setTimeout + Closures

The Famous Loop Interview Question

JavaScript Master Notes

1️⃣ The Problem Setup

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function () {
    console.log(i);
  }, i * 1000);
}
```

🔴 **Expected Output:**

```
1, 2, 3, 4, 5
```

🟢 **Actual Output:**

```
6, 6, 6, 6, 6
```

Reason

2️⃣ Why Does This Happen?

</> **`var` Scope:** The variable `var i` has **Function Scope** (it ignores the loop block entirely).

⏳ **Async Execution:** `setTimeout` is asynchronous. The loop finishes instantly, setting `i` to its final value of 6.

🗂 **Closure:** The callback function forms a closure and remembers the **reference** to `i`. When the timer runs 1 second later, it prints the current value of the reference, which is 6.

3️⃣ The Asynchronous Flow

Execution steps when using `var`:

1. The `for` loop executes completely (**Synchronously**). `i` becomes 6.
2. All five `setTimeout` callbacks are sent to the **Web API** to wait.
3. After the timers finish, the callbacks move to the **Callback Queue**.
4. The **Event Loop** transfers the callbacks to the Call Stack to run.
5. Each callback runs and prints the final, shared value of `i` (which is 6).

👉 **Key takeaway:** Synchronous code runs first; Asynchronous code always runs later.

✅ Fix 1: Use `let` (The Modern Way)

```javascript
for (let i = 1; i <= 5; i++) {
  setTimeout(function () {
    console.log(i);
  }, i * 1000);
}
```

Why it works?

`let` is **Block Scoped**.

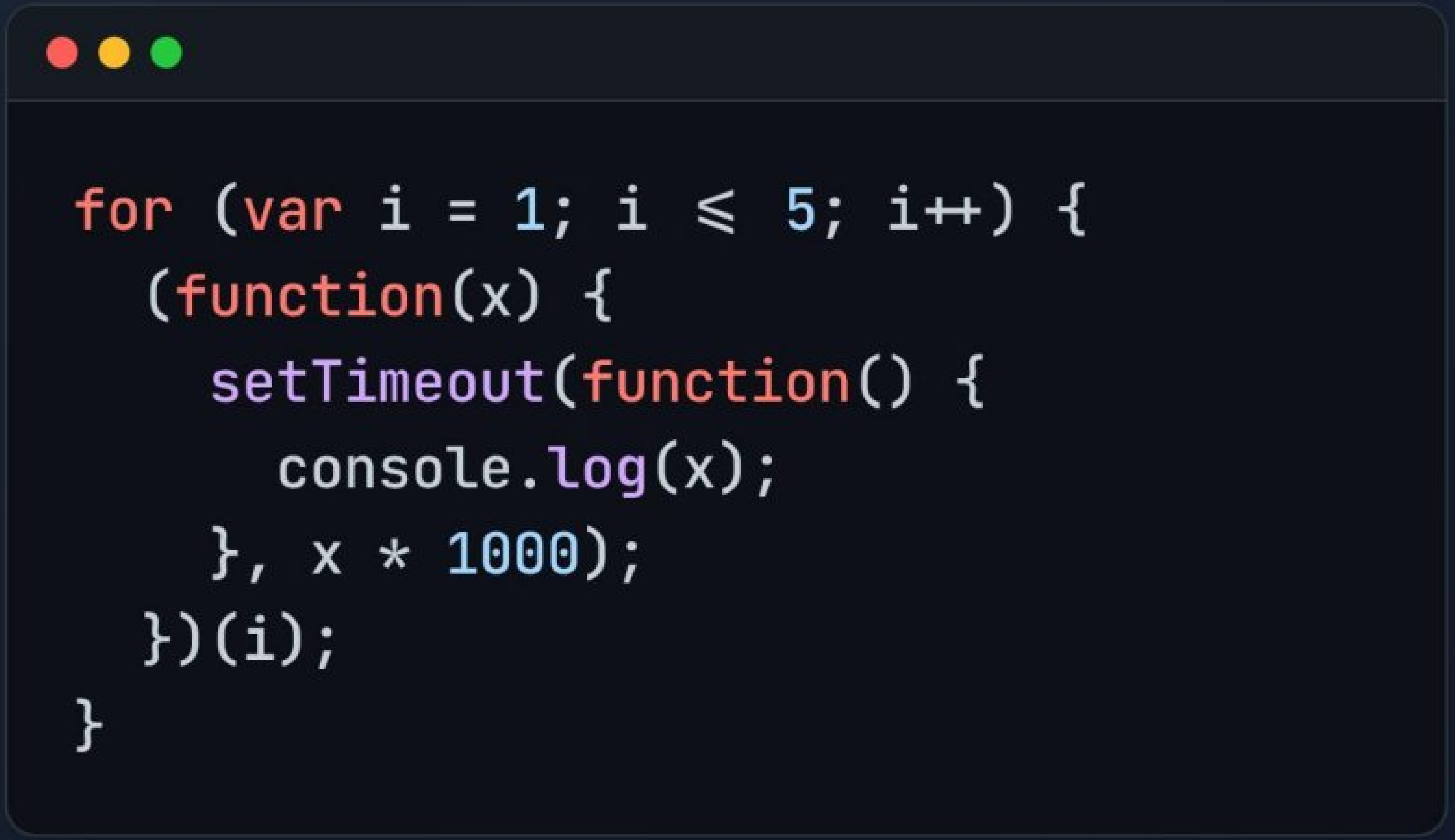
The `for` loop creates a **new block scope** (and a new variable `i`) in **each iteration**.

Each closure captures its own, correct copy of `i` (1, 2, 3, 4, 5).

Output: 1, 2, 3, 4, 5

Solution 2

✅ Fix 2: Using IIFE (The Old Way)



