Problem 1: The Persistent Counter (Initial Incorrect Attempt)

function createCounter() {

let count = 0;

function incrementCounter() {

count++;

return count;

}

return incrementCounter;

}

const counterA = createCounter();

console.log(counterA()); // Expected: 1, Actual: ?

console.log(counterA()); // Expected: 2, Actual: ?

const counterB = createCounter();

console.log(counterB()); // Expected: 1 (independent counter), Actual: ?  
  
It teaches how closures let a function "remember" variables from the outer function—even after the outer function has finished running.

problem 2:

JavaScript

function delayedGreeter(names) {

for (var i = 0; i &lt; names.length; i++) {

setTimeout(function() {

console.log(`Hello, ${names[i]}!`);

}, i \* 1000);

}

}

delayedGreeter([&#39;Alice&#39;, &#39;Bob&#39;, &#39;Charlie&#39;]);

// Expected output (with delays):

// Hello, Alice! (after 0 seconds)

// Hello, Bob! (after 1 second)

// Hello, Charlie! (after 2 seconds)

// Actual output: ?

This shows a common mistake in JavaScript: thinking each loop iteration has its own i. It doesn’t—unless you force it to using let or IIFE. You learn how closures work with async behavior and why using the right kind of variable (let vs. var) is important.

Section 2 : Task 2: Explanation of Hoisting

var: Fully hoisted. Declaration is hoisted and initialized to undefined, so accessing it before the line of code doesn't crash, but returns undefined.

let and const: Also hoisted, but not initialized. Accessing them before the declaration causes a ReferenceError. This is due to the Temporal Dead Zone, the period from the start of the scope until the line where the variable is declared.

A closure is when a function remembers variables from the outer scope even after that outer function has finished running.

In setupCounter, the returned increment and decrement functions keep a reference to their own count.