3.2P: Answer Sheet

Recall task *2.2P Counter Class* and answer the following questions.

1. How many *Counter* objects were created?

> According to the task 2.2P Counter Class, there are three objects were created in *Counter* class including:
> - *myCounters[0]* named "Counter 1".
> - *myCounters[1]* named "Counter 2".
> - *myCounters[2]* assigns a reference to the *Counter* object that already exists at *myCounters[0]*

2. Variables declared without the ***new*** keyword are different to the objects created using ***new***. In the ***Main*** function, what is the relationship between the variables initialized with and without the ***new*** keyword?

> From an object-oriented perspective, variables instantiated with the new keyword result in the creation of discrete, autonomous objects within memory allocation. Specifically, in the context of the Main function, the elements myCounters[0] and myCounters[1] represent separate instances of the Counter class, each occupying a unique memory address.
>
> Conversely, variables assigned through reference assignment—absent the new keyword—do not instantiate new objects; instead, they establish references to existing objects. For instance, when myCounters[2] is assigned the value of myCounters[0] without invoking new, both variables refer to the same underlying object in memory, thereby sharing the same state and behavior.

3. In the ***Main*** function, explain why the statement ***myCounters***[2].***Reset***(); also changes the value of ***myCounters***[0].

> In my view, invoking my ***myCounters***[2].***Reset***() also modifies the value of ***myCounters[0]*** because both references point to the same underlying object. When ***Reset()*** is called on ***myCounters[2]***, it operates directly on this shared instance, resetting its internal _count field to zero. Consequently, since ***myCounters[0***] is merely another reference to the identical object, any subsequent access to ***myCounters[0].Ticks*** will accurately reflect the updated state of the shared object, demonstrating that the change is inherently reflected across all references to that instance.

4. The difference between *heap* and *stack* is that heap holds "*dynamically allocated memory*." What does this mean? In your answer, focus on the size and lifetime of the allocations.

In my point of view, "Dynamically allocated memory" refers to memory that a program sets aside while it's running. Additionally, the difference between heap and stack in holding "dynamically allocated memory" is shown below:

- Heap Memory:
  + Size: You can ask for different amounts of memory as needed.
  + Lifetime: It can last beyond the function that created it.
- Stack Memory:
  + Size: It's fixed and determined when the program is written.
  + Lifetime: It's automatically freed when the function using it ends

5. Are objects allocated on the heap or on the stack? What about local variables?

From my perspective, objects are allocated on the heap and local variables are allocated on the stack