```
for (var i = 1; i <= 5; i++) {
  (function(x) {
    setTimeout(function() {
      console.log(x);
    }, x * 1000);
  })(i);
}
```

Why it works?

IIFE (Immediately Invoked Function Expression) creates a **new function scope** immediately.

We pass the current value of `i` into a local parameter `x`.

The closure captures this local `x` variable, which holds the correct value for that iteration.

✅ Fix 3: Using `bind`

```javascript
for (var i = 1; i <= 5; i++) {
  setTimeout(
    console.log.bind(null, i),
    i * 1000
  );
}
```

Why it works?

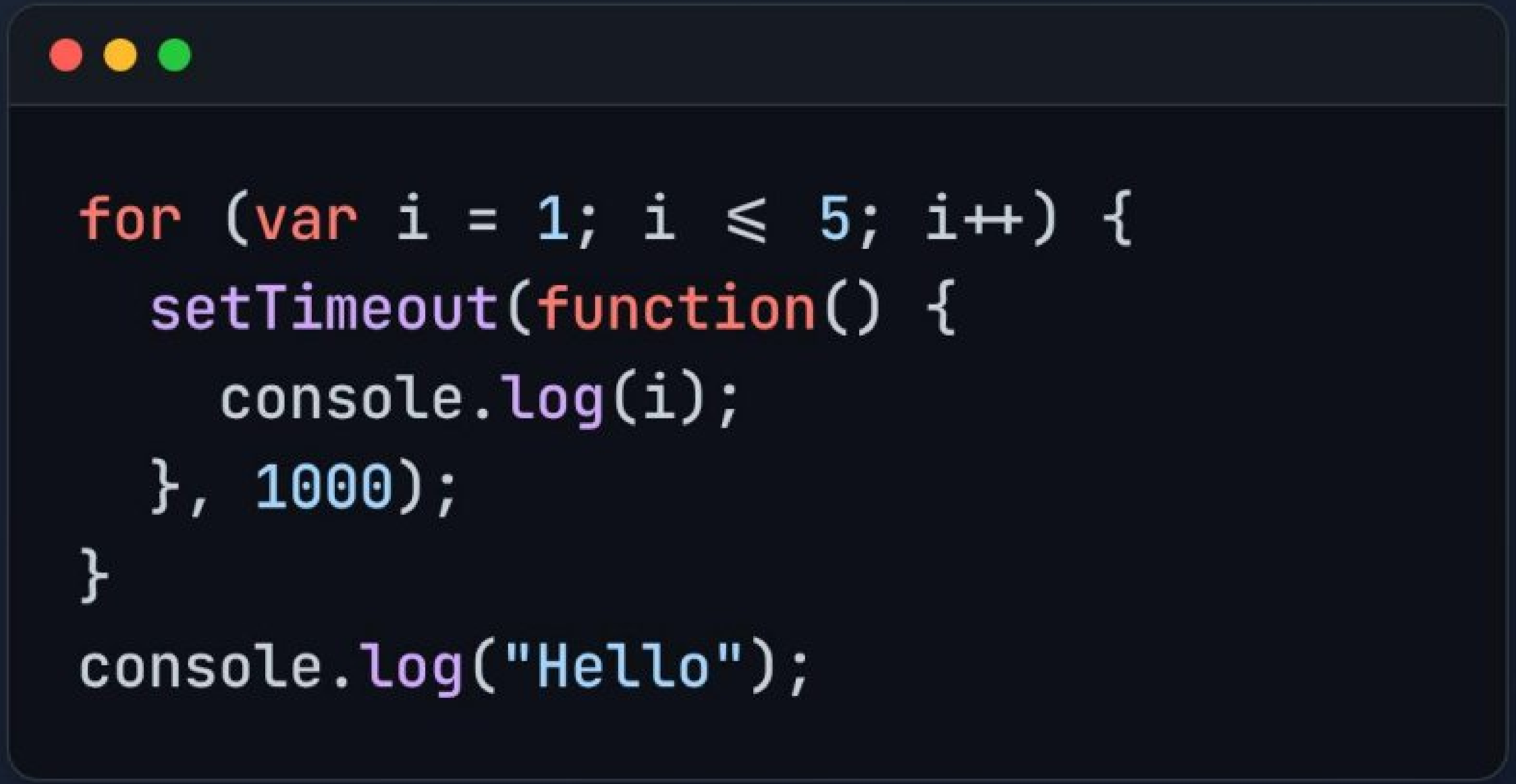
The `bind()` method creates a **new function**.

Crucially, it lets you set the arguments permanently (or "fix" them) when the new function is created.

The current value of `i` is passed as a fixed argument before the loop completes.

Tricky!

4️⃣ Interview Question: Execution Order



