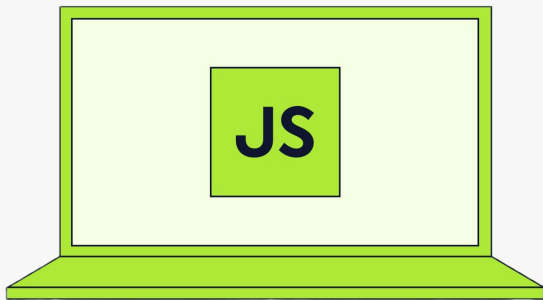


The Complete Javascript Course



Lecture 9: Asynchronous Programming

-Bhavesh Bansal

Table of Contents

- Synchronous Vs Asynchronous Programming
- Asynchronous Programming Fundamentals
- Callbacks: a Deep Dive
- setTimeout and setInterval
- setTimeout working
- Real-world applications
- Error Handling

Synchronous vs Asynchronous

Synchronous Programming

Synchronous programming means doing things one at a time, in the exact order they're written.



Synchronous Programming

So far, we've learned about **functions**, **variable declarations**, **loops**, etc. All these execute one step at a time, in the order written.

```

1 console.log("Start");
2
3 console.log("Synchronous Task 1: Declare variables");
4 let x = 10;
5
6 console.log("Synchronous Task 2: Loop");
7 for (let i = 0; i < 3; i++) {
8   console.log(i);
9 }
10
11 // Output
12 // Start
13 // Synchronous Task 1: Declare variables
14 // Synchronous Task 2: Loop
15 // 0
16 // 1
17 // 2

```

Can you see the output?
Each line executes **only**
after the previous line
finishes.

But do we always do things in order??

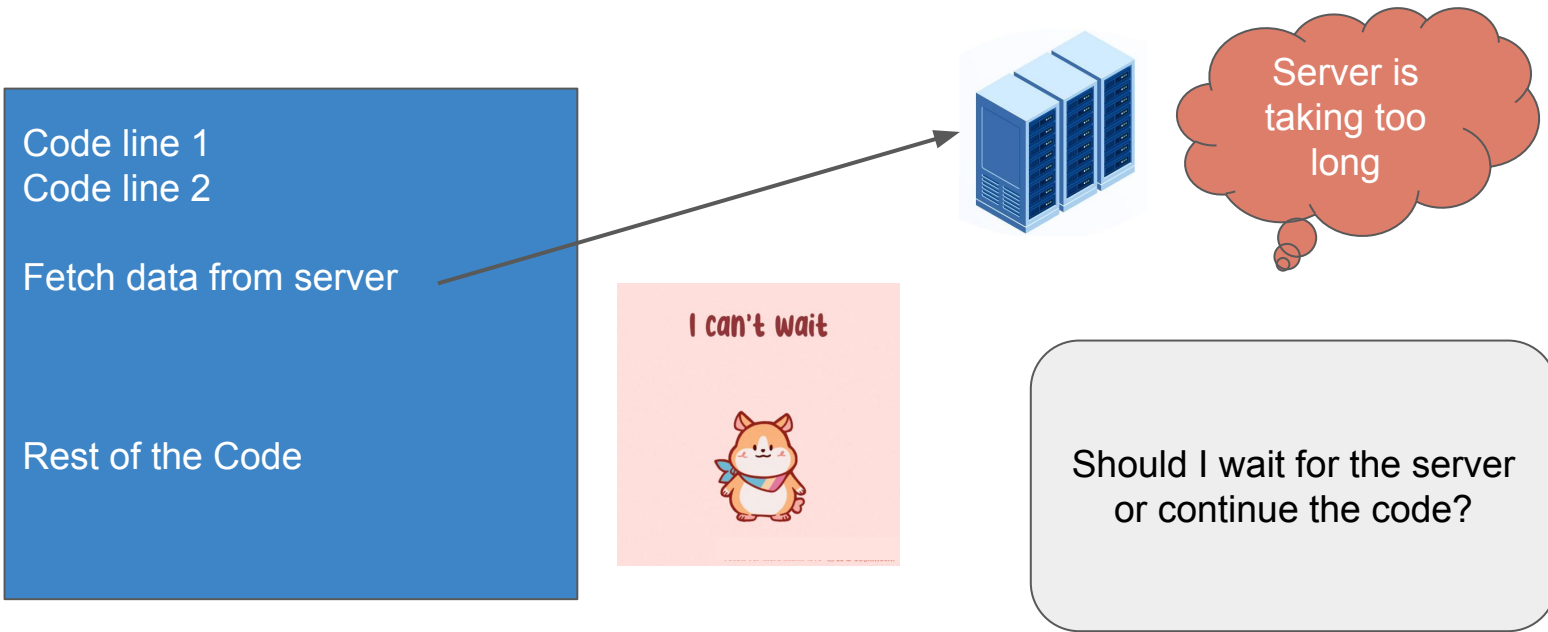
Sometimes we do things one after another, but other times we start something, stop, and switch to something else, i.e., we complete tasks asynchronously.



