Managing Asynchronous Code Execution

- **Callbacks**: Functions passed as arguments to other functions.
- **Promises**: `fetch(url).then().catch()` .
- **Async/Await**: `async function fetchData() { await fetch(url); }` .
- **setTimeout() & setInterval()**: Execute code after a delay.

Asynchronous Programming

Asynchronous programming is a programming paradigm that facilitates **non-blocking operations**, allowing a program to perform tasks **concurrently** without waiting for each task to complete before moving on to the next one. Examples of asynchronous tasks include API requests, `setTimeout` calls, or file reading.

Key Concepts in Asynchronous JavaScript

1. **Callbacks**
    - A callback is a function passed into another function as an argument, which is then invoked inside the outer function.
    - A callback is specified to run after a task completes.
    - Excessive use of callbacks can lead to **callback hell**.
2. **Promises**
    - Promises are used to handle asynchronous operations.
    - They represent a value that may be available now, in the future, or never.
    - Challenges of callbacks are solved by **promises**.
3. **Async/Await**
    - `async/await` is syntactic sugar built on top of **Promises**.
    - It makes asynchronous code easier to write and read.

Example: Asynchronous Execution in JavaScript

```
console.log("Start");
setTimeout(function cb() {
    console.log("Callback function executed");
}, 500);
console.log("End");
```

Browser Web APIs

The browser provides built-in Web APIs that handle asynchronous operations, including:

1. `setTimeout`
2. DOM API
3. `fetch()`
4. `console`
5. `localStorage`
6. `location`

All browser APIs are present in the `window` object, which means we can call them using `window.setTimeout` or simply `setTimeout`.

Execution Flow: Call Stack, Event Loop, and Queues

Call Stack Execution Flow

Call Stack	JavaScript Code Execution	Web APIs Called
	`console.log("Start");`	
	`setTimeout(callbackTimer, 500);`	`callbackTimer` registered
	`fetch("api url").then(callbackFetch);`	`callbackFetch` registered
	`console.log("End");`	

Event Loop & Queues

Microtask Queue (Higher Priority)

```
_____________________
| callbackFetch       |
_____________________
```

Callback Queue

```
____________________
| callbackTimer     |
____________________
```

How it Works:

1. **Callbacks are registered** with Web APIs.
2. Once a timer expires or data is fetched, the callback moves to its respective queue.
3. **Microtask queue (Promises) has higher priority** over the callback queue (Timers, Events).
4. The **Event Loop** checks the call stack:
    - If the call stack is empty, it moves the first callback from the **Microtask queue** to the call stack.
    - Once microtasks are complete, it moves the first callback from the **Callback queue** to the call stack.

Video Reference: JavaScript Event Loop