Week 9: Web Development

1. Promises

Promises in JavaScript provide a way to handle asynchronous operations. Here's a concise explanation:

- **Definition:** Promises represent the eventual completion or failure of an asynchronous operation.

- **States:** Promises have three states: pending, fulfilled, or rejected.

- **Creation:** Created using the `Promise` constructor, which takes an executor function with `resolve` and `reject` parameters.

- **Consumption:** Result is consumed using `then()` method with success and error callbacks, or `catch()` method for error handling.

- **Chaining:** Supports chaining multiple asynchronous operations using `then()` method.

- **Error Handling:** Errors are caught using `catch()` method or by chaining a `catch` handler.

Promises offer a concise and powerful way to work with asynchronous code, improving readability and error handling.

Example :

```
function getData() {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation (e.g., fetching data from an API)
    setTimeout(() => {
      const data = [1, 2, 3, 4, 5];
```

```
      // Simulating successful completion
      resolve(data);

      // Simulating failure
      // reject(new Error('Failed to fetch data'));
    }, 2000); // Simulating a delay of 2 seconds
  });
}

// Consuming the Promise
getData()
  .then(data => {
    console.log('Data received:', data);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

2. SetTimeout

The setTimeout() method is used to call a function after a certain period of time. The setInterval() Javascript method is used to call a function repeatedly at a specified interval of time.

```
 console.log('Starting setTimeout example...');
setTimeout(() => {
  console.log('This message will be displayed after 2 seconds.');
}, 2000); // 2000 milliseconds = 2 seconds
console.log('setTimeout example scheduled.');
```

3. SetInterval

```
let count = 0;
const intervalId = setInterval(() => {
  count++;
  console.log(`Interval count: ${count}`);
  if (count === 5) {
    clearInterval(intervalId); // Stop the interval after 5 iterations
    console.log('Interval stopped after 5 iterations.');
  }
}, 1000); // 1000 milliseconds = 1 second
console.log('setInterval example started.');
```

4. Storage

Here's a brief comparison of Local Storage, Session Storage, and Cookie Storage in web development:

1. **Local Storage:**
   - **Scope:** Data persists even after the browser is closed and across browser sessions.
   - **Storage Limit:** Typically around 5-10 MB per origin, varies by browser.
   - **Usage:** Suitable for storing larger amounts of data that need to be accessed across sessions, such as user preferences, cached data, or application state.
   - **Access:** Accessed using the `localStorage` object in JavaScript.
   - **Expiration:** Data remains until explicitly cleared by the user or removed programmatically.

2. **Session Storage:**
   - **Scope:** Data persists only for the duration of the page session. It is cleared when the page session ends (i.e., when the browser tab is closed).
   - **Storage Limit:** Similar to Local Storage, typically around 5-10 MB per origin, varies by browser.
   - **Usage:** Suitable for storing temporary data or session-specific information that is only needed for the current browsing session.
   - **Access:** Accessed using the `sessionStorage` object in JavaScript.
   - **Expiration:** Data is automatically cleared when the browser tab or window is closed.

3. **Cookie Storage:**
   - **Scope:** Data is stored in small text files (cookies) on the client's device and is sent with every HTTP request to the server.
   - **Storage Limit:** Each cookie has a size limit of around 4 KB and a limit on the number of cookies per domain (typically around 20-50).
   - **Usage:** Often used for storing small amounts of data such as user authentication tokens, session identifiers, or user preferences.
   - **Access:** Accessed and manipulated using document.cookie property or libraries/frameworks.
   - **Expiration:** Cookies can have an expiration date, after which they are automatically deleted. They can also be session cookies, which are deleted when the browser is closed.

Local Storage Example

```
// Storing data in Local Storage
localStorage.setItem('username', 'john_doe');
localStorage.setItem('isLoggedIn', true);

// Retrieving data from Local Storage
const username = localStorage.getItem('username');
const isLoggedIn = localStorage.getItem('isLoggedIn');

console.log('Username:', username);
console.log('Is Logged In:', isLoggedIn);

// Updating data in Local Storage
localStorage.setItem('isLoggedIn', false);

// Removing data from Local Storage
localStorage.removeItem('username');
```

```
// Clearing all data from Local Storage
localStorage.clear();
```

5. Fetch API

```
// Make a GET request to fetch user data from JSONPlaceholder API
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    // Check if the response is successful (status code in the range 200-299)
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    // Parse the JSON response
    return response.json();
  })
  .then(data => {
    // Log the fetched user data
    console.log('Fetched user data:', data);
  })
  .catch(error => {
    // Handle any errors that occurred during the fetch operation
    console.error('Error fetching user data:', error);
  });
```

6. Async Await

```
  - Introduced in ES2017 for asynchronous programming in JavaScript.
  - Simplifies writing and understanding asynchronous code.
  - Utilizes the `async` and `await` keywords.
  - Async functions always return a promise.
  - `async` keyword declares a function as asynchronous.
  - `await` keyword pauses function execution until a promise is settled.
  - Enables writing asynchronous code that looks synchronous.
  - Simplifies error handling using try-catch blocks.
  - Enhances code readability and maintainability.
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Async/Await Example</title>
</head>
<body>
    <button id="fetchButton">Fetch Data</button>
    <script src="script.js"></script>
</body>
```

```
</html>

function fetchData() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve("Data fetched successfully");
        }, 2000);
    });
}

async function fetchDataWithTimeout() {
    console.log("Fetching data...");
    try {
        const data = await fetchData();
        console.log("Data:", data);
    } catch (error) {
        console.error("Error fetching data:", error);
    }
}

document.getElementById("fetchButton").addEventListener("click", () => {
    fetchDataWithTimeout();
});
```

- Class Tasks

Lab Tasks

1. Auth with Storage:
- Implement a user authentication system using Local Storage:
- Create functions login(username, password) and logout() to handle user login and logout.
- Store the user's login status and credentials securely in Local Storage.
- Add a check to see if the user is already logged in when the page loads and display appropriate UI elements accordingly.

2. Party Planning

Task: Create a function to generate invitation messages for a party, incorporating personalized details for each guest.

Requirements:

The party organizer has a list of guests invited to the party, stored as an array of objects.
Each guest object contains information such as their name, age, and RSVP status.
The function should accept the list of guest objects as its argument.
Use the spread operator to handle the array of guest objects.
Use the rest parameter to handle any additional details to be included in the invitation message.
Generate an invitation message for each guest, including their name and any additional details provided.

Return an array of invitation messages.