🎬 Episode 17: Trust Issues with `setTimeout()` ⏳

> *Why JavaScript sometimes delays your `setTimeout()` — even with perfect timing!*

🔥 Let's Observe the Code First:

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

setTimeout(function cb() {
  console.log("Callback");
}, 5000); // ⏱️ 5s timer

console.log("End");

// ...Imagine millions of lines of heavy code here!
```

🧠 Expected Output?

```
Start
End
Callback
```

But... ❗ `Callback` might **not** come exactly after 5s.
It could be **6s, 7s**, or even **10s** ⏳. Why?

💡 Deep Dive: Understanding `setTimeout()` Internals

JavaScript is **single-threaded** → it runs code **one task at a time** via a **Call Stack**.

🛠️ Step-by-Step Breakdown:

1. **GEC (Global Execution Context)** is created and pushed onto the **Call Stack** 📦.
2. `console.log("Start")` is printed ☑️.
3. `setTimeout()` is encountered:
   - JS registers the `cb()` function in the **Web API Environment** 🌐.
   - A timer of 5 seconds starts.
   - Meanwhile, JS doesn't wait. It **moves on** 🏃‍♂️.
4. `console.log("End")` is printed ☑️.
5. Then comes our **heavy 1 million lines of code** taking **10 seconds** to execute 😵.
6. Meanwhile, the **5s timer expires** in the background.
7. `cb()` is now pushed to the **Callback Queue** 🕐.
8. But the **Call Stack is still busy** with the heavy code, so:
   - Event Loop 🚦 keeps checking...

- But can't push `cb()` until the stack is free 😨.

9. After **10 seconds**, the Call Stack is finally empty.
10. Now, Event Loop **pushes `cb()` to Call Stack**, and it gets **executed instantly** ⚡.

🧩 Key Concept: JavaScript's Concurrency Model

JS uses:

- 🌐 **Web APIs** → For timers, DOM, fetch, etc.
- 🎟️ **Callback Queue** → Where completed async callbacks wait.
- 🔁 **Event Loop** → Continuously checks if the Call Stack is empty and pushes queued tasks.

🧪 Example 2: Blocking the Main Thread ❌

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

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

console.log("End");

// Simulating blocking code (10s delay)
let startDate = new Date().getTime();
let endDate = startDate;

while (endDate < startDate + 10000) {
  endDate = new Date().getTime();
}

console.log("While expires");
```

📟 Output:

```
Start
End
While expires
Callback
```

🔍 Why?

- `cb()` had to wait in the **Callback Queue** until `while()` finished hogging the stack for **10s**.
- So even though timer was **5s**, the actual execution was **delayed**.

⏱️ What if Timeout = 0?

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

setTimeout(function cb() {
  console.log("Callback");
}, 0);

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

📃 Output:

```
Start
End
Callback
```

- Even with `0ms`, the `cb()` still goes through:
  - Web API 🌐 → Callback Queue 🎟️ → Call Stack 📦 (via Event Loop 🔁).
- So it always runs **after** the synchronous code finishes!

🔗 Core Takeaways

⚙️ **Concept**	☑️ **Explanation**
`setTimeout()` min guarantee	Executes **after at least** the given time, **not exactly** after it
Single-threaded JS 🧵	Only one thing runs at a time (main thread)
Call Stack 📦	Holds currently executing functions
Callback Queue 🎟️	Queued async callbacks waiting for execution
Event Loop 🔁	Bridges the Callback Queue & Call Stack
Blocking code ❌	Delays everything — avoid long loops/sync ops on main thread

---

🧨 Golden Rule of JavaScript

> 🚫 **Never block the main thread** — JS is single-threaded. Blocking means *nothing else can run* (not even your precious `setTimeout()`!).