You might start cooking, but then remember to check a message from work. You stop cooking and check the message first. Did you follow the order?

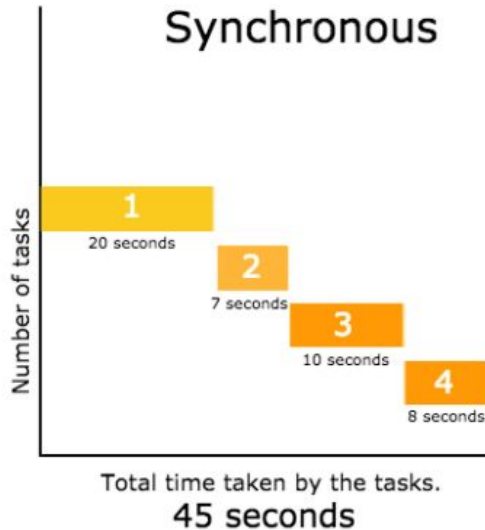
Need for asynchronous programming

Just like in daily life, where tasks aren't always in order, programming uses **asynchronous** tasks to handle unpredictable delays without stopping everything.



Better to Wait and Keep Things Running

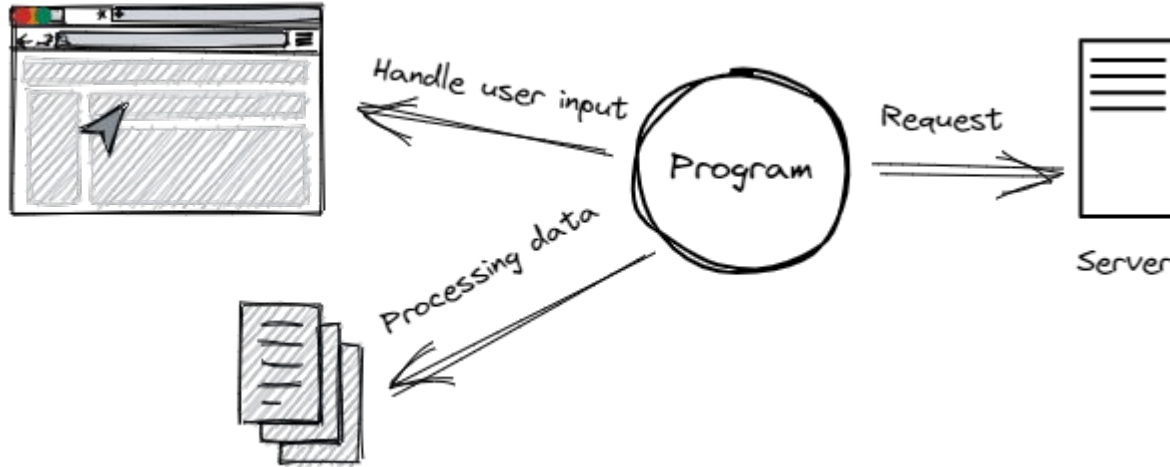
Sometimes it's better to wait for the server response, but instead of pausing everything, we can continue other tasks while waiting.



The idea is to execute shorter tasks first and catch up on longer tasks once they finish, reducing overall execution time.

What is Async Programming?

