What is Synchronous and Asynchronous Programming?

Before understanding asynchronous JavaScript, let's first see how JavaScript normally works.

🔹 Synchronous Programming (Step-by-Step Execution)

- JavaScript is **single-threaded**, meaning it executes one task at a time.
- If a task takes time, everything else waits until it finishes - Synchronous and Blocking..
- If func1() is defined before func2(), func1() no matter even if it has 10,000 Lines of code, func2() will wait until func1() execution gets completed.

📌 Example of Synchronous Code:

```
console.log("Step 1: Start");

console.log("Step 2: Processing...");

console.log("Step 3: End");
```

🔍 Output:

```
Step 1: Start

Step 2: Processing...

Step 3: End
```

Why Do We Need Asynchronous JavaScript?

Imagine if JavaScript were **only** synchronous:

- If one task takes too long (e.g., fetching data from a website), the entire program **stops** until it completes.
- Users would experience **slow websites and freezing pages**.

With **asynchronous JavaScript**, we can:

✔️ Perform multiple tasks **simultaneously**.
✔️ Avoid blocking the main execution thread.
✔️ Improve performance in web applications.

Web APIs (Handling Asynchronous Tasks)

JavaScript doesn't handle asynchronous tasks **on its own**. Instead, it uses the **Web APIs** (provided by browsers). These APIs handle operations like:
✔️ `setTimeout()` (timers)
✔️ `fetch()` (network requests)
✔️ `DOM events` (clicks, inputs)

When an async task is triggered, JavaScript **sends it to the Web API**, allowing other code to continue executing.

Traditional Approach of Asynchronous JavaScript?

JavaScript provides two main timer functions:

✅ **`setTimeout()`** – Runs a function **once after a delay**.

✅ **`setInterval()`** – Repeats a function **at regular time intervals**.

`setTimeout()`: Executing a Function After a Delay

The `setTimeout()` function waits for a specified time **before executing a function**.

```
setTimeout(function, delay);
```

- `function` → The function to execute.
- `delay` → Time in **milliseconds (ms)** before running the function (1 sec = 1000 ms).

Example: Passing Arguments to `setTimeout()`

```
function greet(name)

{

     console.log(`Hello ${name}`)

}

console.log("Hello Alice before greet function");
```

```
setTimeout(greet, 2000, 'Alice');

console.log("Hello Alice after greet function");
```

🔍 **Output:**

```
Hello Alice before greet function

Hello Alice after greet function

Hello Alice
```

✅ The function `greet()` runs **after 2 seconds** with `"Alice"` as an argument.

> **First console log printed → the function is sent to web api where timer starts → next console log executed → Timer expires → function goes to task queue →call stack is empty so function moved to call stack. From task queue and executed**
>
> **(The Event Loop continuously checks:**
> **✅ If the Call Stack is empty**
> **✅ If there are callbacks in the Task Queue**
>
> **If both conditions are met, it moves tasks from the Task Queue to the Call Stack for execution.)**

🔹 Clearing `setTimeout()` with `clearTimeout()`

If you **want to cancel** a `setTimeout()` before it executes, use `clearTimeout()`.

```
console.log("Hello Alice before greet function");

id = setTimeout(greet, 2000, 'Alice');

clearTimeout(id)

console.log("Hello Alice after greet function");
```

🔍 **Output:**

```
Hello Alice before greet function
```

```
    Hello Alice after greet function
```

✅ `Here, clearTimeout(timer) stops the function from running.`

`setInterval()`: Repeating a Function at Intervals

`setInterval()` runs a function **repeatedly at a fixed time interval**.

```
let count = 0;

let interval = setInterval(() => {

    console.log(`Message ${++count}`);

    if (count === 5) {

        clearInterval(interval); // Stops after 5 times

    }

}, 1000);
```

✅ This will **print 5 messages**, then stop.

Callbacks in JavaScript

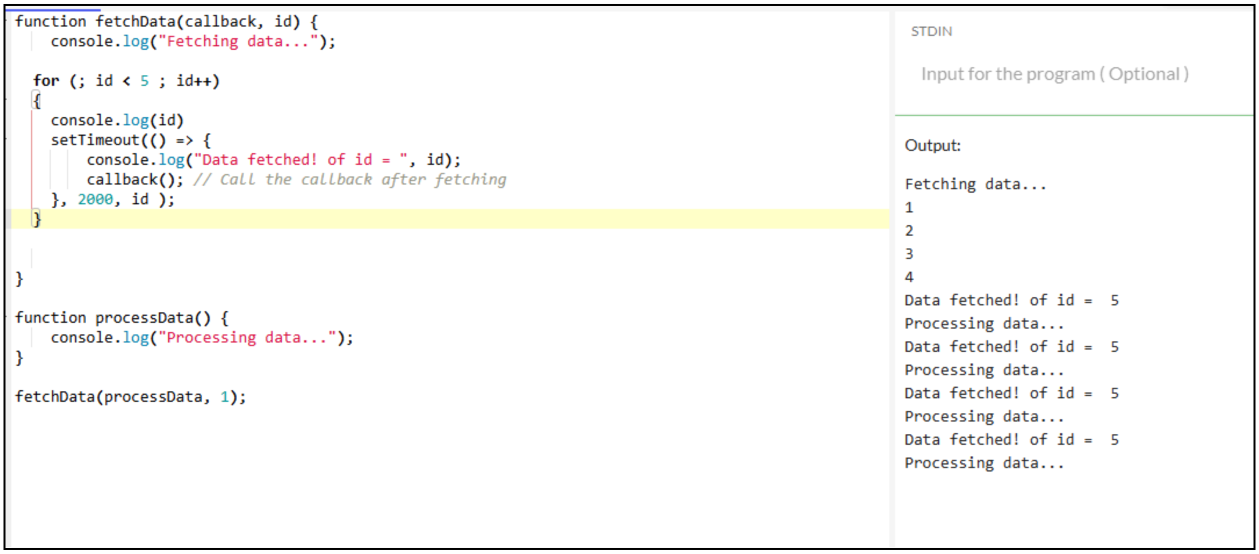
1️⃣ What Are Callbacks?

