1. Introduction to Asynchronous JavaScript

JavaScript is single-threaded, meaning it can execute one task at a time. However, real-world applications require handling multiple operations simultaneously, such as fetching data, reading files, or handling user interactions. This is where asynchronous programming comes into play.

Asynchronous programming allows JavaScript to perform non-blocking operations, meaning tasks can run in the background while the main thread continues executing code. This enhances performance and ensures a smooth user experience.

2. Synchronous vs. Asynchronous with Examples

Synchronous (Blocking)
Code executes line by line.

Each operation must complete before moving to the next.

```
console.log("First");
console.log("Second"); // Waits for "First" to finish
console.log("Third"); // Waits for "Second" to finish
```

output
First
Second
Third

Asynchronous (Non-Blocking)
Operations run in the background.

The program doesn't wait for slow tasks to complete.

```
console.log("First");
setTimeout(() => console.log("Second"), 1000); // Runs in the background
console.log("Third");
```

output

First
Third
Second (after 1 second)

3. How to Know if a Program is Asynchronous
APIs/Functions:

setTimeout, setInterval

fetch(), XMLHttpRequest

fs.readFile (Node.js)

Database operations (MongoDB, MySQL)

Behavior:
If a function takes a callback, returns a Promise, or uses async/await, it's likely asynchronous.

4. Event Loop, Call Stack, Microtask & Macrotask Queue

JavaScript is single-threaded, meaning it can execute only one operation at a time. However, JavaScript is also asynchronous, allowing operations like API requests, timers, and file reads to happen without blocking the main thread. The Event Loop is the mechanism that manages the execution of synchronous and asynchronous tasks efficiently.

Real-Life Example:

Think of a restaurant with a single chef. If the chef prepares one dish at a time and waits for ingredients (synchronous execution), service will be very slow. Instead, the chef starts cooking a dish and, while waiting for an ingredient, begins another dish. When the ingredient arrives, they return to the first dish. This is how the event loop works in JavaScript.

The Event Loop ensures that while JavaScript is waiting for an operation to complete (like fetching data from an API), it can continue executing other tasks.

2. Call Stack

The Call Stack is a data structure that keeps track of function execution in JavaScript. It follows the LIFO (Last In, First Out) principle: the last function that gets pushed onto the stack is the first one to be executed.

Example:
Example:

```
function first() {
console.log("First function");
}
```

```
function second() {
console.log("Second function");
}

first();
second();
```

Execution Flow:

first() is pushed onto the Call Stack → Executes → Pops off

second() is pushed onto the Call Stack → Executes → Pops off

The Call Stack is now empty.

Visualization:

```
| second() |
| first() |
---------- → Stack execution (LIFO - Last In First Out)
```

3. Microtask Queue vs. Macrotask Queue in JavaScript

JavaScript's asynchronous behavior is managed by two key queues: the Microtask Queue and the Macrotask Queue (also known as the Callback Queue). These queues determine the execution order of different asynchronous operations, ensuring JavaScript remains non-blocking while efficiently handling tasks.

Microtask Queue (Higher Priority)

Executed first before any macrotasks.

Processes tasks that are considered "immediate" in the event loop.

Includes:

Promises (.then(), .catch(), .finally())

MutationObserver (detects changes in the DOM)

queueMicrotask() (explicitly queues a microtask)

Runs continuously until the queue is empty before moving on to macrotasks.

Example: Microtask Execution

```
console.log("Start");

Promise.resolve().then(() => console.log("Microtask 1"));
Promise.resolve().then(() => console.log("Microtask 2"));

console.log("End");
```

Execution Order:

console.log("Start") → Printed first (Synchronous execution)

Promises are added to the Microtask Queue

console.log("End") → Printed second (Still synchronous)

Microtasks execute in order ("Microtask 1" and then "Microtask 2")

Output:

Start
End
Microtask 1
Microtask 2

Macrotask Queue (Lower Priority)

Executes after all microtasks have been completed.

Used for tasks that can be deferred until later.

Includes:

setTimeout, setInterval (delayed execution)

setImmediate (Node.js only, executes after I/O events)

I/O operations (file system, network requests, etc.)

UI rendering (browser repainting and rendering tasks)

Example: Macrotask Execution

```
console.log("Script Start");

setTimeout(() => console.log("Macrotask - setTimeout"), 0);

console.log("Script End");
```

Execution Order:

console.log("Script Start") → Printed first (Synchronous execution)

setTimeout is placed into the Macrotask Queue

console.log("Script End") → Printed second (Synchronous execution)

Macrotask Queue executes ("Macrotask - setTimeout" is printed)

Output:

```
Script Start
Script End
Macrotask - setTimeout
```

Microtask Queue vs. Macrotask Queue - Complete Example

```
console.log("Script Start");

setTimeout(() => console.log("Macrotask - setTimeout"), 0);

Promise.resolve().then(() => console.log("Microtask - Promise"));
```

```
console.log("Script End");
```

Execution Order Breakdown:

Synchronous Code Executes First:

console.log("Script Start") → Printed first

setTimeout() is placed in the Macrotask Queue

Promise.resolve().then() is placed in the Microtask Queue

console.log("Script End") → Printed second

Microtask Queue Executes Next:

"Microtask - Promise" is executed before any macrotasks

Macrotask Queue Executes Last:

"Macrotask - setTimeout" is printed after the microtasks

Final Output:
Script Start
Script End
Microtask - Promise
Macrotask - setTimeout

Real-Life Analogy

Think of the Microtask Queue as urgent, high-priority tasks (like replying to your boss's email), while the Macrotask Queue represents less urgent tasks (like checking social media).

