1️⃣ Scope refresher in this context

- **Function scope (`var`)**: One variable binding for the whole function. Loops don't create a new one.
- **Block scope (`let`)**: A new variable binding is created **each time** the loop body runs.

2️⃣ How callbacks "remember" variables

When you pass a function (like the arrow function in `setTimeout`) to be run later, JavaScript doesn't copy the value — it keeps a **reference** to the variable in memory.
This is called a **closure**: the callback closes over the variable from its surrounding scope.

So:

- If all callbacks share the **same variable binding**, they'll all see whatever value it has at the time they run.
- If each callback has its **own variable binding**, they'll each see the value from when they were created.

3️⃣ Putting it together with your code

With `let item`

Js Copy

```
while (...) {
  let item = array.shift();
  setTimeout(() => console.log(item), 1000);
}
```

- Each loop iteration creates a **new** `item` **variable** (block scope).
- The callback closes over **that specific** `item`.
- After 1 second, each callback logs the value it captured at creation time.

Mental model:
Think of `let` as giving each callback its own labeled box with a value inside.
Even if the loop finishes, each box still has its original value.

With `var item`

Js Copy

```
while (...) {
  var item = array.shift();
  setTimeout(() => console.log(item), 1000);
}
```

- There is **only one** `item` **variable** for the whole function (function scope).
- Every callback closes over **the same box**.
- The loop runs to completion before the first timeout fires, so `item` ends up holding the **last value assigned**.
- All callbacks open the same box later — and see the same final value.

Mental model:
With `var`, all callbacks share one box. The loop keeps replacing what's inside that box. By the time callbacks open it, the last value is inside.

4 Visual timeline

Time	Action	`let` behavior	`var` behavior
0ms	Loop starts	Creates `item` #1, #2, #3...	Reuses same `item`
0–5ms	Loop ends	Each callback has its own `item`	One `item` now holds last value
1000ms	Callbacks run	Log: one, two, three	Log: three, three, three

5 Key connection

The **scope** (`let` vs `var`) determines **how many variable bindings exist**.
The **callback** (closure) determines **which binding it keeps a reference to**.
If there's only one binding (`var`), all callbacks share it.
If there's a new binding each iteration (`let`), each callback gets its own.