**Asynchronous Programming & Callbacks in JavaScript**

**1. What is Asynchronous Programming?**

**Asynchronous programming** allows your code to run **without blocking** other tasks.  
It lets your program **do multiple things at once** (like waiting for a network request **while** still allowing the user to click buttons).

**Synchronous vs. Asynchronous Execution**

**Synchronous Code:**

* Executes **line-by-line**, one after another.
* **Blocking** – waits until current task finishes.

console.log("Task 1");

console.log("Task 2");

console.log("Task 3");

Output:

Task 1

Task 2

Task 3

**Asynchronous Code:**

* Some lines **wait in the background**, and the rest of the code keeps running.
* **Non-blocking**.

console.log("Task 1");

setTimeout(() => {

console.log("Task 2"); // runs after 2 seconds

}, 2000);

console.log("Task 3");

Output:

Task 1

Task 3

Task 2 ← comes later!

**Use Cases of Asynchronous Code:**

| **Use Case** | **Example** |
| --- | --- |
| Network requests | Fetch data from server (API) |
| Timers | setTimeout, setInterval |
| Event listeners | Clicks, keypress, scroll etc. |

**What is a Callback?**

**Callback = A function passed as an argument to another function.**

Used to run code **after a task is complete** (like waiting for data or time delay).

**Simple Example:**

function greet(name, callback) {

console.log("Hello " + name);

callback(); // calling another function

}

function sayBye() {

console.log("Goodbye!");

}

greet("Ali", sayBye);

Output:

Hello Ali

Goodbye!

**Async Callback Example (with setTimeout)**

function fetchData(callback) {

setTimeout(() => {

console.log("Data received!");

callback();

}, 2000);

}

function showMessage() {

console.log("Now processing the data.");

}

fetchData(showMessage);

Output:

(Data received after 2 seconds)

Data received!

Now processing the data.

**JavaScript Promises and Error Handling**

**1. Real Life Example First**

Imagine this:  
You tell your friend:  
“Please bring me a pizza after 1 hour.”

Now you are waiting. There are 3 possibilities:

1. He brings the pizza → **Success**
2. He doesn’t bring pizza → **Error**
3. You are waiting → **Pending**

This is exactly how a **Promise** works in JavaScript!

**2. What is a Promise?**

A **Promise** is a way to write code for **future tasks** (like getting data from the internet).

It helps us **handle success and errors** in a clean way.

**Promise States**

| **State** | **Meaning** |
| --- | --- |
| Pending | Task is in progress (waiting) |
| Fulfilled | Task completed successfully |
| Rejected | Task failed (error happened) |

**3. How to Create a Promise**

let pizzaPromise = new Promise(function(resolve, reject) {

let pizzaReady = true;

if (pizzaReady) {

resolve("Pizza is here! 🍕");

} else {

reject("No pizza! 😢");

}

});

* resolve() = success
* reject() = failure or error

**How to Use a Promise**

pizzaPromise

.then(function(successMessage) {

console.log("Success:", successMessage);

})

.catch(function(errorMessage) {

console.log("Error:", errorMessage);

})

.finally(function() {

console.log("Finished waiting.");

});

**What will you see on screen?**

If pizza is ready:

Success: Pizza is here! 🍕

Finished waiting.

If pizza is not ready:

Error: No pizza! 😢

Finished waiting.

**Promise Chaining (Step by Step) – Easy Explanation**

**Code:**

new Promise(function(resolve) {

resolve(5);

})

.then(function(number) {

console.log("Step 1:", number); // 5

return number \* 2;

})

.then(function(number) {

console.log("Step 2:", number); // 10

return number + 3;

})

.then(function(number) {

console.log("Step 3:", number); // 13

});

