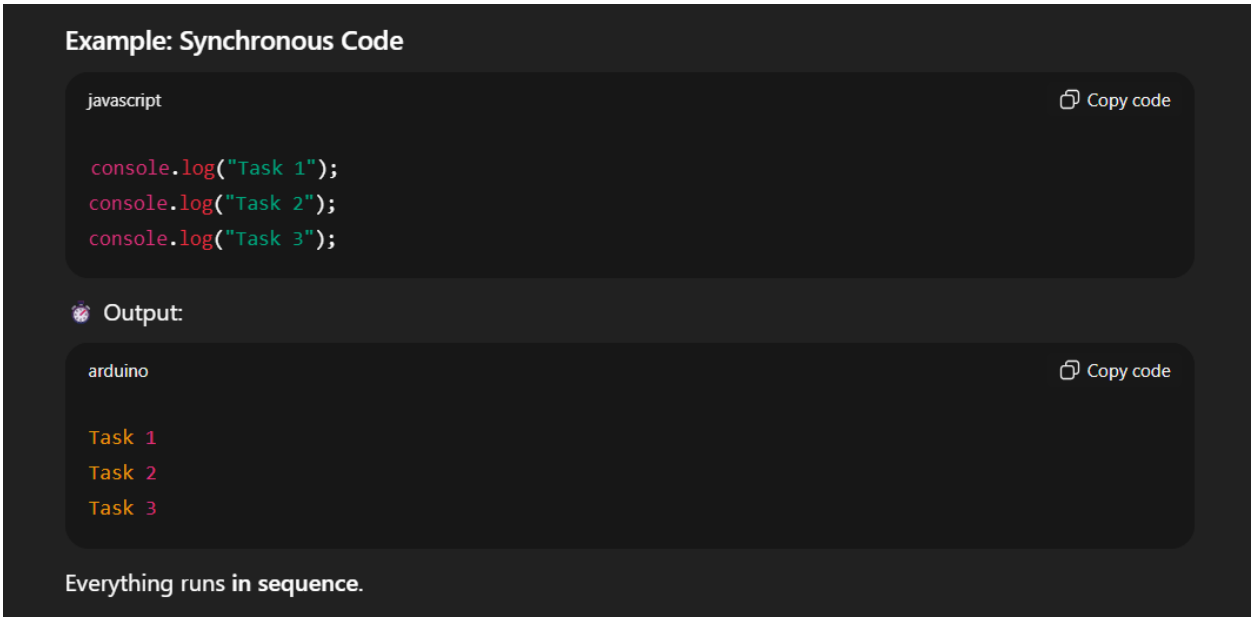
Synchronous vs Asynchronous in JavaScript

1. What is Synchronous?

- **Code runs line by line, in order.**
- **Each statement waits for the previous one to finish.**
- **Blocking behavior → if one task takes time, everything else is paused.**

Example: Synchronous Code



2. What is Asynchronous?

- **Code does not wait for long-running tasks to complete.**
- **Non-blocking behavior → other code can run while waiting.**
- **Useful for tasks like:**
    - **Fetching data from APIs**
    - **Reading/writing files**
    - **Timers**

Example: Asynchronous with `setTimeout`

javascript — Copy code

```javascript
console.log("Task 1");

setTimeout(() => {
  console.log("Task 2 (delayed by 2s)");
}, 2000);

console.log("Task 3");
```

Output:

arduino — Copy code

```arduino
Task 1
Task 3
Task 2 (delayed by 2s)
```

👉 Notice how **Task 3 runs before Task 2**, because `setTimeout` is asynchronous.

3. Why Asynchronous?

Imagine you go to a restaurant:

- **Synchronous: You order food, wait at the counter until it's ready, then eat. (⏳ Time wasted)**
- **Asynchronous: You order food, sit at a table, do other things, and the waiter brings your food when it's ready. (✅ Efficient)**

4. Types of Asynchronous Handling in JS

1. **Callbacks (old way, leads to callback hell)**
2. **Promises (modern solution)**
3. **Async/Await (syntactic sugar over promises, most readable)**

Summary

- **Synchronous = step by step, blocking.**
- **Asynchronous = non-blocking, other code can run while waiting.**
- **JavaScript uses async behavior heavily for I/O, and API calls.**

1. What is a Promise?

A **Promise** is an object in JavaScript that represents the eventual result of an asynchronous operation.
It can be in one of three states:

- **Pending** → Initial state (waiting for result).
- **Fulfilled** → Operation completed successfully.
- **Rejected** → Operation failed.

Example: Creating a Promise

```
1
2  const myPromise = new Promise((resolve, reject) => {
3    let success = true;
4
5    if (success) {
6      resolve("✅ Task completed!");
7    } else {
8      reject("❌ Task failed!");
9    }
10 });
11
12 // Using the promise
13 myPromise
14   .then(result => console.log(result)) // if resolved
15   .catch(error => console.error(error)) // if rejected
16   .finally(() => console.log("Promise finished!"));
17
```

