# 2665. Counter II

Easy

Leetcode Link

## Problem Description

The task here is to design a function named `createCounter` that when executed will return an object composed of three methods. This generated object empowers us to manage a counter's value, which is initialized using the parameter `init`. The `init` parameter is an integer that sets the starting point for the counter.

The three methods that must be present in the returned object enable different operations on the counter:

- `increment()`: This function should increase the current value of the counter by 1 whenever it is called, then return the updated value.
- `decrement()`: Conversely, this function is responsible for reducing the current value of the counter by 1 each time it is invoked, then returning the new, decreased value.
- `reset()`: This method serves to restore the counter's value back to the initial value specified by `init` when the counter was first created and should return the reset value.

The specified operations must adhere to encapsulation, ensuring the inner state of the counter cannot be directly modified from outside the object, except through these three functions. Implementing such functionality suggests an understanding of closure in JavaScript/TypeScript, as it's necessary to maintain the current value of the counter between function calls.

## Intuition

The solution requires the concept of closure, which allows a function to remember and access its lexical scope even when it's executed outside that lexical scope. In JavaScript and TypeScript, functions can be nested within other functions, and the inner function will have access to the variables declared in its surrounding (outer) scope.

By utilizing closures, we can create private variables that are only accessible to the nested functions. In this scenario, we declare a variable `val` inside the `createCounter` function, which holds the value of the counter. This `val` variable is not directly accessible from the outside but can be manipulated only through the methods `increment`, `decrement`, and `reset` that modify and return `val`.

The `increment` and `decrement` methods make simple use of the pre-increment and pre-decrement operators (`++` and `--`, respectively), which alter the value of `val` and then return it.

The `reset` method, on the other hand, reassigns `val` to the original `init` value, effectively resetting the counter.

The `createCounter` function then returns an object literal with these three methods. Each method has access to the `val` variable through closure, which allows these functions to maintain and modify the state of `val` without exposing it directly to the outside world, preserving data privacy and integrity.

## Solution Approach

The implementation of the `createCounter` function employs the closure feature of JavaScript/TypeScript, which enables us to create an environment where a private variable that stores the current value of the counter (`val`) is accessible only within the scope of the function that creates it.

Here's a breakdown of the algorithm and patterns used:

1. **Initialization:** When `createCounter` is called with an `init` value, we initiate a local variable `val` and assign it the value of `init`. This `val` acts as a private variable that holds the state of our counter.

2. **Encapsulation:** The object returned by `createCounter` encapsulates the `increment`, `decrement`, and `reset` functions. Each of these functions has a specific purpose, and all of them have access to the `val` variable through closure. This ensures that `val` cannot be modified directly from outside the object, maintaining encapsulation and data privacy.

3. **Increment Operation:** The `increment` method is straightforward; it uses the pre-increment operator (`++val`) to increase the value of `val` before returning it. This change persists within the closure so that subsequent calls to `increment` or the other methods will see the updated value.

4. **Decrement Operation:** Similarly, the `decrement` method uses the pre-decrement operator (`--val`) to decrease `val` by 1 and then returns the new value.

5. **Reset Operation:** The `reset` method sets `val` back to the initial value (`init`) and then returns it. This restores the counter to its initial state.

6. **Closure:** Importantly, each of these methods is a closure; they all 'remember' the environment in which they were created (which includes the `val` variable). This is how they maintain the current value of the counter between function calls without exposing `val` directly.

Understanding how closures work is essential in this solution because they provide the mechanism to retain and manipulate the current value of the counter in a controlled way. The local variables within the `createCounter` function form a state that persists across the function calls through the returned object methods.

This pattern of using closures to maintain a private state is common in JavaScript and TypeScript for creating encapsulated structures, which is a foundational concept in functional programming.

In summary, the solution approach capitalizes on the language's functional programming capabilities, utilizing closures for state management while ensuring that the state remains private and mutable only through specific methods.

### Example Walkthrough

Let's take for an instance we call `createCounter` with an `init` value of 5. Here's how we might walk through the solution approach:

1. **Initialization:** We call `createCounter(5)`. Inside `createCounter`, a variable `val` is set to 5. This variable `val` is the counter's current value and is private.

2. **Encapsulation:** `createCounter` returns an object with three methods: `increment`, `decrement`, and `reset`. Each of these methods can operate on `val`, but nothing outside of these methods can alter `val`.

3. **Increment Operation:** We call the `increment` method on the returned object. It uses `++val`, so `val` becomes 6, and this value is returned. If we call `increment()` again, `val` then becomes 7.

4. **Decrement Operation:** After a couple of increments, we call the `decrement` method. If `val` is currently at 7, `decrement` uses `--val`, decreasing the value to 6, and returns this new value.

5. **Reset Operation:** At any point, if we call the `reset` method, `val` is set back to the initial value, which is 5 in this example. Calling `reset()` would return this initial value.

6. **Closure:** Throughout the operations, our counter maintains the current value because `increment`, `decrement`, and `reset` are closures. They have access to `val` from their creation environment, even though the environment is not accessible directly.

By following this walkthrough with an initial value of 5, it becomes clear how each call influences the state of the counter and how the counter's value progresses through the operations, while still being encapsulated within the closure created by `createCounter`.