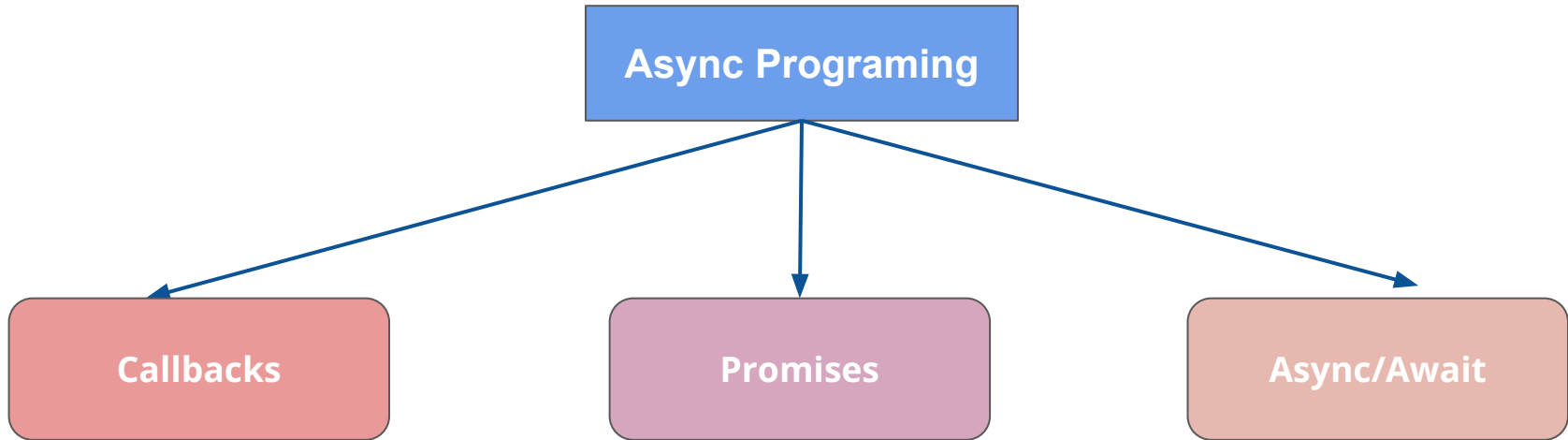
Asynchronous programming lets tasks run independently, allowing your program to continue other tasks while waiting for long operations to finish.



Here, the program sends a request to the server and, while waiting for the response, continues accepting and processing user inputs.

Ways to implement async programming

There are three main ways to implement asynchronous programming in JavaScript:



Callbacks: a Deep Dive

Callbacks: Task to Execute Later

In simple terms, a callback is just a function that you give to another function. The receiving function will then decide when and how to execute.

```
1 function task1(callback) {  
2     console.log("Task 1 completed");  
3     callback();  
4     // Execute the callback after Task 1  
5 }  
6  
7 function task2() {  
8     console.log("Task 2 completed");  
9 }  
10  
11 task1(task2);  
12 // Passing task2 as a callback to task1
```

Here it is up to task1() function when and where it executes callback function. Here it is executing it immediately, but that is not always the case.

Executing callbacks after a while

Let's add a delay of 2000 milliseconds before the callback executes using `setTimeout()`.

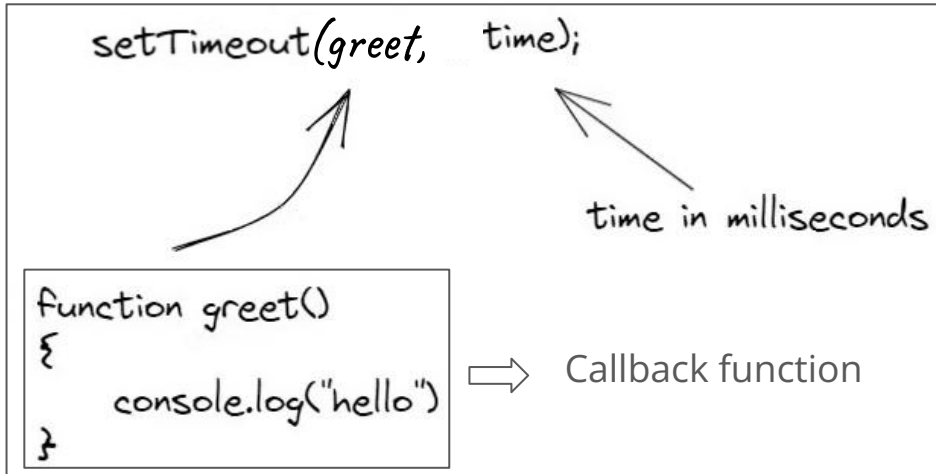
```
1 function task1(callback) {
2     setTimeout(() => {
3         console.log("Task 1 completed");
4         callback();
5         // Execute the callback once Task 1 is done
6     }, 2000); // Simulate a delay
7 }
8
9 function task2() {
10     console.log("Task 2 completed");
11 }
12
13 task1(task2);
14 // Passing task2 as a callback to task1
```

Don't panic, we will
learn how
`setTimeout()` works in
the next slides.