**What is happening?**

**Step-by-step theory:**

1. new Promise(function(resolve) {...})  
   ➤ A new promise is created.  
   ➤ resolve(5) means: “Promise completed, and the result is 5”.
2. .then(function(number) {...})  
   ➤ It receives the value from resolve() → **5**  
   ➤ It prints: Step 1: 5  
   ➤ It returns number \* 2 → which is 5 \* 2 = 10 → goes to next .then
3. Next .then(function(number) {...})  
   ➤ It receives 10 from the previous step  
   ➤ It prints: Step 2: 10  
   ➤ It returns number + 3 → 10 + 3 = 13
4. Last .then(function(number) {...})  
   ➤ It receives 13  
   ➤ It prints: Step 3: 13

**Final Output:**

Step 1: 5

Step 2: 10

Step 3: 13

**What is chaining?**

Each .then() takes the **result** of the previous step, **does something**, and **sends it forward**.

It’s like passing a ball:  
→ One player catches it → modifies it → passes to next → and so on.

**6. Error Handling with .catch() – Easy Explanation**

**Code:**

new Promise(function(resolve, reject) {

let internetWorking = false;

if (internetWorking) {

resolve("Data loaded successfully!");

} else {

reject("Internet not working!");

}

})

.then(function(data) {

console.log("Success:", data);

})

.catch(function(error) {

console.log("Error:", error);

})

.finally(function() {

console.log("Finished process.");

});

**What is happening?**

**Step-by-step theory:**

1. A promise is created using new Promise(...).
2. Variable internetWorking is false, so:  
   ➤ reject("Internet not working!") runs.
3. The promise is now **Rejected**, so:
   * .then() is **skipped**.
   * .catch() runs and prints:  
     Error: Internet not working!
4. .finally() always runs no matter what.  
   ➤ It prints: Finished process.

**Final Output:**

Error: Internet not working!

Finished process.

**JavaScript async/await and API Calls**

**Easy Theory + Real Examples**

**1. What is async and await?**

**Real-life example:**

You order food online. Now you're waiting.

* While food is coming → You can do other things (this is asynchronous)
* When food arrives → You receive it (this is await)

**JavaScript Meaning:**

| **Keyword** | **Meaning** |
| --- | --- |
| async | Makes a function asynchronous (can wait) |
| await | Tells the code to "pause here" until task is done |

**Example:**

async function getData() {

console.log("Start");

await new Promise(resolve => setTimeout(resolve, 2000)); // wait 2 seconds

console.log("Data received");

}

getData();

Output:

Start

(wait 2 seconds)

Data received

await pauses the function until the Promise (timer) is done.

**2. Using async/await to Call an API**

async function fetchUser() {

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

const data = await response.json();

console.log(data);

}

fetchUser();

**What happens:**

1. fetch(...) sends request to the API
2. await waits for the response
3. response.json() converts the result to usable data
4. console.log(data) shows the result

Output (example):

{

"id": 1,

"name": "Leanne Graham",

...

}

**Handling Errors using try...catch**

If something goes wrong (e.g., no internet or wrong link), you can handle it like this:

async function fetchUser() {

try {

const response = await fetch("https://wrong-url.com");

const data = await response.json();

console.log(data);

} catch (error) {

console.log("Error happened:", error.message);

}

}

fetchUser();

Output if API fails:

Error happened: Failed to fetch

try runs the main code  
catch handles any errors (if something goes wrong)

**Multiple Async Functions (Sequential & Together)**

**Running one after another (sequential)**

async function getName() {

return "Zahid";

}

async function getAge() {

return 35;

}

async function showInfo() {

const name = await getName(); // wait for name

const age = await getAge(); // wait for age

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

console.log("Age:", age);

}

showInfo();

Output:

Name: Zahid

Age: 35

**Run them together using Promise.all() (faster)**

async function showInfo() {

const [name, age] = await Promise.all([getName(), getAge()]);

console.log(name, age);

}

Promise.all() starts both tasks at the same time.

**Final Real API Example (with try...catch)**

async function getUserData() {

try {

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

const user = await response.json();

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

} catch (error) {

console.log("Something went wrong:", error.message);

}

}

getUserData();