A **callback** is a function passed as an argument to another function. It is executed later, usually after an asynchronous operation completes.

In JavaScript, callbacks are often used for tasks like:
✔️ Fetching data from an API
✔️ Reading a file from disk
✔️ Waiting for a timer to finish

```
setTimeout(()=>{

  console.log("hello")

}, 2000);



function greet()

{

        console.log("hello 2");

}

setTimeout(greet, 2000)
```

✅ In both of the above cases the call back function serves as a parameter, Either you can given name of the function or define an entire function definition using arrow function.



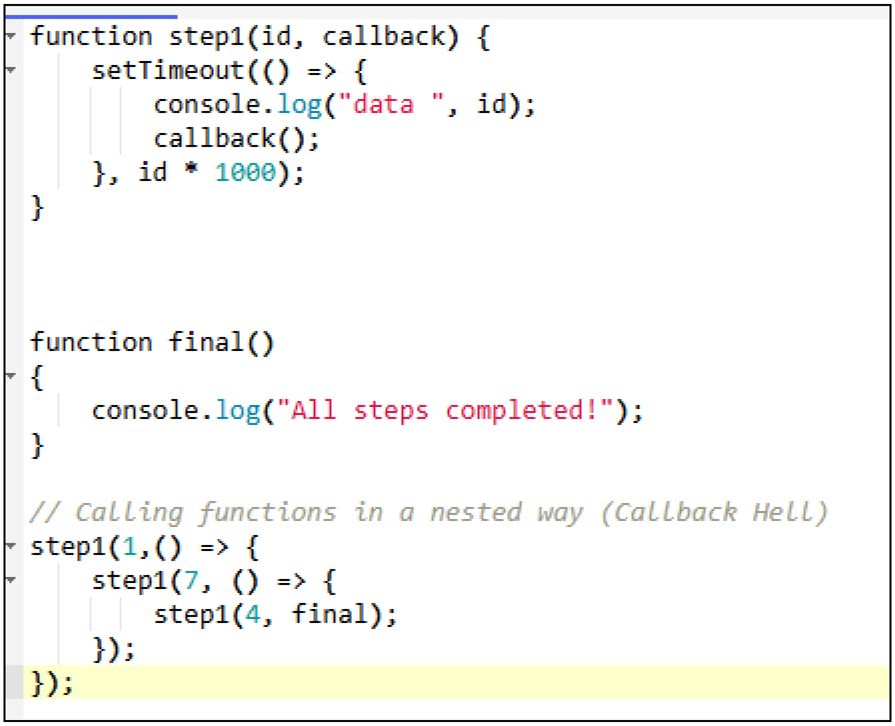
```
function fetchData(callback, id) {
    console.log("Fetching data...");

  for (; id < 5 ; id++)
  {
    console.log(id)
    setTimeout(() => {
        console.log("Data fetched! of id = ", id);
        callback(); // Call the callback after fetching
    }, 2000, id );
  }


}

function processData() {
    console.log("Processing data...");
}

fetchData(processData, 1);
```

Output:

```
Fetching data...
1
2
3
4
Data fetched! of id =  5
Processing data...
Data fetched! of id =  5
Processing data...
Data fetched! of id =  5
Processing data...
Data fetched! of id =  5
Processing data...
```

Callback Hell (Nested Callbacks Problem)

When multiple asynchronous operations depend on each other, we end up **nesting callbacks inside callbacks**. This is called **callback hell**, and it makes the code **hard to read and maintain**.



```
function step1(id, callback) {
    setTimeout(() => {
        console.log("data ", id);
        callback();
    }, id * 1000);
}



function final()
{
    console.log("All steps completed!");
}

// Calling functions in a nested way (Callback Hell)
step1(1,() => {
    step1(7, () => {
        step1(4, final);
    });
});
```

Nested Function Calls (Callback Hell):

- `step1(1, () => {...})` starts execution with `id = 1`, logging `"data 1"` after 1 second.
- Once completed, `step1(7, () => {...})` runs next, logging `"data 7"` after 7 seconds.
- After that, `step1(4, final)` runs, logging `"data 4"` after 4 seconds.
- Finally, `final()` logs `"All steps completed!"`

✅ Each step waits for the previous one to finish.

❌ But the **nested structure** makes the code **hard to read**.

4️⃣ Problems with Callback Hell

💀 Hard to read and debug
💀 Error handling is difficult
💀 Hard to scale (adding more steps makes it even worse)

✔️ **Solution?** Use **Promises** and **async/await** instead of callbacks!