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

📘 TOPIC: Event Loop & Concurrency Model

(Call Stack • Web APIs • Callback Queue • Microtasks • Macrotasks)

🟦 NOTE 1: Why the Event Loop exists

- JavaScript is **single-threaded**
- Only **one call stack**
- Async operations must not block execution
- Event Loop coordinates **when async callbacks run**

👉 Event Loop is part of the **runtime**, not the JS language.

🟦 NOTE 2: Core components (mental map)

JavaScript concurrency involves **5 major parts**:

1. Call Stack
2. Web APIs (Browser) / Libuv (Node)
3. Callback (Task) Queue
4. Microtask Queue
5. Event Loop (scheduler)

🟩 CODE 1: Simple async example

```js

console.log("Start");

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

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

Output

```powershell

Start
End
Timeout
```

🟦 NOTE 3: Call Stack (execution engine)

- Stack follows **LIFO** (Last In, First Out)

- JS executes only what's on the stack
- Stack must be **empty** before async callbacks run

🟩 CODE 2: Call stack example

```js
function a() {
  b();
}

function b() {
  console.log("Inside b");
}

a();
```

🟦 NOTE 4: Web APIs / Host APIs

- Provided by browser / Node
- Handle async work:
  - Timers
  - DOM events
  - Fetch / HTTP
- JS engine **delegates** async work here

🟩 CODE 3: Delegation to Web API

```js
setTimeout(() => {
  console.log("Done");
}, 1000);
```

🟦 NOTE 5: Callback (Macrotask) Queue

- Stores callbacks after async task completes
- Examples:
  - `setTimeout`
  - `setInterval`
  - DOM events
- Executed **only when call stack is empty**

🟦 NOTE 6: Event Loop (scheduler logic)

Event Loop continuously checks:

1. Is Call Stack empty?
2. Are Microtasks pending?
3. Execute all Microtasks
4. Take ONE task from Callback Queue
5. Push it to Call Stack
6. Repeat forever

🟩 CODE 4: Visual order example

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

setTimeout(() => console.log("B"), 0);

Promise.resolve().then(() => console.log("C"));

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

Output

```css
A
D
C
B
```

🟦 NOTE 7: Microtask Queue (HIGH PRIORITY)

- Executed **before macrotasks**
- Examples:
  - `Promise.then`
  - `catch`
  - `finally`
  - `queueMicrotask`
- Can starve macrotasks if abused

🟩 CODE 5: Microtask dominance

```js
setTimeout(() => console.log("Timeout"), 0);
```

```
Promise.resolve().then(() => console.log("Promise"));
```

Output

```
javascript

Promise
Timeout
```

🟦 NOTE 8: Macrotasks (lower priority)

Examples:

- `setTimeout`
- `setInterval`
- UI events
- MessageChannel

🟩 CODE 6: Macrotask queue

```
js

setTimeout(() => console.log("Task 1"), 0);
setTimeout(() => console.log("Task 2"), 0);
```

🟦 NOTE 9: Infinite microtask trap (INTERVIEW)

- Microtasks run until queue is empty
- Recursive microtasks block rendering & timers

🟩 CODE 7: Microtask starvation

```
js

function loop() {
  Promise.resolve().then(loop);
}

loop();
// setTimeout callbacks never run
```

🟦 NOTE 10: `queueMicrotask`

- Explicitly schedules microtask
- Same priority as Promise callbacks

🟩 CODE 8: queueMicrotask example

```
js
```

```
queueMicrotask(() => console.log("Microtask"));
console.log("Sync");
```

Output

```
rust

Sync
Microtask
```

🟦 NOTE 11: Rendering & Event Loop (Browser)

- Browser renders **between macrotasks**
- Long JS blocks rendering
- Microtasks delay painting

🟩 CODE 9: Rendering block example

```
js

Promise.resolve().then(() => {
  while (true) {}
});
```

🟦 NOTE 12: Node.js Event Loop (difference hint)

- Uses **libuv**
- Phases:
  - timers
  - I/O callbacks
  - idle
  - poll
  - check
  - close callbacks
- Microtasks still have priority

🟦 NOTE 13: Common interview traps

- Promises run before `setTimeout`
- Event Loop runs only when stack is empty
- Microtasks can block macrotasks
- `setTimeout(fn, 0)` ≠ immediate execution
- Event Loop ≠ Call Stack

🟦 FINAL MENTAL MODEL

- JS runs on one thread
- Async work is delegated
- Call Stack executes JS
- Microtasks run first
- Macrotasks run later
- Event Loop schedules everything

🛑 **STOP HERE**

Next **recommended continuation**:

1️⃣ **Browser vs Node.js Runtime (deep differences)**
2️⃣ **Fetch API + async flow**
3️⃣ **Error Handling (sync vs async)**

Reply with **next** or the **number** you want to continue.