Callbacks in action: setTimeout

`setTimeout` is a JavaScript function that executes a callback function after a specified delay (in milliseconds).

Syntax:



setTimeout: example 1

We can print a specific message after a certain period of time using setTimeout.



```
1  function greet() {  
2      console.log("Hello");  
3  }  
4  
5  // greet function to runs after 2 seconds  
6  // i.e. (2000 milliseconds)  
7  setTimeout(greet, 2000);
```


Here we get output after 2000 milliseconds, i.e. 2 secs

Output:

Hello


setTimeout: example 1

The same example can be rewritten like this using named functions and arrow functions.



```
1  setTimeout(function greet(){
2      console.log("Hello");
3  }, 2000);
```

Using named Function



```
1  setTimeout(()=> {
2      console.log("Hello");
3  }, 2000);
```

Using arrow Function

setTimeout: example 2

Here we have implemented a countdown using setTimeout.

```
1 function startCountdown(seconds) {  
2   function countdown() {  
3     if (seconds > 0) {  
4       console.log(seconds);  
5       // Log the current second  
6       seconds--;  
7       // Decrease the time  
8       setTimeout(countdown, 1000);  
9       // Call countdown again after 1 second  
10    } else {  
11      console.log("Time's up!");  
12    }  
13  }  
14  countdown();  
15 }  
16  
17 // Start a countdown from 5 seconds  
18 startCountdown(5);
```

Output:

5

4

3

2

1

Time's up!

Callbacks in action: setInterval

`setInterval` is a JavaScript function that repeatedly executes a specified callback function at fixed time intervals (in milliseconds).

Syntax:

```
setInterval(greet, time);
```

time in milliseconds

```
function greet()
{
  console.log("hello")
}
```



setInterval: example 1

Here we are printing and updating the message count every 1000 milliseconds. The `setInterval()` function returns a unique identifier, known as an interval ID, which is stored to later clear the interval



```
1 let count = 0;
2
3 const intervalId = setInterval(() => {
4     console.log(`Message ${++count}`);
5 }, 1000);
```

Output:

Message 1

Message 2

Message 3

..

..

setInterval: example 2

Here we are displaying the most updated time using setInterval. The latest time gets updated every 1000 milliseconds, i.e. every second

```
1 function displayTime() {
2     const now = new Date();
3     const time = now.toLocaleTimeString();
4     // Get the current time as a string
5
6     console.log(time);
7     // Display the current time
8 }
9
10 // Update the time every second
11 setInterval(displayTime, 1000);
```

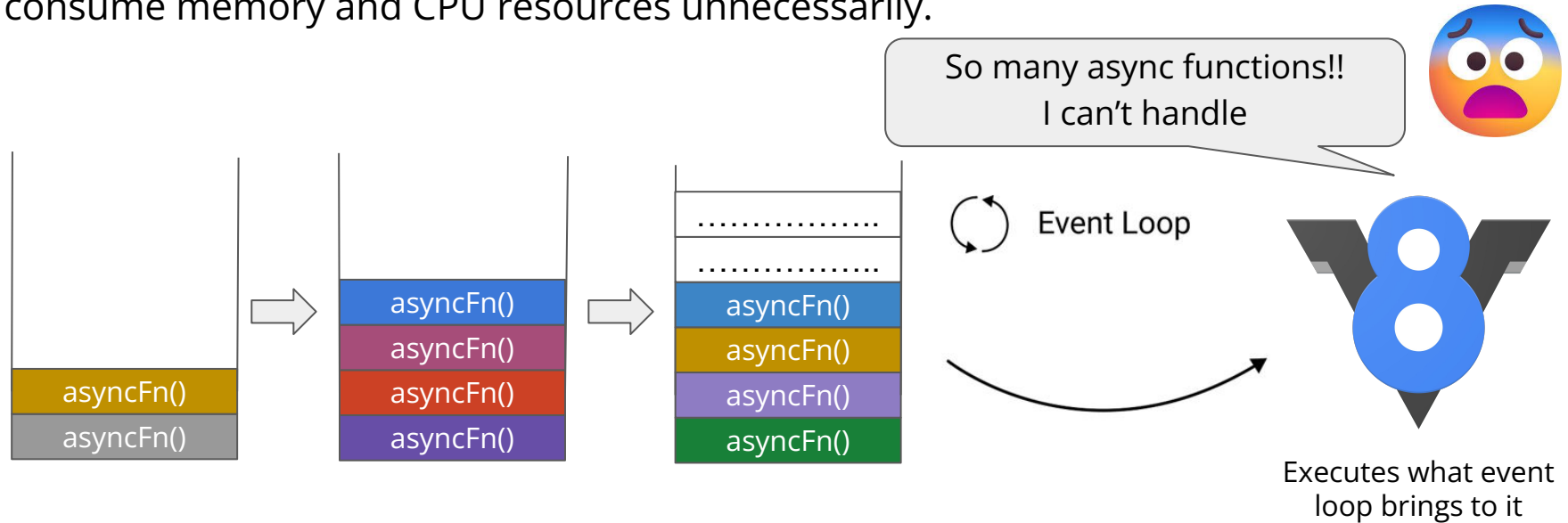
Output:

