11. setTimeout + Closures Interview Question

📘 Index (Table of Contents)

❓ Questions and Answers

1. What is the Common Issue with `setTimeout` in Loops?

Q: What problem arises when using `setTimeout` inside a loop?

A: When `setTimeout` is used inside a loop with `var`, all the callbacks share the same reference to the loop variable. As a result, by the time the callbacks execute, the loop variable has already reached its final value, leading to unexpected behavior.

Example:

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
// Output:
// 6
// 6
// 6
```

```
// 6
// 6
```

Explanation: All five `setTimeout` callbacks reference the same `i`, which equals 6 after the loop completes. Therefore, each callback logs 6.

---

2. How Can We Fix This Issue Using `let`?

Q: How can using `let` resolve the issue?

A: Using `let` in the loop creates a new block scope for each iteration, ensuring that each callback captures its own reference to `i`.

Example:

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
// Output:
// 1
// 2
// 3
// 4
// 5
```

Explanation: Each `setTimeout` callback now logs the expected value of `i` due to the block-scoping behavior of `let`.

---

3. What is an Alternative Solution Using an IIFE?

Q: Is there another way to fix this issue without using `let`?

A: Yes, you can use an Immediately Invoked Function Expression (IIFE) to create a new scope for each iteration, thereby capturing the current value of `i`.

Example:

```
for (var i = 1; i <= 5; i++) {
  (function(i) {
    setTimeout(function() {
```

```
    console.log(i);
  }, i * 1000);
 })(i);
}
// Output:
// 1
// 2
// 3
// 4
// 5
```

Explanation: The IIFE creates a new scope for each iteration, passing the current value of `i` to the `setTimeout` callback, ensuring the correct value is logged.

🔑 Summary and Key Takeaways

- **Problem:** Using `setTimeout` inside a loop with `var` can lead to all callbacks sharing the same reference to the loop variable, causing unexpected behavior.
- **Solution 1:** Use `let` in the loop to create a new block scope for each iteration, ensuring each callback captures its own reference to the loop variable.
- **Solution 2:** Use an IIFE to create a new scope for each iteration, capturing the current value of the loop variable.