Assignment: Async Javascript

1. Demonstrate JavaScript's Single-Threaded Nature

Question:

Write an example to show that JavaScript is **single-threaded** by creating two competing tasks, one that blocks the event loop and another async function that waits for a promise.

Ans) **JavaScript code**: link

2. Why Does JavaScript Not Execute Asynchronously by Default?

Question:

JavaScript is often called **synchronous** and **single-threaded**, yet it handles asynchronous tasks like AJAX requests, timers, and event listeners.

a. Explain why JavaScript does not execute asynchronously by default.

b. Write a code snippet to prove that JavaScript is inherently synchronous.

Ans)
JavaScript is single threaded as it has only **one call stack** i.e. it can run only **one piece of code** at a time.

1. The code runs from t**op to bottom** in order it is written(Synchronous).
2. Long running code block the call stack until it is completed. The code below will not be running until the current block is completed.
3. Async behaviors comes from the browser's **Web APIs** or **Node.js APIs** not from JavaScript engine itself.
4. These APIs run tasks in the background, and when they're ready, they use the event loop to push callbacks into the **microtask** or **callback queue.**

JavaScript itself is synchronous; async behavior is achieved through cooperation with external APIs + event loop scheduling.

Example: **JavaScript code**: link

3. Chaining Promises with setTimeout

Modify the delay function to **chain multiple** promises so that three messages are logged **in sequence with delays.**

Ans) **JavaScript code**: link

4. What are the different states of a Promise, and how do they transition?

Ans)

State	Meaning	Can change to
Pending	Initial state, the promise hasn't settled yet.	Fulfilled or Rejected
Fulfilled	The operation completed successfully (has a resolved value).	Immutable
Rejected	The operation failed (has a reason/error).	Immutable

1. **Creation** → Promise starts in pending.
2. **Resolve** → moves from pending → fulfilled (with a value).
3. **Reject** → moves from pending → rejected (with a reason/error).
4. **Once** in fulfilled or rejected, the state is immutable — it never changes again.

5. How does the JavaScript event loop handle Promises differently from setTimeout?

Ans)
The JavaScript event loop treats **Promises** and **setTimeout** differently because they go into different queues.

Flow

1. Synchronous code in the call stack runs first.
2. Once the call stack is empty, the event loop checks for any remaining synchronous tasks. If none exist, it moves on to asynchronous tasks.
3. setTimeout is a runtime feature provided by the browser or **Node.js**. It runs outside the JavaScript engine, and after the specified delay, its callback is placed in the **macrotask** queue (also called the callback queue).
4. The event loop takes the callback from the **macrotask** queue, pushes it onto the call stack, and executes it.
5. Promises are created synchronously, but their resolution (resolve/reject) happens asynchronously.
6. The .then() / .catch() / .finally() handlers are queued in the **microtask** queue.
7. ***Order of execution***: After each **macrotask**, the event loop empties the **microtask** queue before moving to the next **macrotask**. Therefore, if both a promise and a setTimeout callback are ready, the promise's handlers run first.