The Problem Recap

With `var`, all callbacks share the same variable binding. By the time the timeouts run, that single variable has been overwritten with the last value from the loop.

The Fix: Create a New "Box" Manually

We can do this by wrapping each iteration's value in its own **function scope** — either with an IIFE (Immediately Invoked Function Expression) or by passing the value as an argument to the callback.

Option 1 — IIFE Closure

Js Copy

```
for (var i = 1; i <= 3; i++) {
  (function(current) {
    setTimeout(() => console.log("var + IIFE:", current), 1000);
  })(i);
}
```

How it works:

- `(function(current) { ... })(i)` creates a brand-new function scope for each iteration.
- `current` is a new parameter variable inside that scope, holding the value of `i` at that moment.
- The arrow function closes over `current` (its own box), not the shared `i`.

Option 2 — Pass as Argument to `setTimeout`

Js Copy

```
for (var i = 1; i <= 3; i++) {
  setTimeout((val) => console.log("var + arg:", val), 1000, i);
}
```

How it works:

- `setTimeout` can take extra arguments after the delay.
- Here, `i` is passed as `val` to the callback when it runs.
- Each scheduled call gets its own copy of the value.

Connecting Back to `let`

- With `let`, the JavaScript engine **automatically** creates a new binding for each loop iteration — no IIFE or extra arguments needed.
- With `var`, you have to **manually** create a new scope to “freeze” the value for each callback.