## Understanding JavaScript's Execution Model

# Event Loop

## What is the Event Loop?

The **event loop** is a mechanism in JavaScript that handles asynchronous operations and ensures smooth execution of tasks by managing the **call stack**, **Web APIs**, and **callback queue.** It allows JavaScript to remain **single-threaded** while efficiently handling non-blocking tasks like network requests, timers, and user interactions.

## How the Event Loop Works

1. **Call Stack Execution**
    - JavaScript is **single-threaded**, meaning it executes one operation at a time using the **call stack**.
    - The call stack follows a **Last In, First Out (LIFO)** order.
    - Synchronous code runs directly in the call stack.
2. **Web APIs & Asynchronous Tasks**
    - When an asynchronous function (e.g., `setTimeout`, `fetch`, event listeners) is encountered, JavaScript offloads it to the **Web APIs** (provided by the browser).
    - These tasks run in the background without blocking the main thread.
3. **Callback Queue & Microtask Queue**
    - Once an async task completes, its **callback function** moves to the **callback queue**.
    - **Microtasks (Promises, MutationObservers)** are added to the **microtask queue**, which has higher priority than the callback queue.
4. **Event Loop Execution**
    - The event loop continuously checks whether the **call stack is empty**.
    - If empty, it first processes all **microtasks** before handling callbacks from the **callback queue**.
    - This cycle repeats indefinitely, ensuring smooth execution of both synchronous and asynchronous tasks.

## Example of Event Loop in Action

```
console.log("Start");

setTimeout(() => {
    console.log("Timeout callback");
}, 0);

Promise.resolve().then(() => {
    console.log("Promise resolved");
});

console.log("End");
```

### Execution Flow

1. "Start" is logged (synchronous).
2. `setTimeout` is sent to **Web APIs** with a delay of `0ms` (async).
3. `Promise.resolve().then()` is added to the **microtask queue**.
4. "End" is logged (synchronous).
5. Since the **call stack is empty**, the event loop first processes **microtasks** ("Promise resolved").
6. Finally, the **callback queue** executes ("Timeout callback").

### Output

```
Start
End
Promise resolved
Timeout callback
```

## Key Takeaways

1. **Call Stack** runs synchronous code first.
2. **Web APIs** handle asynchronous tasks like `setTimeout` , `fetch` , etc.
3. **Microtasks (Promises) execute before callback queue**.
4. **Event Loop** ensures smooth execution by coordinating these tasks.