Here's the structured content for **Day 6: Introduction to Asynchronous JavaScript**, including **theory, coding examples, and explanations** for **Callbacks, Promises, and Async/Await**.

Day 6: Introduction to Asynchronous JavaScript

1. Understanding Asynchronous JavaScript

JavaScript is **single-threaded**, meaning it executes code **line by line**. However, some operations like **network requests, file reading, and timers** take time. To avoid blocking the execution, JavaScript uses **asynchronous programming**.

There are three main ways to handle **asynchronous operations** in JavaScript:

1. **Callbacks** (Older approach)
2. **Promises** (Improved approach)
3. **Async/Await** (Modern and cleaner approach)

---

2. Callbacks in JavaScript

A **callback function** is a function **passed as an argument** to another function. It is executed after the completion of an asynchronous task.

2.1 Example of Callbacks

```
function fetchData(callback) {
    console.log("Fetching data...");
    setTimeout(() => {
        let data = { name: "Alice", age: 25 };
        callback(data);
    }, 2000);
}


function displayData(data) {
    console.log("Data received:", data);
}
```

```
// Calling fetchData with displayData as a callback

fetchData(displayData);
```

2.2 Problems with Callbacks ("Callback Hell")

When multiple callbacks are nested, the code becomes hard to read and maintain. Example of **callback hell**:

```
setTimeout(() => {
    console.log("Step 1");
    setTimeout(() => {
        console.log("Step 2");
        setTimeout(() => {
            console.log("Step 3");
        }, 1000);
    }, 1000);
}, 1000);
```

☠ **Solution**: Use **Promises** instead of callbacks.

---

3. Promises in JavaScript

A **Promise** is an object that represents the **eventual completion (or failure)** of an asynchronous operation.

3.1 Promise States

A Promise can be in one of the following states:

- **Pending** (Initial state)
- **Fulfilled** (Successfully completed)
- **Rejected** (Failed)

3.2 Creating and Using Promises

```
function fetchData() {
    return new Promise((resolve, reject) => {
        console.log("Fetching data...");
        setTimeout(() => {
            let success = true;
            if (success) {
                resolve({ name: "Alice", age: 25 });
            } else {
                reject("Failed to fetch data");
            }
        }, 2000);
    });
}

// Handling Promises using then and catch
fetchData()
    .then((data) => {
        console.log("Data received:", data);
    })
    .catch((error) => {
        console.error("Error:", error);
    });
```

3.3 Promise Methods

- **then()** → Executes when the Promise is resolved.
- **catch()** → Executes when the Promise is rejected.
- **finally()** → Always executes, regardless of success or failure.

```
fetchData()
    .then((data) => console.log("Success:", data))
    .catch((error) => console.error("Error:", error))
    .finally(() => console.log("Operation completed"));
```

---

4. Async/Await: The Modern Approach

Async/Await provides a cleaner way to work with Promises and makes asynchronous code look synchronous.

4.1 Writing Async Functions

```
async function getData() {
    console.log("Fetching data...");
    let response = await fetchData(); // Waiting for the promise to resolve
    console.log("Data received:", response);
}

getData();
```

4.2 Handling Errors with Try-Catch

```
async function getData() {
    try {
        console.log("Fetching data...");
        let response = await fetchData();
        console.log("Data received:", response);
    } catch (error) {
        console.error("Error:", error);
    }
}

getData();
```

---

5. Summary

Method	Description
Callbacks	Basic way to handle async code but can lead to **callback hell**.
Promises	A more structured way with **then, catch, finally**.
Async/Await	Modern, cleaner syntax for handling Promises.