12:34:56 PM
12:34:57 PM
12:34:58 PM
12:34:59 PM
...

In JavaScript, the **Date** object represents a specific moment in time. Using the **Date()** constructor without arguments creates a new **Date** object set to the current date and time.

Zombie Tasks: Task Queue Congestion

When you have too many async functions running (or scheduled but never cleared), they can be described as 'zombie callbacks' or 'zombie tasks.' These lingering tasks consume memory and CPU resources unnecessarily.



Clearing up Zombie Tasks: setTimeout

So it is always a good practice to free up such asynchronous operations that are taking up system resources

```

1 function startCountdown(seconds) {
2   let countdownTimeout;
3
4   function countdown() {
5     if (seconds > 0) {
6       console.log(seconds); // Log the current second
7       seconds--; // Decrease the time
8       countdownTimeout = setTimeout(countdown, 1000);
9     } else {
10      console.log("Time's up!");
11    }
12  }
13
14  countdown(); // Start the countdown from the given seconds
15
16  // Example: Stop the countdown after 3 seconds
17  setTimeout(() => {
18    clearTimeout(countdownTimeout); // Stop the countdown
19    console.log("Countdown stopped!");
20  }, 3000);
21 }

```

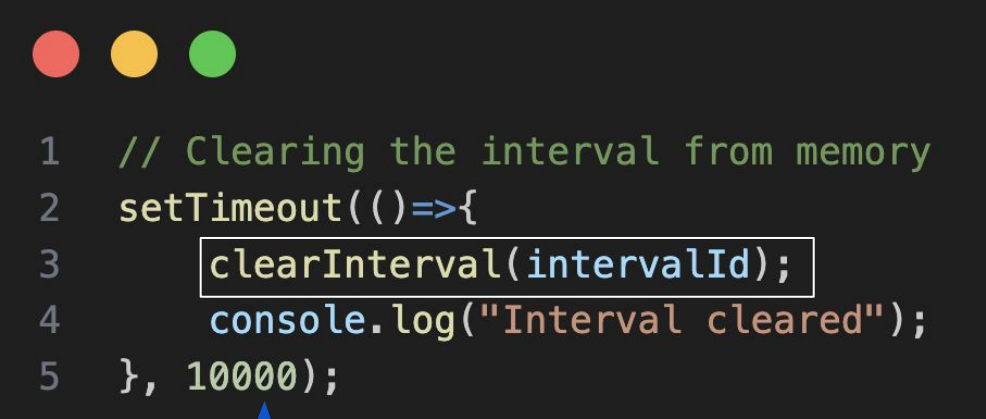


Thank God! Now I feel relaxed!!


It is always a good idea to release setTimeout() when it is no longer used to save up the resources.

Clearing up Congestion: setInterval

You should always conditionally clear the interval after use to avoid needless consumption of resources. Otherwise, it would get repeated infinitely Here is how you can do it:



```
1 // Clearing the interval from memory
2 setTimeout(()=>{
3     clearInterval(intervalId);
4     console.log("Interval cleared");
5 }, 10000);
```



This argument sets clear time to 10000 milliseconds, i.e. 10 secs

Output:

Interval cleared

Some Real World Examples

Quick Reminder System

Imagine you're working on a task and suddenly remember you need to call a friend in 5 minutes. A quick and simple solution can help you set a one-time reminder to pop up after a specific delay.



