Day 5 → JavaScript Asynchronous Programming:

1. Introduction to Asynchronous Programming:
Asynchronous programming allows JavaScript to handle time-consuming operations without blocking the execution of other tasks. It's particularly useful for tasks like fetching data from an API, reading and writing files, or making database queries. In JavaScript, asynchronous programming is achieved using callbacks, Promises, and async/await.

2. Callback Functions:
Callback functions are a traditional way of handling asynchronous operations in JavaScript. A callback is a function passed as an argument to another function and gets executed when the asynchronous task completes. Here's an example:

```javascript
function fetchData(callback) {
  setTimeout(function() {
    const data = 'Hello, world!';
    callback(data);
  }, 2000);
}

function processData(data) {
  console.log(data);
}

fetchData(processData); // Output: Hello, world!
```

```

In this example, the `fetchData` function simulates an asynchronous operation with a 2-second delay. The `processData` function is the callback function that gets executed when the data is fetched.

3. Promises:
   Promises provide a more structured and readable approach to handling asynchronous operations. A Promise represents the eventual completion (or failure) of an asynchronous operation and allows you to attach callbacks using `.then()` and `.catch()`. Here's an example:

```javascript
function fetchData() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      const data = 'Hello, world!';
      resolve(data);
    }, 2000);
  });
}

fetchData()
  .then(function(data) {
    console.log(data);
  })
  .catch(function(error) {
    console.log(error);
```
```

```
  });
```

In this example, the `fetchData` function returns a Promise that resolves with the fetched data after a 2-second delay. The `.then()` callback handles the resolved value, and the `.catch()` callback handles any errors.

4. Async/Await:
   Async/await is a modern approach to asynchronous programming that simplifies the syntax and makes code more readable. It allows you to write asynchronous code in a synchronous manner. Here's an example:

```javascript
function fetchData() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      const data = 'Hello, world!';
      resolve(data);
    }, 2000);
  });
}

async function processData() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (error) {
    console.log(error);
```

```
  }
}

processData();
```

In this example, the `processData` function is declared as `async` and uses the `await` keyword to wait for the Promise to resolve before proceeding. The `try/catch` block handles both resolved values and errors.

Hands on project:

```
<!DOCTYPE html>
<html>

<head>
    <title>Asynchronous Task Manager</title>
</head>

<body>
    <h1>Asynchronous Task Manager</h1>

    <input type="text" id="taskInput" placeholder="Enter task">
    <button id="button" onclick="addTask()">Add Task</button>

    <ul id="taskList"></ul>

    <script>
        let button = document.getElementById('button')
        function addTaskToList(task) {
            const taskList = document.getElementById('taskList');
            const li = document.createElement('li');
```

```
            li.textContent = task;
            taskList.appendChild(li);
        }

        function clearTaskInput() {
            document.getElementById('taskInput').value = '';
        }

        function showLoader(){

            button.textContent = 'Loading...';
            button.disabled = true;
        }
        function hideLoader(){
            button.textContent = 'Add Task';
            button.disabled = false;
        }

        function simulateAPICall(task) {

            return new Promise((resolve, reject) => {
                // Simulating an API call with a 2-second delay
                showLoader()
                setTimeout(function () {
                    if (task.length < 10) {
                        resolve(task);
                    } else {
                        reject('Task length exceeds the limit');
                    }
                    hideLoader()
                }, 2000);
            });
        }

        function addTask() {
            const taskInput = document.getElementById('taskInput');
            const task = taskInput.value;

            simulateAPICall(task)
                .then(function (resolvedTask) {
```

```
                    addTaskToList(resolvedTask);
                    clearTaskInput();
                })
                .catch(function (error) {
                    alert(error)
                    console.log('Error:', error);
                });
        }

    </script>
</body>

</html>
```