1 With `let item` (current code)

- `let` is **block-scoped** — each loop iteration gets its **own separate** `item` **variable**.
- When you do:

  Js Copy

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

  the arrow function "remembers" the value of `item` **from that specific iteration**.
- This means after 1 second, each `console.log(item)` prints the correct word that was removed in that iteration.

Example output after 1 second:

Code Copy

```
one
two
three
```

...and so on, in the right order.

2 With `var item`

- `var` is **function-scoped**, not block-scoped.
- All iterations inside the `while` loop share **the same single** `item` **variable**.
- The `setTimeout` callbacks don't run immediately — they run later, after the loop finishes.
- By the time the first timeout fires, the loop has already ended and `item` now holds **the last value assigned** in that loop.

So instead of each callback having its own `item`, they all "see" the same one — the last one set before the loop stopped.

What happens in your code:

- Suppose the first batch starts with `"one"`, `"two"`, `"three"`.
- With `var`, by the time the timeouts run, `item` will be `"three"` for all three callbacks.
- Output after 1 second would be:

Code Copy

```
three
three
three
```

— wrong, because they all share the same reference.

3️⃣ Why `let` works better here

- `let` creates a **new binding** for each loop iteration, so each scheduled callback gets the correct value.
- `var` would require extra workarounds (like an IIFE or passing `item` as an argument to `setTimeout`) to preserve the value.

🔍 Quick visual of the difference

Feature	`let` (block scope)	`var` (function scope)
Scope	New variable per loop iteration	One variable shared by all iterations
Callback value	Correct per iteration	All callbacks see the last value
Fix needed?	No	Yes — need closure trick