It's time to call my
friend!!!

Build a timer using setTimeout

A quick solution lets you set a one-time reminder to alert you after a delay.

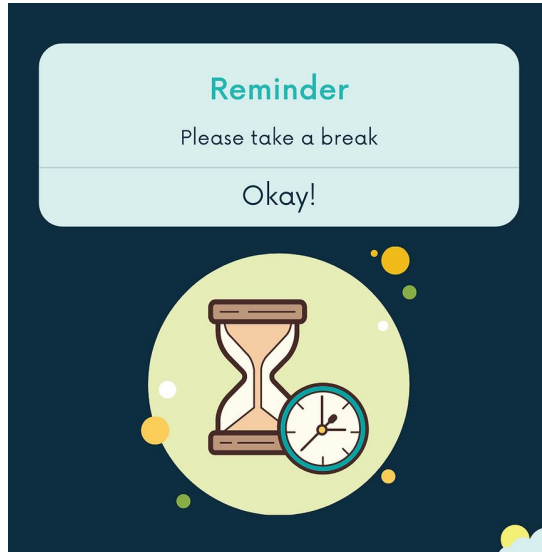
```
1 function setReminder(message, delay) {  
2     console.log(`${message} in ${delay / 1000} seconds.`);  
3     setTimeout(() => console.log(`🕒 Reminder: ${message}`),  
4     delay);  
5 }  
6 // Set a reminder for 5 minutes (300000 ms)  
7 setReminder("Call your friend!", 300000);
```

Output:

Call your friend in 300 seconds

Simple Productivity Timer

You're working on a project and want to stay focused by tracking your time. To stay productive, set a timer to remind you every 30 minutes to take a break.



Build a reminder using setInterval

An easy solution is to build a recurring reminder using setInterval

```

1  function startBreakReminder(interval) {
2      console.log("Timer started! ");
3      setInterval(() => {
4          console.log("🕒 Time for a break!")
5      }, interval);
6  }
7
8  // Reminder every 30 minutes (1800000 ms)
9  startBreakReminder(1800000);
  
```

Output:

Time for a break! ← 30 mins

Time for a break! ← 60 mins

Time for a break! ← 90 mins



setTimeout

Workshop

Workshop: Question 1

You are building a web app that reminds users to save their work every 10 minutes. Write a function using `setTimeout` that logs "Please save your work!" after a 10-minute delay.

What should the function look like?

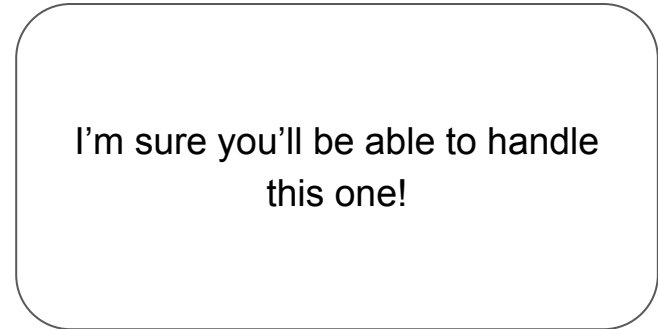
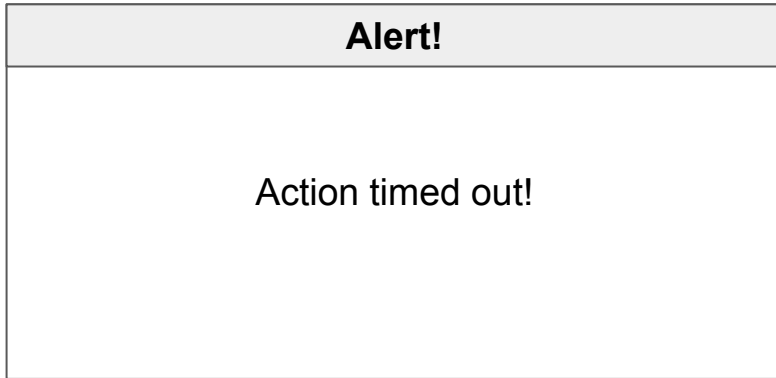
| Reminder |
|--|
| Save your work; otherwise, progress would be lost. |

Can you write a code to implement this reminder??

