Asynchronous Programming in JavaScript

==========

Javascript 😀 :

==========

Javascript is a Single Threaded Synchronous Programming Language.

Single Threaded

==========

In JavaScript, "single-threaded" means that it executes one Statement or code or instruction at a time.

meaning it executes one statement before moving to the next.

```
console.log("Step 1");
console.log("Step 2");
console.log("Step 3");
```

```
Output:

Step 1
Step 2
Step 3
```

Synchronous

==========

Synchronous means JavaScript executes code line by line, in order, waiting for each operation to complete before moving to the next one.

Programming Language

==========

A programming language is a set of instructions that a computer can understand and execute. It allows developers to write code that tells a computer what to do.

Think of it as a language for computers, just like we use English or Hindi to communicate with people. Computers understand only binary (0s and 1s), so

programming languages make it easier for humans to give instructions without writing in binary.

Drawbacks of Synchronous Programming 😣

Synchronous programming executes code line by line, waiting for each task to finish before moving to the next. This can cause several issues, especially when dealing with time-consuming operations.

1. Blocking Execution (Slow Performance) 🚫⏳

- If one task takes too long, everything stops and waits until it finishes.
- Example: A long-running loop or a slow API request can freeze the entire program.

2. Not Suitable for Large-Scale Applications 🏗️

- Web applications that need to handle multiple users simultaneously (like social media or e-commerce

websites) cannot afford to wait for one task to complete before handling another.

3. **Poor User Experience (Freezing UI)** 🖥️❄️

   - In frontend applications (like web pages), if a task takes too long (e.g., fetching data from a server), the UI becomes unresponsive until the task completes.

4. **Inefficient Resource Utilization** ⚡

   - The CPU sits idle while waiting for tasks (like network requests, file reading) to finish, instead of doing other useful work.

Asynchronous Programming in Javascript 😀

with asynchronous programming, JavaScript can handle multiple operations without blocking the execution of other tasks.

In asynchronous programming, JavaScript does not block other operations while waiting for a task

```
console.log("Task 1");

setTimeout(() => {
    console.log("Task 2 (delayed)");
}, 2000);

console.log("Task 3");
```

```
Output :
Task 1
Task 3
Task 2 (delayed)
```

How to Achieve Asynchronous Programming in JavaScript ? 🤔

1. Using Timer functions like setTimeout & setInterval.
2. Callbacks
3. Promises
4. Async await

Callbacks 🙂

========

A callback is a function that is passed as an argument to another function and is executed later, usually after some operation is completed. Callbacks are commonly used in asynchronous programming to handle tasks that take time, such as API requests, file operations, or database queries.

```
function mainFunction(callback) {
    // Some operation
    console.log("Main function executed");

    // Execute the callback function
    callback();
}

// Define the callback function
function myCallback() {
    console.log("Callback function executed");
}

// Call mainFunction and pass myCallback as an argument
mainFunction(myCallback);
```

1. Define a function (mainFunction) that accepts another function (callback) as a parameter.
2. Call the callback function inside mainFunction when needed.
3. Pass a function (myCallback) as an argument when calling mainFunction.

Promises 😀:

Promise is an object in Javascript which stores success or error data.

Object is a data structure in Javascript

Data Structures are the pattern or mechanism or technique to store and manage the data.

How to create a Promise and Consume it ?

🤔

```
let myPromise = new Promise((resolve, reject) => {
    // Asynchronous operation
    let success = true; // Simulating success or failure

    if (success) {
        resolve("success data");
    } else {
        reject("error data!");
    }
});

// Consuming the Promise
myPromise
    .then(result => console.log(result))  // Runs if resolved
    .catch(error => console.log(error)); // Runs if rejected
```

Syntax Breakdown

1. new Promise((resolve, reject) => { ... })
    - Creates a new Promise object.
    - The executor function takes resolve (on success) and reject (on failure).
2. Inside the executor function:
    - If successful, call resolve(value).

- If failed, call reject(error).

3. Handling the Promise:
   - Use .then() for success.
   - Use .catch() for errors.