**Synchronous Meaning:**Synchronous means the code runs in a particular sequence of instructions given in the program. Each instruction waits for the previous instruction to complete its execution.

**Asynchronous Need:**

Due to synchronous programming, sometimes imp instructions get blocked due to some previous instructions, which causes a delay in the UI. Asynchronous code execution allows to execute next instructions immediately and doesn't block the flow.

**console.log("one")**

**setTimeout(()=>{console.log("Two",5000)**

**console.log("three")**

**How Asynchronous JavaScript Works?**

JavaScript is single-threaded, meaning it executes one instruction at a time in the main thread. However, it achieves asynchronous behavior using the Event Loop, Callback Queue.

**Work Flow:**

To handle asynchronous tasks (like API calls, timers, and file I/O), JavaScript uses:

1. Call Stack → Runs synchronous code.
2. Web APIs (Browser/Node.js) → Handles async operations (setTimeout, fetch, event listeners).
3. Callback Queue → Stores completed async tasks.
4. Event Loop → Moves tasks from Callback Queue to Call Stack when it's empty.

console.log("Start");

setTimeout(() => {

console.log("Inside Timeout");

}, 2000);

console.log("End");

**Explanation:**

Step-by-Step Execution

1. "Start" is logged immediately (Call Stack).
2. setTimeout() is sent to Web API (handled in a separate thread, non-blocking).
3. "End" is logged immediately.
4. After 2 seconds, setTimeout() callback moves to Callback Queue.
5. Event Loop moves it to Call Stack only when it's empty.
6. "Inside Timeout" is logged.

**What is a Promise?**

* A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.
* It helps in handling asynchronous code more efficiently, avoiding callback hell.

**States of a Promise**

A Promise can be in one of three states:

1. **Pending** → Initial state, neither fulfilled nor rejected.
2. **Fulfilled** → The operation was completed successfully.
3. **Rejected** → The operation failed.

**Creating a Promise (Real World Example):**

function getUser(userId) {

return new Promise((resolve) => {

setTimeout(() => resolve({ id: userId, name: "John Doe" }), 1000);

});

}

function getOrders(user) {

return new Promise((resolve) => {

setTimeout(() => resolve(["Order1", "Order2"]), 1000);

});

}

function processOrders(orders) {

return new Promise((resolve) => {

setTimeout(() => resolve("Processed Orders"), 1000);

});

}

function sendEmail(processed) {

return new Promise((resolve) => {

setTimeout(() => resolve(`Email sent for ${processed}`), 1000);

});

}}

**//calling it (with chaining)**

getUser(123)

.then(getOrders)

.then(processOrders)

.then(sendEmail)

.then((confirmation) => {

console.log("Order Processed:", confirmation);

})

.catch((error) => {

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

})

.finally(() => console.log("Promise completed"));

**Handling a Promise**

myPromise

.then((result) => console.log(result)) // Runs if resolved

.catch((error) => console.log(error)) // Runs if rejected

.finally(() => console.log("Promise completed")); // Always runs

**Methods**

**Promise.all():** Runs multiple promises in parallel and waits for all to resolve.

Promise.all([

Promise.resolve(1),

new Promise((resolve) => setTimeout(() => resolve(2), 2000)),

Promise.resolve(3),

]).then(console.log); // Output: [1, 2, 3] (after 2 seconds)

**Promise.race():** Returns the result of the first promise that settles (either resolved or rejected).

Promise.race([

new Promise((resolve) => setTimeout(() => resolve("Fast"), 1000)),

new Promise((resolve) => setTimeout(() => resolve("Slow"), 3000)),

]).then(console.log); // Output: "Fast" (after 1 second)

**Promise.allSettled():** Waits for all promises to settle (either fulfilled or rejected).

Promise.allSettled([

new Promise((resolve, reject) => setTimeout(() => resolve("Success"), 1000)),

new Promise((resolve, reject) => setTimeout(() => reject("Error"), 3000)),

]).then(console.log);

**Promise.any():** Returns the first fulfilled promise, ignoring rejections.

Promise.any([

Promise.reject("Error"),

new Promise((resolve) => setTimeout(() => resolve("Success"), 2000)),

]).then(console.log); // Output: "Success" (after 2 seconds)

**RealWorld Example:**

function fetchWeather(city) {

const apiKey = "your\_api\_key"; // Replace with an actual API key

const url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

fetch(url) // fetch() returns a Promise

.then((response) => {

if (!response.ok) {

throw new Error("City not found");

}

return response.json();

})

.then((data) => {

console.log(`Weather in ${city}: ${data.weather[0].description}`);

})

.catch((error) => {

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

});

}

// Calling the function

fetchWeather("New York");

**using axios:**

const axios = require('axios');

function fetchWeather(city) {

const apiKey = "your\_api\_key"; // Replace with an actual API key

const url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

axios.get(url) // Axios returns a Promise

.then((response) => {

console.log(`Weather in ${city}: ${response.data.weather[0].description}`);

})

.catch((error) => {

console.log("Error:", error.response ? error.response.data.message : error.message);

});

}

// Calling the function

fetchWeather("New York");

**Advantages of Axios over Fetch**

✔ Automatically parses JSON (no need for response.json()).  
✔ Handles HTTP errors better (fetch doesn't reject on HTTP errors).  
✔ Supports request cancellation and timeouts.  
✔ Allows automatic transformation of data.

**Better Alternative**

**async & await:** Simplifies working with Promises.

**Syntax:**

async function fetchData() {

try {

let result = await myPromise;

console.log(result);

} catch (error) {

console.log(error);

}

}

fetchData();

async function fetchWeatherAsync(city) {

try {

const apiKey = "your\_api\_key";

const url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

let response = await fetch(url);

if (!response.ok) {

throw new Error("City not found");

}

let data = await response.json();

console.log(`Weather in ${city}: ${data.weather[0].description}`);

} catch (error) {

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

}

}

// Calling the function

fetchWeatherAsync("New York");