Workshop: Question 2

In your app, you want to show a message to the user saying "Action timed out!" if they don't click a button within 5 seconds. Use `setTimeout` to implement this timeout mechanism.

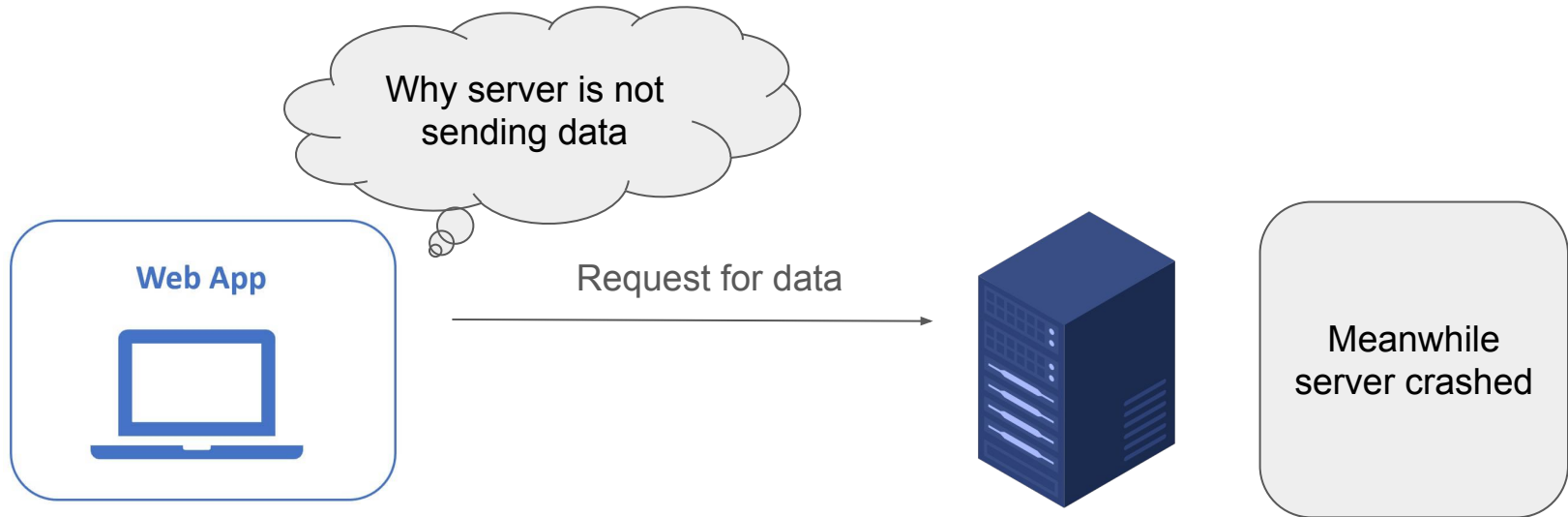
How will you implement this feature using `setTimeout`?



