



# Object Oriented Programming

## Pass Task 2.2: Counter Class and Arithmetic Overflow-checking

### Overview

In this task you will create a *Counter* class and use it to create and work with *Counter* objects.

**Purpose:** Practice with properties and use object-oriented encapsulation.

**Task:** Implement a program that creates and uses a number of counters to explore how objects work. **The task contains personalized requirements.**

**Deadline:** Due by the end of week three, **Fri, 23 May 2025, 23:59 Hanoi Time. (Firmed)**

### Submission Details

All students have access to the Adobe Acrobat tools. Please print your solution to PDF and combine it with the screenshots taken for this task.

- C# code files of the classes created.
- Screenshot of output.

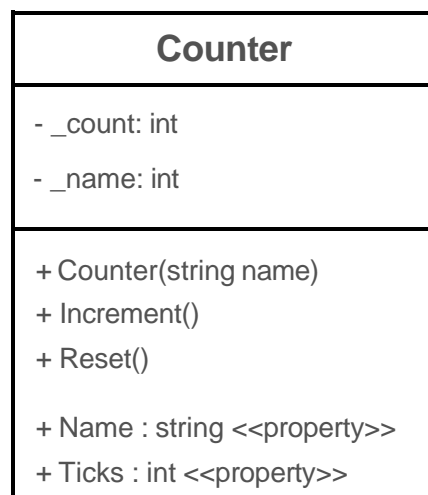
## Instructions

In this task you will create a *Counter* class and explore how fields can be used by an object to store and maintain information.

Each *Counter* object:

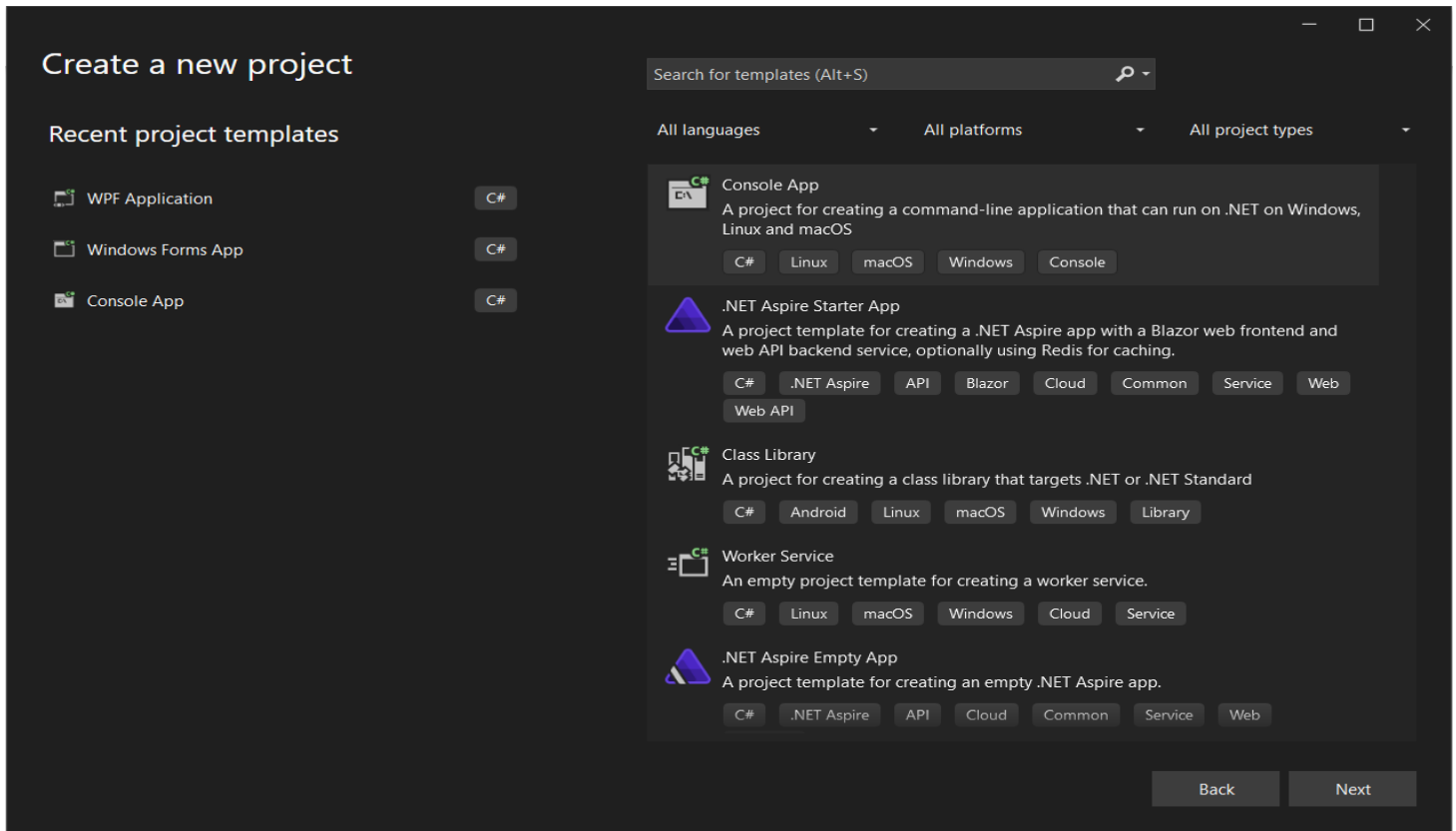
- Knows its *count* - by using a **\_count** field to store an integer value,
- Knows its *name* - by using a **\_name** field to store a string value,
- Can generated by using the *constructor* with a string parameter **name** that initializes the object's **\_count** field to zero and sets the object's **\_name** field to the value of **name**,
- Can increment object's **\_count** field by one using the **Increment** method,
- Can reset itself by using the **Reset** method that sets the **\_count** field to 0,
- Can give you its name via the **Name** property (i.e., get {...}),
- Can change its name via the **Name** property by assigning it a new value (i.e., set {...}),
- Can give you its value via the **Ticks** property (i.e., get {...}).

