Perfect. Continuing **systematically**, **NotebookLM-ready**, with **NOTES and CODE strictly separated**.

📘 TOPIC: Asynchronous JavaScript

(Callbacks → Promises → `async/await` ● Internals ● Interview Traps)

🟦 NOTE 1: Why JavaScript needs async behavior

- JavaScript is **single-threaded**
- Only **one call stack**
- Blocking operations would freeze the app
- Async JS allows **non-blocking behavior**

👉 JS doesn't run tasks in parallel

👉 It **delegates work to the runtime** (browser / Node)

🟦 NOTE 2: Sync vs Async (mental model)

Sync	Async
Blocks execution	Non-blocking
One task at a time	Tasks overlap
Freezes UI	Keeps UI responsive

🟩 CODE 1: Synchronous example

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

for (let i = 0; i < 1e9; i++) {}

console.log("End"); // UI freezes until loop finishes
```

🟦 NOTE 3: Asynchronous delegation

- JS engine executes JS only
- Async work is handled by:
  - Browser Web APIs
  - Node APIs (libuv)
- Completion is notified later

🟩 CODE 2: Basic async example

```js
```

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

setTimeout(() => {
  console.log("Async task");
}, 1000);

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

🟦 NOTE 4: Callbacks (first async pattern)

- A callback is a function passed to be executed later
- Used in timers, events, old APIs
- Simple but becomes unmanageable

🟩 CODE 3: Callback example

```
js

function fetchData(cb) {
  setTimeout(() => {
    cb("Data loaded");
  }, 1000);
}

fetchData(data => {
  console.log(data);
});
```

🟦 NOTE 5: Callback Hell (problem)

- Nested callbacks
- Hard to read
- Hard to debug
- Hard to handle errors

🟩 CODE 4: Callback hell

```
js

setTimeout(() => {
  console.log("Step 1");
  setTimeout(() => {
    console.log("Step 2");
    setTimeout(() => {
      console.log("Step 3");
    }, 1000);
  }, 1000);
}, 1000);
```

🟦 NOTE 6: Promises (solution to callback hell)

- Promise represents a **future value**
- Has 3 states:
  - `pending`
  - `fulfilled`
  - `rejected`
- State is **immutable once settled**

🟩 CODE 5: Creating a Promise

```
js


const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Success");
  }, 1000);
});
```

🟦 NOTE 7: Consuming promises

- `.then()` → success
- `.catch()` → error
- `.finally()` → always runs

🟩 CODE 6: Promise consumption

```
js


promise
  .then(result => console.log(result))
  .catch(err => console.error(err))
  .finally(() => console.log("Done"));
```

🟦 NOTE 8: Promise chaining

- `.then()` returns a new promise
- Enables linear async flow
- Errors propagate automatically

🟩 CODE 7: Promise chaining

```
js
```

```
Promise.resolve(1)
 .then(x => x + 1)
 .then(x => x * 2)
 .then(console.log); // 4
```

🟦 NOTE 9: Promise error propagation (IMPORTANT)

- Error in any `.then()` skips to `.catch()`
- Single `.catch()` can handle multiple errors

🟩 CODE 8: Error propagation

```
js

Promise.resolve()
 .then(() => {
   throw new Error("Oops");
 })
 .then(() => console.log("Not run"))
 .catch(err => console.log(err.message));
```

🟦 NOTE 10: `async` / `await` (syntactic sugar)

- Built on promises
- Makes async code look synchronous
- `async` function **always returns a Promise**

🟩 CODE 9: async function

```
js

async function greet() {
 re
```