Error Handling

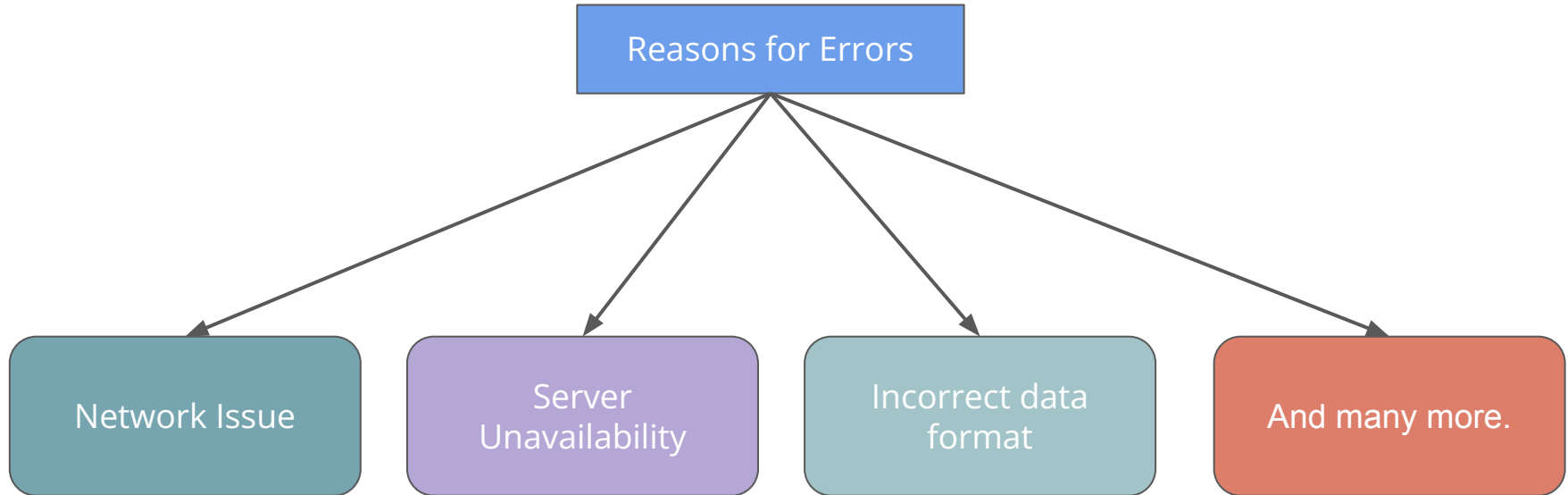
Need for error handling

Imagine building a web app that fetches user data from an API. Network issues, server downtime, or request failures can lead to errors, and if not handled, the app might crash or behave unexpectedly.



Common Causes of Errors

There are several reasons due to which errors can occur:



Handling Errors using try/catch block

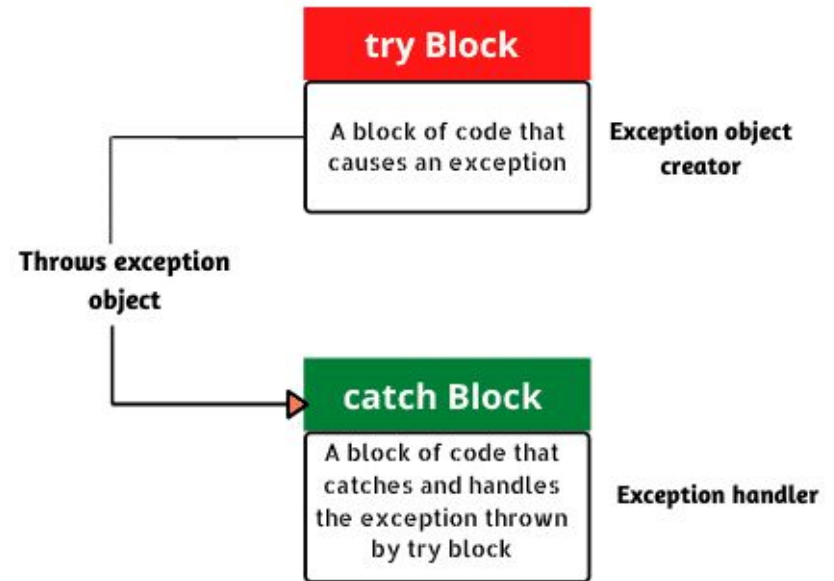
JavaScript provides the `try...catch` statement to handle errors gracefully.

Syntax:

```

1  try {
2
3    // Code that might throw an error
4  } catch (error) {
5    // Code to handle the error
6  }

```



Dummy Example: try/catch

try/catch for fetching data, but for simplicity we have replaced fetch with Math.random()

```
1  async function fetchData() {
2      try {
3          // Simulate an error by using a random number
4          let response = Math.random() > 0.5;
5
6          if (!response) {
7              throw new Error("Failed to fetch data!");
8          }
9
10         console.log("Data fetched successfully!");
11     } catch (error) {
12         console.log("Error:", error.message);
13     }
14 }
```

Output:

// if response === false

Error: Failed to fetch data!

We used *async* keyword to make function asynchronous.

Real Example: try/catch

Let's use proper fetch function to fetch the data from the server

```

1  async function fetchData() {
2    try {
3      let response = await fetch("https://api.example.com/data");
4
5      if (!response.ok) {
6        throw new Error(`HTTP error! Status: ${response.status}`);
7      }
8
9      let data = await response.json();
10     console.log(data);
11
12   } catch (error) {
13     console.error("Error occurred:", error.message);
14     alert("Failed to fetch data. Please try again.");
15   }
16 }

```

Don't stress about fetch() api, we will learn it later

Output:

// if response === false

Error: Failed to fetch data!



In Class Questions

References

1. **MDN Web Docs: JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**