The following UML class diagram shows the basic outline for this class.



**Note:** The << ... >> annotations in UML are known as stereotypes. They are used to add notes to aspects of the diagram. In this case, <<property>> notes that the **Name** attribute here is a property. Properties are *virtual fields*, that is, properties may or may not mapped to actual instance variables. Properties can be read-only, write-only, or read-write. In class *Counter*, property **Name** is read-write, whereas property **Ticks** is read-only.

1. Create a new *Console App* and name it **CounterTask**.



Project name

CounterTask

2. Create a new *Counter* class.

```
public class Counter
{
    // ...
}
```

3. Add the private ***\_count*** and ***\_name*** fields, enabling a *Counter* object to *know* its count and name values.

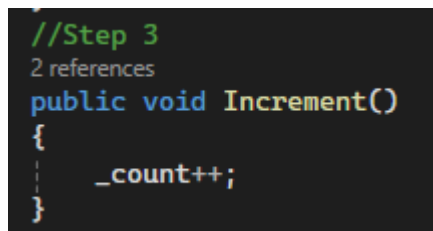
```
private int _count; /
private string _name;
```

4. Change the constructor so that it takes a string parameter that is used to set the ***\_name*** field of the *Counter* object, and assign 0 to the ***\_count*** field.

```
public class Counter
{
    private int _count;
    private string _name;

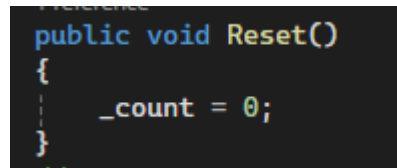
    public Counter(string name)
    {
        _name = name;
        _count = 0;
    }
}
```

5. Add the **Increment** method that increases the value of the **\_count** field by one.



```
//Step 3
2 references
public void Increment()
{
    _count++;
}
```

6. Add a **Reset** method that assigns 0 to the **\_count** field.



```
public void Reset()
{
    _count = 0;
}
```

You have now created the code needed to work with *Counter* objects. Each *Counter* object knows its *count* and *name* and can increment and reset its count value. Notice, the things a *Counter* object knows are *hidden* within the object (due to the **private** modifier on the fields). This is one of the guiding principles of *object-oriented encapsulation*. Object-oriented encapsulation is a mechanism that allows you to hide specific information and control access to the object's internal state. In general, you achieve object-oriented encapsulation by making all instance variables **private**, and provide read or write access to instance variables via **public** methods only.