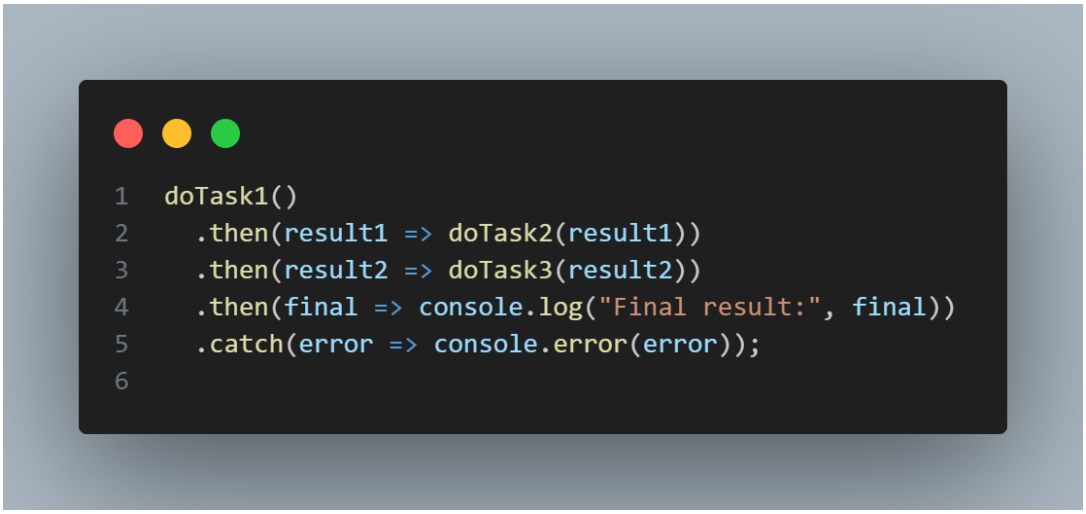
2. Why Promises?

Before Promises, we used **callbacks**, which often caused **callback hell** (nested functions). Promises make async code easier to read and manage.

Callback Hell Example

```
doTask1(function(result1) {
doTask2(result1, function(result2) {
  doTask3(result2, function(result3) {
    console.log("Final result:", result3);
  });
});
});
```

Promise Chaining



```
doTask1()
  .then(result1 => doTask2(result1))
  .then(result2 => doTask3(result2))
  .then(final => console.log("Final result:", final))
  .catch(error => console.error(error));
```

3. Async & Await

async and await were introduced to make working with promises look more like synchronous code.

- **async → makes a function return a promise.**
- **await → pauses execution until the promise resolves.**

Example: Using Async/Await

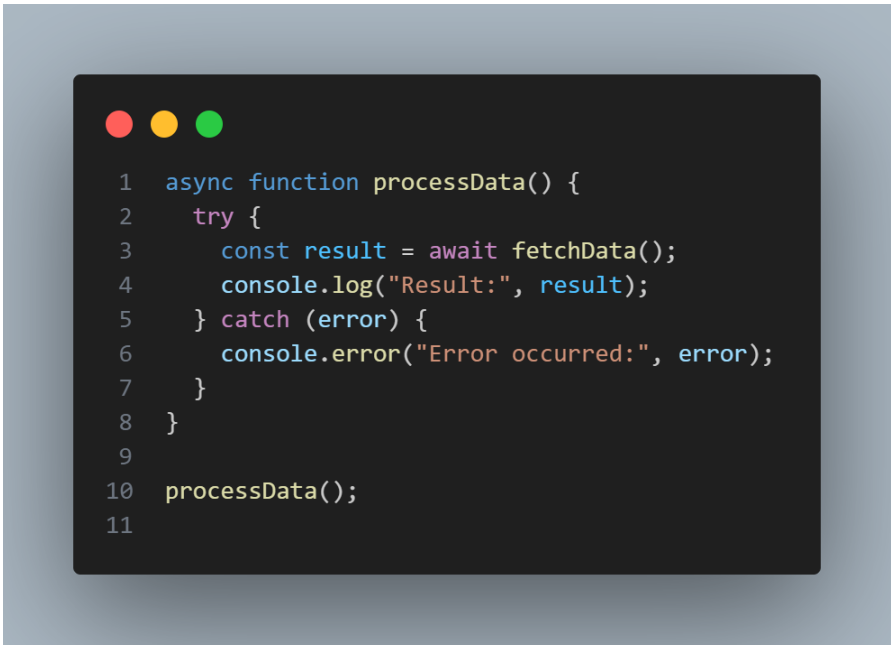
```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("📦 Data received!"), 2000);
  });
}

async function getData() {
  console.log("Fetching data...");
  const result = await fetchData(); // waits for fetchData to resolve
  console.log(result);
  console.log("Done!");
}

getData();
```

Output:

```
Fetching data...
📦 Data received!
Done!
```

4. Handling Errors with Try/Catch

Instead of .catch(), you can use try...catch with async/await.



```
async function processData() {
  try {
    const result = await fetchData();
    console.log("Result:", result);
  } catch (error) {
    console.error("Error occurred:", error);
  }
}

processData();

```

5. Running Promises in Parallel

Sometimes you need to run multiple async tasks at the same time.

Example: Promise.all

```
const p1 = new Promise(resolve => setTimeout(() => resolve("Task 1 done"), 1000));
const p2 = new Promise(resolve => setTimeout(() => resolve("Task 2 done"), 2000));

async function runTasks() {
  const results = await Promise.all([p1, p2]);
  console.log(results); // ["Task 1 done", "Task 2 done"]
}

runTasks();
```

6. Real-World Example (Fetching API Data)

```
async function fetchUser() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users/1");
    const user = await response.json();
    console.log("User:", user);
  } catch (err) {
    console.error("Failed to fetch user:", err);
  }
}

fetchUser();
```

Summary

- **Promises represent future values.**
- **Use .then(), .catch(), .finally() for handling.**
- **async/await makes async code easier to read.**
- **Always use try/catch for error handling.**
- **Use Promise.all for parallel execution.**