Day 4 function and clousers

Perfect 👍 Let's break **Day 4 – Functions & Closures** into a detailed roadmap so you fully revise, learn, and practice each concept:

🔹 Step 1: Function Declaration vs Expression

Function Declaration

```js
function add(a, b) {
  return a + b;
}
```

- Hoisted (can call before defining).
- Name is mandatory.

Function Expression

```js
const add = function(a, b) {
  return a + b;
};
```

- Not hoisted.
- Can be anonymous or named.

👉 **Key difference:** declarations are hoisted, expressions are not.

🔹 Step 2: IIFE (Immediately Invoked Function Expression)

- A function executed immediately after being created.

```js
(function() {
  console.log("IIFE runs immediately!");
})();
```

- Used for data privacy (before `let` / `const`, it was common).

- Helps avoid polluting global scope.

🔹 Step 3: Closures

Definition:
A closure is when a function "remembers" variables from its **lexical scope**, even when executed outside that scope.

Example:

```js
function outer() {
  let counter = 0;
  return function inner() {
    counter++;
    return counter;
  };
}

const increment = outer();
console.log(increment()); // 1
console.log(increment()); // 2
```

Here, `inner` closes over `counter` and keeps it alive.

🔹 Step 4: Implementations

✅ Counter with closures

```js
function createCounter() {
  let count = 0;
  return {
    increment() { count++; return count; },
    decrement() { count--; return count; },
    reset() { count = 0; return count; }
  };
}
```

```
const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
```

✅ once(fn) function

Runs function only once, then caches the result.

```js
function once(fn) {
  let called = false;
  let result;
  return function(...args) {
    if (!called) {
      result = fn.apply(this, args);
      called = true;
    }
    return result;
  };
}

const start = once(() => "Started!");
console.log(start()); // "Started!"
console.log(start()); // "Started!" (but not re-executed)
```

🔹 Step 5: Practice Problems

1. **Closure Timer**

```js
function makeTimer() {
  for (var i = 1; i <= 3; i++) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  }
}
makeTimer();
```

👉 Why does it log `4, 4, 4`? How to fix with `let` or closure?

2. **Private variables**
   Create a `bankAccount` closure with `deposit`, `withdraw`, and `getBalance`.
3. **Memoization**
   Use closure to implement `memoize(fn)` that caches results.
4. **Build your own once() again** but without using `apply`—with spread operator only.

🔹 Step 6: Reflection

- Why do closures matter in async code (`setTimeout`, promises)?
- How are closures used in **event handlers**?
- How are they used in **module patterns**?

Would you like me to **give you closure-based coding interview questions** right now (so you try solving), or should I first explain in detail with answers?

Is this conversation helpful so far?