## **Asynchronous JavaScript**

JavaScript is **single-threaded**, meaning it executes code line by line. However, it uses **asynchronous programming** to handle time-consuming tasks like API calls, file reading, and timers without blocking execution.

### **1️⃣ Callback Functions**

A **callback function** is a function passed as an argument to another function and executed later.

#### **Example: Callback Function**

function fetchData(callback) {

setTimeout(() => {

console.log("Data fetched!");

callback(); // Execute the callback function after fetching data

}, 2000);

}

function processData() {

console.log("Processing data...");

}

fetchData(processData);

🔹 **Problem with Callbacks**: When multiple callbacks are nested, it leads to **callback hell**, making code hard to read.

### **2️⃣ Promises & .then()**

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

#### **Example: Using Promises**

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

let success = true;

if (success) resolve("Data fetched!");

else reject("Failed to fetch data.");

}, 2000);

});

}

fetchData()

.then(response => console.log(response)) // Success case

.catch(error => console.error(error)); // Error handling

✅ **Advantage**: Promises provide a **cleaner way** to handle async operations and avoid callback hell.

### **3️⃣ Async/Await & Error Handling**

The async keyword makes a function return a Promise, while await pauses execution until the Promise is resolved.

Here, "pause" means that the function execution is temporarily suspended, and its context is removed from the call stack. Once the Promise is resolved, the function resumes execution by retrieving its context from the microtasks queue.

#### **Example: Async/Await**

async function fetchData() {

try {

let response = await new Promise((resolve) =>

setTimeout(() => resolve("Data fetched!"), 2000)

);

console.log(response);

} catch (error) {

console.error(error);

}

}

fetchData();

✅ **Benefit**: Makes async code look **synchronous and more readable**.

### **4️⃣ Event Loop & Microtasks Queue**

JavaScript handles asynchronous operations using the **Event Loop**, which manages the **Call Stack**, **Callback Queue**, and **Microtasks Queue**.

#### **Execution Order in JavaScript's Event Loop:**

1. **Call Stack** runs synchronous code.
2. **Microtasks Queue** (Promises & MutationObservers) executes before Callback Queue.
3. **Callback Queue** (setTimeout, setInterval, DOM events) runs after Microtasks.

#### **Example: Event Loop Execution Order**

console.log("Start");

setTimeout(() => console.log("Timeout Callback"), 0);

Promise.resolve().then(() => console.log("Promise Resolved"));

console.log("End");

📌 **Output Order**:

Start

End

Promise Resolved

Timeout Callback

🔹 **Why?** The Promise executes first since **Microtasks Queue has higher priority** than the Callback Queue.

### **Best Practices**

✔ **Use Promises or Async/Await** instead of callbacks for better readability.  
✔ **Handle errors** using .catch() for Promises or try/catch for Async/Await.  
✔ **Understand the Event Loop** to predict execution order in JavaScript.