```
for (var i = 1; i <= 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
console.log("Hello");
```

Output Order:

1. **Hello** (Synchronous code runs first!)
2. **6, 6, 6, 6, 6** (Asynchronous code runs after a 1-second delay)

Event Loop Rule: The Call Stack must be empty before async tasks are executed.

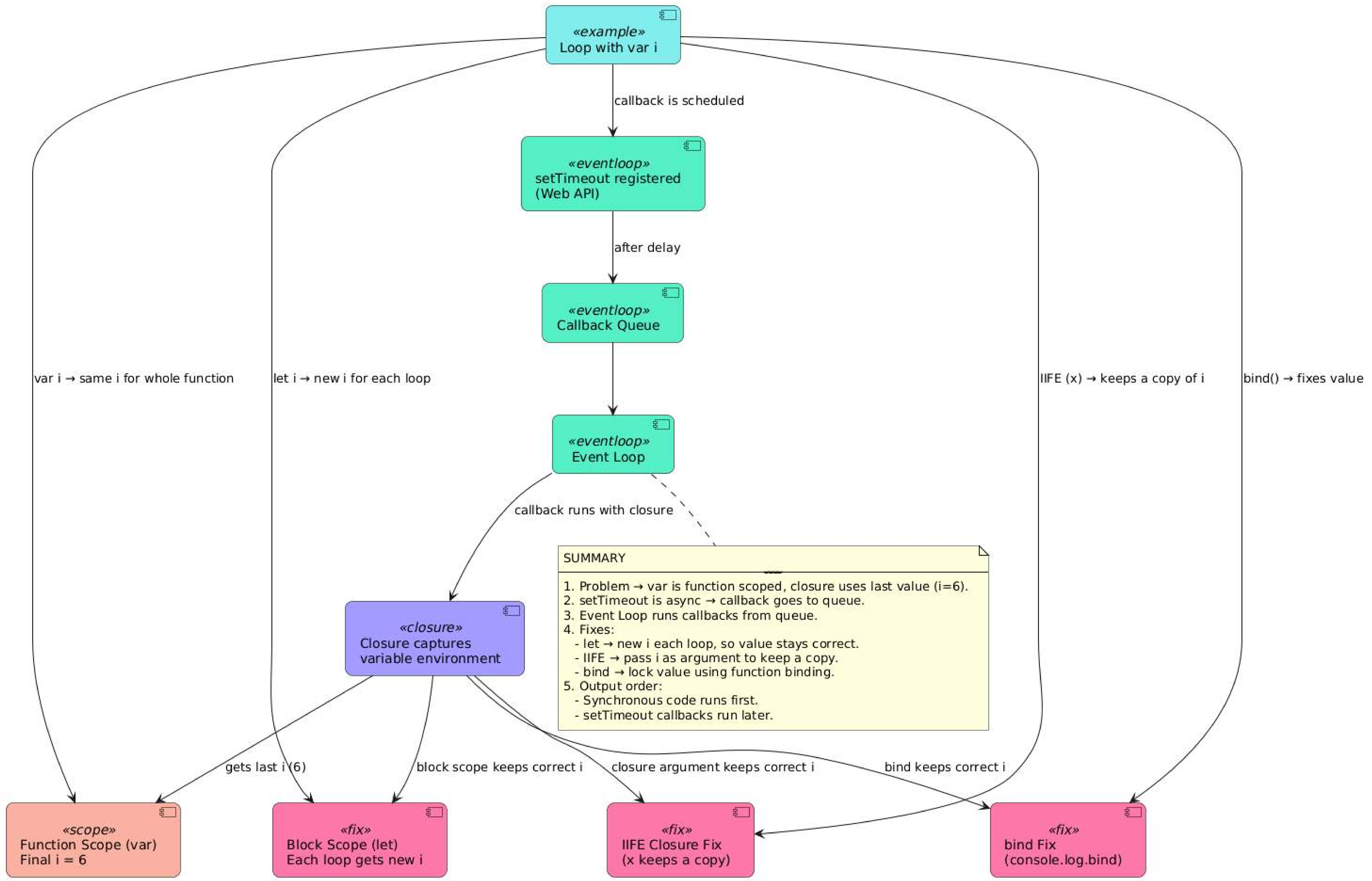
📝 Final Summary (Interview Ready)

🐞 **The Bug:** `var` is function-scoped. The closure captures the final, incorrect value (6).

💡 **The Core Fix:** You must create a **new scope** or **fix the value** for each iteration.

🧰 **The Solutions:** Use `let` (easiest), IIFE, or `bind()`.

setTimeout + Closures in Loop (Interview Hot Question)


«example» Loop with var i
callback is scheduled
«eventloop» setTimeout registered (Web API)
after delay
«eventloop» Callback Queue
«eventloop» Event Loop
callback runs with closure
var i → same i for whole function
let i → new i for each loop
IIFE (x) → keeps a copy of i
bind() → fixes value
«closure» Closure captures variable environment
gets last i (6)
block scope keeps correct i
closure argument keeps correct i
bind keeps correct i
«scope» Function Scope (var) Final i = 6
«fix» Block Scope (let) Each loop gets new i
«fix» IIFE Closure Fix (x keeps a copy)
«fix» bind Fix (console.log.bind)

SUMMARY

1. Problem → var is function scoped, closure uses last value (i=6).
2. setTimeout is async → callback goes to queue.
3. Event Loop runs callbacks from queue.
4. Fixes:
   - let → new i each loop, so value stays correct.
   - IIFE → pass i as argument to keep a copy.
   - bind → lock value using function binding.
5. Output order:
   - Synchronous code runs first.
   - setTimeout callbacks run later.

Thank You!

Mastered the Interview Trap?

🔔 Follow for More JavaScript Deep Dives!

By Ali Hassan