You finish replying to your boss first (Microtask Queue completes first),

Then check social media (Macrotask Queue executes after Microtasks are empty).

Callbacks (Handling Async Operations)

A callback is a function passed as an argument to another function, which is executed later when an operation is completed. This was the primary way of handling asynchronous operations before Promises and async/await.

Example: Using Callbacks for Asynchronous Operations

```
function fetchData(callback) {
setTimeout(() => {
console.log("Data fetched from server");
callback("User Data");
}, 2000);
}

function processData(data, callback) {
setTimeout(() => {
console.log("Processing:", data);
```

```
callback("Processed Data");
}, 1500);
}

function displayData(data) {
console.log("Displaying:", data);
}

// Using callbacks to chain asynchronous operations
fetchData((userData) => {
processData(userData, (processedData) => {
displayData(processedData);
});
});
```

Problems with Callbacks (Callback Hell)

When multiple callbacks are nested within each other, it leads to callback hell (also known as the pyramid of doom), making code harder to read, debug, and maintain.

```
fetchData((data1) => {
processData(data1, (data2) => {
anotherAsyncFunction(data2, (data3) => {
yetAnotherAsyncFunction(data3, (data4) => {
console.log("Final data:", data4);
});
});
});
});
```

Drawback: Inversion of Control

The function receiving the callback controls when and how the callback executes.

This can cause problems when dealing with errors or modifying execution behavior.

2. XMLHttpRequest (Why It's Not in Use Much)

Before Fetch API and Promises, XMLHttpRequest (XHR) was used to make network requests. However, it had several downsides:

It required multiple event listeners (onreadystatechange, onload, onerror).

The syntax was cumbersome and harder to read.

It did not support Promises, making chaining difficult.

Example: Making a Request with XMLHttpRequest

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data", true);

xhr.onload = function () {
if (xhr.status === 200) {
console.log("Response:", xhr.responseText);
} else {
console.error("Request failed:", xhr.status);
}
};

xhr.onerror = function () {
console.error("Network error");
};

xhr.send();
```

Why Not Used Much?

Complex syntax

Difficult error handling

No built-in Promise support

3. Promises (Better Way to Handle Async)
A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation. It has three states:

Pending → Initial state (waiting for the operation to complete)

Fulfilled → Operation completed successfully (.then())

Rejected → Operation failed (.catch())

Example: Using Promises

```
function fetchData() {
return new Promise((resolve, reject) => {
setTimeout(() => {
let success = true;
if (success) resolve("Data received");
else reject("Error fetching data");
}, 2000);
});
}

fetchData()
.then((data) => {
console.log(data);
```

```
})
.catch((error) => {
console.error(error);
})
.finally(() => {
console.log("Request completed");
});
```

Promise Methods

Method	Description
`.then(callback)`	Runs when promise resolves successfully
`.catch(callback)`	Runs when promise is rejected
`.finally(callback)`	Runs regardless of resolve/reject

Advanced Promise Methods

Method	Description
`Promise.all([p1, p2])`	Resolves when **all** promises are resolved, fails if **one** fails.
`Promise.race([p1, p2])`	Resolves/rejects as soon as the **first** promise settles.
`Promise.any([p1, p2])`	Resolves as soon as **one** promise resolves (ignores failures).
`Promise.allSettled([p1, p2])`	Resolves when **all** promises settle (success or failure).

Example: Promise.all vs. Promise.race

```
const p1 = new Promise((res) => setTimeout(() => res("P1 done"), 1000));
const p2 = new Promise((res) => setTimeout(() => res("P2 done"), 2000));

Promise.all([p1, p2]).then(console.log); // ["P1 done", "P2 done"]

Promise.race([p1, p2]).then(console.log); // "P1 done" (whichever is faster)
```

4. Async/Await (Modern & Efficient)

async/await is syntactic sugar over Promises, making asynchronous code look synchronous, improving readability and maintainability.

Example: Using Async/Await

```
async function fetchData() {
try {
let response = await new Promise((resolve) =>
setTimeout(() => resolve("Data received"), 2000)
);
console.log(response);
} catch (error) {
console.error("Error:", error);
} finally {
console.log("Done");
}
}
```

```
fetchData();
```

Benefits of Async/Await

Improves Readability: No need for .then() chaining.

Easier Error Handling: Works well with try-catch.

Better Control Flow: Looks more like synchronous code.

5. Error Handling (try-catch & .catch())

Handling errors in asynchronous code is crucial to prevent crashes.

1. Handling Errors in Promises

```
fetchData()
.then((data) => console.log(data))
.catch((error) => console.error("Promise error:", error));
```

2. Handling Errors in Async/Await (Recommended)

```
async function fetchData() {
try {
let response = await someAsyncFunction();
console.log(response);
} catch (error) {
console.error("Async error:", error);
}
}
```

3. Using .finally()

.finally() ensures that a piece of code runs regardless of success or failure.

```
fetchData()
.finally(() => console.log("Execution completed"));
```

6. Should You Use Promises or Async/Await?

Both Promises and Async/Await are effective for handling asynchronous operations in JavaScript. The choice between them depends on readability, control flow, and specific use cases rather than one being inherently better than the other.

When to Use Promises (.then() & .catch())

Useful when handling multiple parallel async operations (Promise.all, Promise.race).

Works well in libraries where callbacks are needed.

Can be used when handling event-based async tasks.

When to Use Async/Await

Best for sequential async operations (one step depends on the previous result).

Makes code look synchronous, improving readability.

Easier to use with try-catch for error handling.