of a webpage.**
    - **Example: document.getElementById(), document.querySelector(), element.innerHTML**
2. **Fetch API: Provides a modern way to make HTTP requests to retrieve resources or send data.**
    - **Example: fetch('https://api.example.com/data')**
3. **LocalStorage and SessionStorage: Provides a way to store data in the browser persistently (localStorage) or for the session (sessionStorage).**
    - **Example: localStorage.setItem('key', 'value')**
4. **Geolocation API: Allows access to the device's geographic location.**
    - **Example: navigator.geolocation.getCurrentPosition()**
5. **Canvas API: Used for drawing graphics, creating animations, and manipulating images directly in the browser.**
    - **Example: const ctx = canvas.getContext('2d');**
6. **Web Storage API: Includes localStorage and sessionStorage for storing data in the browser.**
    - **Example: localStorage.getItem('username')**
7. **WebSockets API: Provides a way to open a persistent connection between the client and the server for real-time communication.**
    - **Example: const socket = new WebSocket('ws://example.com');**
8. **Notification API: Allows web pages to display system notifications to the user.**
    - **Example: Notification.requestPermission(), new Notification('Title', { body: 'Message' })**

9. **File API: Allows JavaScript to interact with files selected by the user (e.g., file uploads).**
    - **Example: input.files[0]**

Q6. Describe the purpose and usage of the localStorage and sessionStorage APIs in web development. Give suitable examples for each.

localStorage:

- **localStorage allows data to be stored persistently across browser sessions. The data persists even after the browser is closed and reopened.**
- **Usage: Store user preferences, login state, etc.**
- **Example:**
- **// Save data**
- **localStorage.setItem('username', 'Alice');**
-
- **// Retrieve data**
- **const username = localStorage.getItem('username');**
- **console.log(username); // Output: Alice**

sessionStorage:

- **sessionStorage allows data to be stored for the duration of the page session (data is cleared when the tab or browser is closed).**
- **Usage: Store data that should only persist for the current session (e.g., form data, temporary settings).**
- **Example:**
- **// Save data for the session**
- **sessionStorage.setItem('cartItems', JSON.stringify([1, 2, 3]));**
-
- **// Retrieve data for the session**
- **const cartItems = JSON.parse(sessionStorage.getItem('cartItems'));**
- **console.log(cartItems); // Output: [1, 2, 3]**

Both localStorage and sessionStorage store data as key-value pairs, but their lifetimes differ: localStorage is persistent until manually cleared, while sessionStorage is cleared when the page session ends.