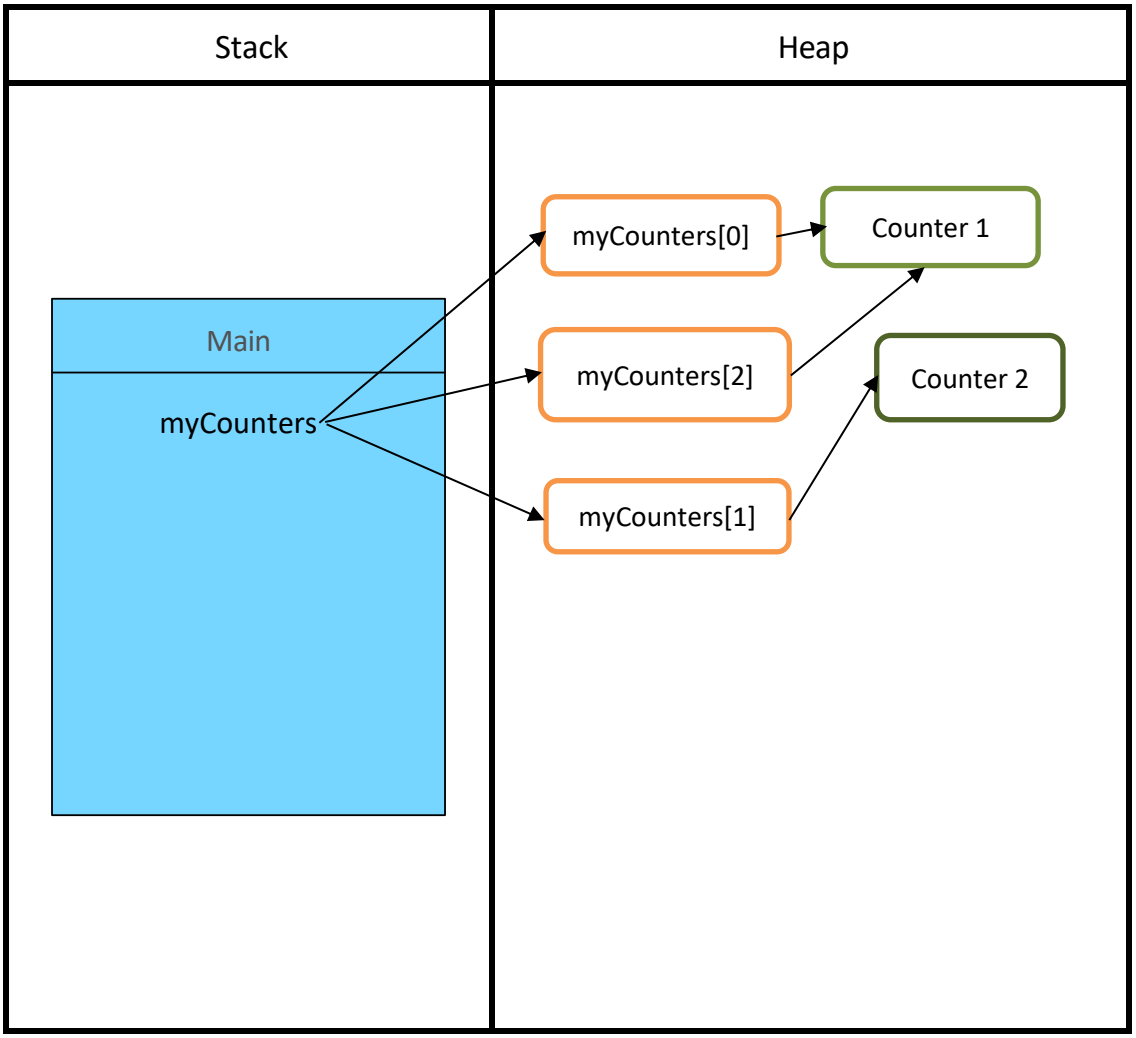
6. What is the meaning of the expression ***new*** *ClassName*(), where *ClassName* refers a class in your application? What is the value of this expression?

In my understanding, the expression ***new*** *ClassName(),* where ClassName denotes a class within the application, signifies the creation of a new object instance of that class. This process involves invoking the class's constructor to initialize its fields and establish the object's initial state. Moreover, the value of this expression is a reference to the freshly instantiated object, representing a distinct and independently initialized entity in memory.

7. Consider the statement "*Counter* ***myCounter***;". What is the value of ***myCounter*** after this statement? Why?

From my perspective, the value of **myCounter** after this statement is unassigned because In C#, local variables of reference types (like Counter) are not automatically initialized to null by the compiler. Instead, they are initially unassigned. The C# compiler enforces a "definite assignment" rule, meaning you cannot use myCounter until it has been explicitly assigned a value (either an object reference using new or null). Attempting to use an unassigned local variable will result in a **compile-time error**.

8. Based on the code you wrote in task *2.2P Counter Class*, draw a diagram showing the locations of the variables and objects in function ***Main*** and their relationships to one another.



9. **If the variable myCounters is assigned to null, then you want to change the value of myCounters[X], where X is the last digit of your student ID, what will happen? Please provide your observation with screenshots and explaination.**

If the variable myCounters is assigned to null, then you want to change the value of myCounters[X], where X is the last digit of your student ID, a **System.NullReferenceException** will occur at runtime.

When **myCounters** is assigned null, it no longer points to any array object. Attempting to access an element like **myCounters**[X] on a null reference causes a **System.NullReferenceException**. This occurs because there's no array object to perform the indexing operation on. I cannot provide screenshots.

Hint. You may want to read this material for this task
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/null

For further reading at your own.

- Null pointer CrowdStrike Bug, https://www.thestack.technology/crowstrike-null-pointer-blamed-rca/
- CrowdStrike Blog, https://www.crowdstrike.com/blog/tech-analysis-channel-file-may-contain-null-bytes/