Assignment Questions 5

Q.1 What's the difference between Synchronous and Asynchronous?

Ans -

Synchronous -

In synchronous operations codes are executed one after another, in a sequential manner. When a synchronous operation is initiated, the program waits for it to complete before moving on to the next operation. During this time, the program is blocked, and no other code can be executed. Synchronous operations can sometimes cause delays, especially if they involve time-consuming tasks or external resources.

Asynchronous -

asynchronous operations allow the program to continue executing while the task is being processed in the background. Instead of waiting for the operation to complete, the program registers a callback function and continues with its execution. Once the asynchronous operation finishes, it triggers the callback to handle the result. As a result, the program doesn't get blocked, and other tasks can continue simultaneously.

Q.2 What are Web Apis ?

Ans -

Web APIs (Application Programming Interfaces) are sets of rules and protocols that allow different software applications to communicate and interact with each other. In the context of web development, Web APIs are specifically designed to enable communication between web browsers and web servers, or between different web-based services.

Q.3 Explain SetTimeOut and setInterval ?

Ans -

SetTimeOut -

The setTimeout function is used to execute a piece of code or a function after a specified amount of time (in milliseconds). It takes two arguments: the function to be executed and the delay time.

E.g-

```
setTimeout(() => {
      console.log("setTimeout method");
   }, 2000);
```

setInterval -

The setInterval function is used to repeatedly execute a piece of code or a function at a fixed time interval. It takes two arguments: the function to be executed and the interval time.

E.g-

```
  setInterval(() => {
     console.log("setInterval function");
   }, 2000);
```

Q.4 How can you handle Async code in JavaScript ?

Ans -

In JavaScript, there are several approaches to handle asynchronous code effectively.

- Callbacks
- Promises
- async/await

Q.5 What are Callbacks & Callback Hell ?

Ans -

Callbacks -

Callbacks are functions that are passed as arguments to other functions and are intended to be called at a later time or after the completion of an asynchronous operation. Callbacks are a common pattern in JavaScript to handle asynchronous code execution.

E.g-

```
function processData(callback) {
  const data = "Some data";
  callback(data);
}

const handleData = (data) => {
  console.log("Data received:", data);
};

processData(handleData);
```

Callback hell -

callbacks extensively for handling asynchronous operations can lead to a phenomenon called "Callback Hell" or "Pyramid of Doom." Callback Hell occurs when multiple asynchronous operations are nested within each other, resulting in deeply nested callback functions. This can make the code difficult to read, understand, and maintain.

E.g-

```
asyncOperation1(function(result1) {
  asyncOperation2(result1, function(result2) {
    asyncOperation3(result2, function(result3) {
      // More nested callbacks...
    });
  });
});
```

Q.6 What are Promises & Explain Some Three Methods of Promise

Ans -

Promises are a way to manage the flow of asynchronous code and handle the results or errors that occur during the execution of the operation. Promises provide a cleaner and more organized approach to dealing with asynchronous tasks compared to using callbacks.

There are three main methods of promise -

Then() :- The then() method is used to handle the fulfilled state of a promise.

catch() :- The catch() method is used to handle the rejected state of a promise.

finally() :- The finally() method is used to specify a callback function that should be executed regardless of whether the promise is fulfilled or rejected.

```
// Example asynchronous function that returns a promise
function fetchData() {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const data = 'Example data';
      // Simulating a successful operation
      resolve(data);
```

```
      // Uncomment the line below to simulate an error
      // reject('Error occurred');
    }, 2000);
  });
}

// Using promises and their methods
fetchData()
  .then((result) => {
    console.log('Fulfilled:', result);
    // You can return a new value or a new promise from here
    // to chain additional `then()` or `catch()` methods
    // return doSomethingWithResult(result);
  })
  .catch((error) => {
    console.log('Rejected:', error);
  })
  .finally(() => {
    console.log('Finally block executed');
    // Any cleanup or resource deallocation tasks can be performed
here
  });
```

Q.7 What's async & await Keyword in JavaScript

Ans -

In JavaScript, the async and await keywords are used to simplify the handling of asynchronous operations and make asynchronous code look more like synchronous code. They provide a more straightforward and sequential way to write asynchronous code without relying on callbacks or chaining promises.

async - The async keyword is used to define an asynchronous function. It allows the function to use the await keyword inside its body.

await - The await keyword is used to pause the execution of an asynchronous function until a promise is fulfilled, and it can only be used inside an async function

```
// Example asynchronous function that returns a promise
function fetchData() {
```

```
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const data = 'Example data';
      // Simulating a successful operation
      resolve(data);
      // Uncomment the line below to simulate an error
      // reject('Error occurred');
    }, 2000);
  });
}

// Example async function using await
async function fetchDataAsync() {
  try {
    // Await the completion of the fetchData() promise
    const result = await fetchData();
    console.log('Fulfilled:', result);
    // You can perform further operations here
    // const processedData = await processResult(result);
    // console.log('Processed data:', processedData);
  } catch (error) {
    console.log('Rejected:', error);
  } finally {
    console.log('Finally block executed');
    // Any cleanup or resource deallocation tasks can be performed
here
  }
}

// Calling the async function
fetchDataAsync();
```

Q.8 Explain Purpose of Try and Catch Block & Why do we need it?

Ans -

try() and **catch()** block is used to handle the error while executing the code. If we know that there can be an error in the code then we wrap it in the try() block and then handle it using catch() block.

E.g-

```
try{
   // code
} catch(err){
   console.log(err);
}
```

Q.9 Explain fetch?

Ans -

The **fetch()** function is a built-in web API in JavaScript used to make HTTP requests to a server and retrieve data. It provides a modern and straightforward way to fetch resources asynchronously over the network.

```
fetch('https://api.example.com/data')
  .then(response => {
    // Check if the request was successful
    if (response.ok) {
      // Convert the response to JSON
      return response.json();
    } else {
      throw new Error('Network response was not ok.');
    }
  })

  .then(data => {
    // Process the fetched data
    console.log(data);
  })
  .catch(error => {
    // Handle any errors that occurred during the fetch
    console.log('Error:', error.message);
  });
```

Q.9 How do you define an asynchronous function in JavaScript using async/await?

Ans -

async factorial(n) {

```
    if(n == 0){
        return 1;
    } else {
        return n * await factorial(n - 1);
    }
}
```