## Python Solution

```python
# Define the operations that the Counter object will support
class CounterOperations:
    def increment(self) -> int:
        pass

    def decrement(self) -> int:
        pass

    def reset(self) -> int:
        pass

# The current value of the counter
current_value: int

# Initialize the counter with a specific value
def initialize_counter(initial_value: int) -> None:
    global current_value
    current_value = initial_value

# Increment the counter and return the new value
def increment_counter() -> int:
    global current_value
    current_value += 1
    return current_value

# Decrement the counter and return the new value
def decrement_counter() -> int:
    global current_value
    current_value -= 1
    return current_value

# Reset the counter to the initial value and return it
def reset_counter(initial_value: int) -> int:
    global current_value
    current_value = initial_value
    return current_value

# Create a counter with the given initial value and return an object with operation methods
def create_counter(initial_value: int) -> CounterOperations:
    initialize_counter(initial_value)

    # This inner class represents a Counter and implements the CounterOperations interface
    class Counter(CounterOperations):
        def increment(self) -> int:
            return increment_counter()

        def decrement(self) -> int:
            return decrement_counter()

        def reset(self) -> int:
            return reset_counter(initial_value)

    # Return an instance of the nested Counter class
    return Counter()

# Example usage:
# counter = create_counter(5)
# print(counter.increment())  # Outputs: 6
# print(counter.reset())      # Outputs: 5
# print(counter.decrement())  # Outputs: 4

```

## Java Solution

```java
// Interface defining the operations of a counter
interface CounterOperations {
    int increment();
    int decrement();
    int reset();
}

// Class that encapsulates the logic of a counter
class Counter implements CounterOperations {
    // The current value of the counter
    private int currentValue;

    // The initial value of the counter for reset purposes
    private int initialValue;

    // Constructor to initialize the counter with a specific value
    public Counter(int initialValue) {
        this.initialValue = initialValue;
        this.currentValue = initialValue;
    }

    // Increment the counter and return the new value
    @Override
    public int increment() {
        return ++currentValue;
    }

    // Decrement the counter and return the new value
    @Override
    public int decrement() {
        return --currentValue;
    }

    // Reset the counter to the initial value and return it
    @Override
    public int reset() {
        currentValue = initialValue;
        return currentValue;
    }
}

// The main class to demonstrate the usage of the Counter class
public class Main {
    public static void main(String[] args) {
        // Create a counter with initial value 5
        CounterOperations counter = new Counter(5);

        // Increment the counter and output the value
        System.out.println(counter.increment());  // Outputs: 6

        // Reset the counter and output the value
        System.out.println(counter.reset());      // Outputs: 5

        // Decrement the counter and output the value
        System.out.println(counter.decrement());  // Outputs: 4
    }
}

```

## C++ Solution

```cpp
#include <iostream>

// Class to manage counter operations
class CounterOperations {
private:
    int currentValue; // The current value of the counter

public:
    // Constructor initializes the counter with a specific value
    CounterOperations(int initialValue) : currentValue(initialValue) {}

    // Increment the counter and return the new value
    int increment() {
        return ++currentValue;
    }

    // Decrement the counter and return the new value
    int decrement() {
        return --currentValue;
    }

    // Reset the counter to the initial value and return it
    int reset() {
        currentValue = initialValue;
        return currentValue;
    }
};

// Function to create a counter with the given initial value
// and return an instance with operation methods
CounterOperations createCounter(int initialValue) {
    CounterOperations counter(initialValue);
    return counter;
}

// Example usage:
int main() {
    // Create a counter starting at 5
    CounterOperations counter = createCounter(5);
    std::cout << counter.increment() << std::endl; // Outputs: 6
    std::cout << counter.reset() << std::endl;     // Outputs: 5
    std::cout << counter.decrement() << std::endl; // Outputs: 4
    return 0;
}

```

## Typescript Solution

```typescript
// Define the type for the object returned by 'createCounter' function
type CounterOperations = {
    increment: () => number;
    decrement: () => number;
    reset: () => number;
};

// The current value of the counter
let currentValue: number;

// Initialize the counter with a specific value
function initializeCounter(initialValue: number): void {
    currentValue = initialValue;
}

// Increment the counter and return the new value
function incrementCounter(): number {
    return ++currentValue;
}

// Decrement the counter and return the new value
function decrementCounter(): number {
    return --currentValue;
}

// Reset the counter to the initial value and return it
function resetCounter(initialValue: number): number {
    currentValue = initialValue;
    return currentValue;
}

// Create a counter with the given initial value and return an object with operation methods
function createCounter(initialValue: number): CounterOperations {
    initializeCounter(initialValue);
    return {
        increment: incrementCounter,
        decrement: decrementCounter,
        reset: () => resetCounter(initialValue),
    };
}

// Example usage:
// const counter = createCounter(5)
// console.log(counter.increment()); // Outputs: 6
// console.log(counter.reset());     // Outputs: 5
// console.log(counter.decrement()); // Outputs: 4

```

## Time and Space Complexity

### Time Complexity

The functions within `createCounter`:

- `increment()` has a time complexity of `O(1)` because it performs a single operation of incrementing the `val` variable.
- `decrement()` also has a time complexity of `O(1)` due to a single operation of decrementing the `val`.
- `reset()` has a time complexity of `O(1)` as it assigns the `init` value to `val`.

Therefore, all methods provided by `createCounter` are constant time operations.

### Space Complexity

The space complexity of `createCounter` is `O(1)` because it uses a fixed amount of space:

- A single closure is created that captures the `val` variable.
- No additional space that grows with the input size is used inside `createCounter`. The functions increment, decrement, and reset do not allocate any additional memory that depends on the size of the input or number of operations performed.

In summary, the space requirement remains constant regardless of the initial value or the number of operations performed.