The keywords **private** and **public** serve as scope modifiers in C#. A feature marked **private** is only visible within the scope of the defining class and its objects. Features marked **public** are visible to all clients of a class and its objects. The term client refers to other classes, other objects, and even other applications. You should aim at achieving a suitable balance between public and private visibility within a class and its objects. Object-oriented encapsulation is not a mechanism whose principles are set in stone – sometimes it can be beneficial to loosen access restrictions if the domain abstraction calls for it.

C# includes a feature, called *properties*, that allows you to provide access to data in a controlled way. From the outside, properties look and feel like instance variables of an object. Hence, they are also known as *virtual fields*. However, properties are actually mapped to a pair of methods:

**get** – to retrieve a value, and **set** – to update a value. Properties are not simply used to provide access to instance variables. Instead, you often use properties to perform safety checks or compute composite values for a set of attributes.

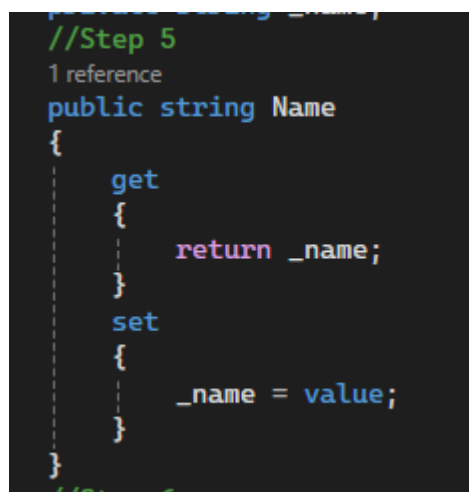
7. Create a **Name** property for *Counter* objects using the following code:

```
public class Counter
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}
```

Properties have the general format as shown below. However, you can add any code you want within the get and set methods, as long as get returns a value and set changes a value.

```
public [TYPE] PropertyName
{
    get
    {
        return ...
    }
    set
    {
        ... = value;
    }
}
```



The screenshot shows a code editor with the following C# code for the `Name` property of the `Counter` class. The code is highlighted with a dashed box. Above the code, there is a comment `//Step 5` and a line `1 reference`. Below the code, there is a comment `//Step 6`.

```
//Step 5
1 reference
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}
//Step 6
```

8. Create the **Ticks** property for the *Counter* class. **Ticks** is a read-only property that returns

**Hint:** Read-only properties have only a ***get*** method, write-only properties have only a ***set*** method.

Your *Counter* class is now complete. Build your solution and fix any error before you proceed.

```
public double Ticks //Change int to double
{
    get
    {
        return _count;
    }
}
```

9. Return to the *Program.cs* file.

10. Implement the following pseudocode for the static method **PrintCounters** method:

```
PrintCounters(counters)
1: // parameter counters is an array of Counter objects
2: foreach c in counters
3:     Tell Console to WriteLine with the format "{0} is {1}"
4:         and the result of Tell c to Name
5:         and the result of Tell c to Ticks
```

### Tips:

To declare a static method, you need to annotate the signature of the method with the keyword **static**. For example, **static public void Print** ( string[] names ) { ... }

**Foreach** loops are a simple way of traversing over all of the elements of an array in C#. For example **foreach** (string name **in** names) { ... }

The loop variable *c* in the pseudocode is a *Counter* object. In C#, you need to write *Counter c* so that the loop variable has a proper type.

**Note:** Console's **WriteLine** method can take a variable number of parameters. The {0} marker means inject the 1<sup>st</sup> value following the string at this point. For example:

```
Console.WriteLine("Hello, {0}{1}", "World", "!");
```

Please note, both **Main** and **PrintCounters** are static methods. Static methods are not associated with any object. Hence, you do not need to create a *Program* object to use **PrintCounters**.

```
internal class Program
{
    private static void PrintCounters(Counter[] counters)
    { ... }

    static void Main(string[] args)
    { ... }
}
```



```
//Step 7
2 references
private static void PrintCounters(Counter[] counters)
{
    foreach (Counter counter in counters)
    {
        Console.WriteLine($"{counter.Name}: {counter.Ticks}");
    }
}
```

