**Asynchronous JavaScript**

JavaScript is **single-threaded**, meaning it can do only **one thing at a time**.  
But what if you need to fetch data from a server or wait for a timer?  
That’s where **asynchronous programming** comes in — it allows JavaScript to **continue running other code** while waiting for a slow task (like fetching data) to finish.

**1. Synchronous vs Asynchronous**

**Synchronous (one after another):**

console.log("Start");

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

console.log("End");

### Output:

Start

Processing...

End

**Asynchronous (non-blocking):**

console.log("Start");

setTimeout(() => {

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

}, 2000);

console.log("End");

### Output:

Start

End

Processing... (after 2 seconds)

The setTimeout() function is **asynchronous** — it allows the rest of the code to continue while it waits.

**2. Callbacks**

A callback function is a function passed as an argument into another function. The receiving function can then "call back" and execute the passed function at an appropriate time, which can be either immediately (synchronously) or after some kind of routine or action is completed (asynchronously).

function greet(name, callback) {

console.log("Hi " + name);

callback();

}

function sayBye() {

console.log("Goodbye!");

}

greet("Raj", sayBye);

### Output:

Hi Raj

Goodbye!

⚠️ But too many nested callbacks cause **“callback hell”**, which looks messy:

doTask1(() => {

doTask2(() => {

doTask3(() => {

console.log("All done!");

});

});

});

To solve this problem, modern JavaScript introduced **Promises** and **async/await** as cleaner and more readable alternatives for handling asynchronous operations

**3. Promises**

A **Promise** is an object that represents a task that will finish in the **future** (success or failure).

### Syntax:

let promise = new Promise((resolve, reject) => {

// async task

});

Example:

let task = new Promise((resolve, reject) => {

let success = true;

if (success) resolve("✅ Task completed!");

else reject("❌ Task failed!");

});

task

.then((message) => console.log(message)) // success

.catch((error) => console.log(error)) // failure

.finally(() => console.log("Task finished!"));

### Output:

✅ Task completed!

Task finished!

**4. Chaining Promises**

You can connect multiple .then() calls for sequential operations:

fetchData()

.then(processData)

.then(displayResult)

.catch(handleError);

Example:

new Promise((resolve) => resolve(5))

.then((num) => num \* 2)

.then((result) => console.log(result)); // Output: 10

**5. async / await (ES8 Feature)**

A cleaner, modern way to handle promises — looks synchronous but runs asynchronously.

async function fetchData() {

let promise = new Promise((resolve) => {

setTimeout(() => resolve("✅ Data received!"), 2000);

});

console.log("⏳ Waiting for data...");

let result = await promise; // wait until promise resolves

console.log(result);

}

fetchData();

### Output:

⏳ Waiting for data...

✅ Data received! (after 2 sec)

**6. Real Example — Fetching API Data**

The fetch() function is built-in for HTTP requests (returns a Promise).

async function getUser() {

try {

let response = await fetch("https://jsonplaceholder.typicode.com/users/1");

let data = await response.json();

console.log("User Name:", data.name);

} catch (error) {

console.log("❌ Error fetching data:", error);

}

}

getUser();

### Output:

User Name: Leanne Graham

## Summary

|  |  |  |
| --- | --- | --- |
| **Feature** | **Description** | **Example** |
| **Callback** | Function passed as argument | doTask(() => {}) |
| **Promise** | Represents future value | .then(), .catch() |
| **async/await** | Cleaner syntax for promises | await fetch() |
| **fetch()** | Makes API calls | fetch(url) |

**🧩 Mini Assignment**

1. Use setTimeout to print:
2. Loading...
3. Done!

after 2 seconds.

1. Create a promise that resolves if a number is positive and rejects if negative.
2. Use async/await to fetch data from  
   https://jsonplaceholder.typicode.com/posts/1  
   and display its title in the console.