11. Use the following pseudocode to implement the **Main** method.

```
Main()
1: Let myCounters be an array of three Counter objects
2: myCounters[0] := new Counter with name "Counter 1"
3: myCounters[1] := new Counter with name "Counter 2"
4: myCounters[2] := myCounter[0]
5: for i := 1 to 9
6:     Tell myCounters[0] to Increment
7: for i := 1 to 14
8:     Tell myCounters[1] to Increment
9: Tell Program to PrintCounters(myCounters)
10: Tell myCounters[2] to Reset
11: Tell Program to PrintCounters(myCounters)
```

```
namespace Clock
{
    0 references
    internal class Program
    {
        6 references
        private static void PrintCounter(Counter counter)
        {
            Console.WriteLine($"Counter Name: {counter.Name}, Ticks: {counter.Ticks}");
        }
        0 references
        static void Main(string[] args)
        {
            Counter[] myCounters = new Counter[3];
            myCounters[0] = new Counter("Counter 1");
            myCounters[1] = new Counter("Counter 2");
            myCounters[2] = myCounters[0];

            for (int i = 0; i <= 9; i++)
            {
                myCounters[0].Increment();
            }

            for (int i = 1; i <= 14; i++)
            {
                myCounters[1].Increment();
            }

            Console.WriteLine("Print counters before reseting myCounters[2]");
            PrintCounter(myCounters[0]);
            PrintCounter(myCounters[1]);
            PrintCounter(myCounters[2]);
            Console.WriteLine("-----");
            Console.WriteLine("Print Counters after reseting myCounter[2]");
            myCounters[2].Reset();
            PrintCounter(myCounters[0]);
            PrintCounter(myCounters[1]);
            PrintCounter(myCounters[2]);
        }
    }
}
```

In pseudocode, a **for**  $i := 0$  **to** 3 statement has four iterations:  $i == 0$ ,  $i == 1$ ,  $i == 2$ , and  $i == 3$ .  
How many iterations does **for**  $i := 1$  **to** 4 have?

From my perspective, there are four iterations include:

$i == 0$   
 $i == 1$   
 $i == 2$   
 $i == 3$

The loop starts from  $i == 1$  (initial value) and continues until to reach value  $i == 4$ , incrementing by 1 in each step.

**Hint:** You can declare the array using

```
Counter[] myCounters = new Counter[3];
```

12. Create a new Reset method, named, `ResetByDefault`, in the Counter class to reset the corresponding count to the integer valued at 2147483647XXX, where the XXX is the last three digits of your student ID.

```
public void ResetByDefault()
{
    _count = 214783647520;
}
```

13. Tell the Counter to increase the count value. Does the code still run without any bugs/crash? What is the reason behind? You can provide the answers as the comments in the code.

From my perspective, the code still run without any bugs or crashes because the `Increment()` method adds 1 to the `_count` variable, which is now a double. Additionally, doubles can handle both integer and floating-point values, and the increment operation is a valid arithmetic operation for this data type. As a result, there are no type-related errors or operations that would lead to a program crash in this scenario.

**Hint.** Please read Microsoft Learn portal

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked>

14. Compile and run your program.



```
Microsoft Visual Studio Debug Console
Print counters before resetting myCounters[2]
Counter Name: Counter 1, Ticks: 10
Counter Name: Counter 2, Ticks: 14
Counter Name: Counter 1, Ticks: 10
-----
Print Counters after resetting myCounter[2]
Counter Name: Counter 1, Ticks: 0
Counter Name: Counter 2, Ticks: 14
Counter Name: Counter 1, Ticks: 0
D:\University\May_2025\COS20007_Object_Oriented_Programming\Weekly_exercises\3.1P\Clock\Clock\bin\Debug\net8.0\Clock.exe
(process 16840) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Once your program is complete you can prepare it for your portfolio. This can be placed in your portfolio as evidence of what you have learnt.

1. Review your code and ensure it is formatted correctly.
2. Run the program and use your preferred screenshot program to take a screenshot of the Terminal showing the program's output.
3. Save and backup your work to multiple locations, if possible.
  - Once your program is working you do not want to lose your work.
  - Work on your computer's storage device most of the time, but backup your work when you finish each task.
  - You may use a cloud storage provider to safely store your work.
  - USB and portable hard drives are good secondary backups, but there is a risk that the drive gets damaged or lost.

**Note:** Each week you should aim to submit *all tasks*. **Submit** this task once it is complete. The assessment criteria give you a list of things to check before you submit.

### ***Assessment Criteria***

Make sure that your task has the following in your submission:

- The "Universal Task Requirements" (see Canvas) have been met.
- Your program (as text or screenshot).
- Screenshot of output.