# Object Oriented Programming

## Pass Task 2.1: Counter Object **Overview**

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

**Purpose:** Explore the connected nature of objects, and to work with properties and encapsulation.

**Task:** You will implement a program that creates and uses a number of counters to explore how objects work.

**Time:** This task should be completed before the start of week 3.

**Resources:**

- C# Station Tutorials
  - Lesson 1 to Lesson 5
  - Encapsulation and Properties
- Tutorials Point
  - C# Programming Tutorials
  - C# Programming Quick Guide
- Any C# books chapters on:
  - Types, Operators, Control Flow, Method declarations
- UML Class Diagrams Tutorial by Robert C. Martin
- Swinburne Videos on iTunesU
  - Quick Start with C-style syntax
  - Introducing Objects

### *Submission Details*

You must submit the following files to Canvas:

- C# code files of the classes created.
- Screenshot of output.
- Answers to questions in provided answer sheet.

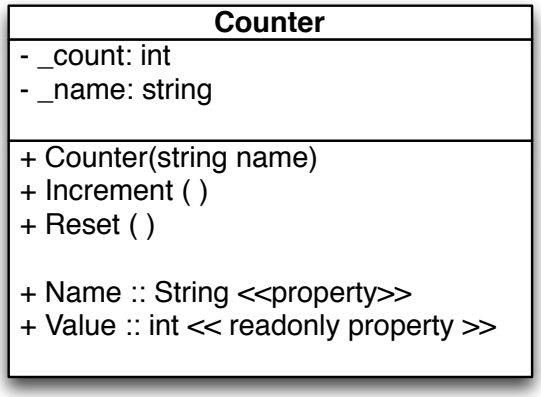
# Instructions

In this task you will create a Counter class to examine how fields can be used by an object to remember information.

Each Counter will:

- Know its **count** - by using a _count field to store an integer value.
- Know its **name** - by using a _name field to store a string
- Be able to be constructed with a name - by using a **constructor** with a string parameter, that initialises the object's _count field to zero, and sets the object's name.
- Be able to increment - an **Increment** method that increases the object's _count field by one.
- Be able to reset itself - a **Reset** method that sets the _count field to 0.
- Be able to give you its name - via a **Name** property
- Be able to change its name - via a **Name** property
- Be able to give you its value - via a **value** property (read-only)

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 — which is a combination of a get and a set method that is used to access a value from an object.

1. Create a new **Console Project**, name it **CounterTest**.
2. Create a new **Counter** class.
3. Add the **private** _count and _name fields, enabling the Counter to *know* its count and name values.
4. Change the **constructor** so that it takes a string parameter that is used to set the name of the Counter, and also assigns 0 to the _count field.

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

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

5. Add an **Increment** method that increases the value of the _count field by one.
6. Add a **Reset** method that assigns 0 to the *_count* field.

At this point you have created the code needed to build Counter objects, where each Counter knows it count and name, and can increment and reset it count value. The main problem now is that the things the Counter know are hidden within the object (due to the *private* modifier on the fields). This is a part of the idea of **encapsulation**. Encapsulation means "to place within a capsule". Here you can picture Counter objects as capsules with aspects that are hidden inside (private) and others that are accessible (public).

C# includes scope modifiers that allow you to indicate if features are available outside of the class, or whether they are enclosed within the object. Features marked as **public** are available outside the object, whereas **private** features exist entirely within the object. You should aim to keep as much *private* as you can, but you do need to make some aspects public. In this case the methods need to be public so that others can tell the object what to do, but the fields should be private as others should not be able to directly change these values. So, how can we get data out of the object without exposing the field directly.

C# includes a feature, called **properties**, that allows you to provide access to data in a controlled way. Externally (outside the object) these *look* and *feel* like data, but inside they are actually a pair of methods: one to get and another to set the value. This provides a convenient way of giving other objects access to an objects data, without them actually having direct access to the fields themselves. The methods can include things like validation code, and you can provide only the get method to make it read only, or the set method to make it write only.

7. Create a **Name** property for the Counter 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 should change a value.

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

This gives you control over how the property is read and written. For example, you can provide validation code in the set, or calculate the value in the get.

8. Create the **Value** property for the Counter objects. It should return the value from the _count field, but not have a set part to make it read only.

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

Now you have created the Counter class, it can be used to create Counter objects.

9. Switch to the MainClass in Program.cs.
10. Implement the following pseudocode for a PrintCounters static method:

```
Static Method: Print Counters
--------------------
Parameters:
 - counters: array of references to Counter objects
Local Variables:
 - c: a reference to a Counter (declared in the foreach loop)
--------------------
Steps:
 1: Loop for each Counter c in counters
 2:    Tell Console, to WriteLine with the format "{0} is {1}",
             and the result of asking c for its Name,
             and the result of asking c for its Value
```

**Tip**: For each loops are a simple way of looping over all of the elements of an array in C#. For example `foreach (string name in names) { … }`

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

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

Make sure that this is a **static method**. This means that it is a feature of the **MainClass** and you do not need to create a MainClass object in order to call this method, you can just call it directly on the MainClass itself.

```
public class MainClass
{
    private static void PrintCounters(Counter[] counters)
    { … }

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

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

```
Static Method: Main
--------------------
Local Variables:
 - myCounters: an array of 3 references to Counter objects
 - i: is an integer
--------------------
Steps:
 1: Assign myCounters[0] a new Counter with name "Counter 1"
 2: Assign myCounters[1] a new Counter with name "Counter 2"
 3: Assign myCounters[2], the value in myCounters[0]

 3: Loop i from 0 to 4 (using a for loop)
 4:    Tell myCounters[0] to Increment

 5: Loop i from 0 to 9 (using a for loop)
 6:    Tell myCounters[1] to Increment

 7: Tell MainClass to Print Counters, pass in counters

10: Tell myCounters[2] to Reset
11: Tell MainClass to Print Counters, pass in counters
```

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

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

12. Compile and run your program.

Download the Answer sheet and answer the following questions:

- How many Counter objects were created?
- What is the relationship between the variables and the objects?
  - What is myCounter[0] for example, and how does it relate to the object with the name "Counter 1"?
- Why does resetting myCounters[2] change the value of the counter in myCounters[0]?
- Where are objects allocated? The stack or the heap?
- How does a class' **new** method affect memory? What does it do and what does it return?
- Draw a diagram showing the locations of the variables and objects in main.

Once you are happy with your Counter Test program you can prepare it for your portfolio. This work can be placed in your portfolio as evidence of what you have learnt.

1. Review your code and ensure it is formatted correctly.
2. Save or print your answer sheet to PDF.
3. Use **Skitch** (or your preferred screenshot program) to take a screenshot of your work.
4. Remember to save the document and **backup** your work! Storing your work in multiple locations will help ensure that you do not lose anything if one of your computers fails, or you lose your USB Key.

**Note**: Each week you should aim to submit *all tasks.* **Submit** this task to **Canvas** 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 program is implemented correctly with Counters that can increment and reset.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show it outputting the correct details.
- Your explanation must show that you have understood the basic relationship between variables and objects, and the idea that objects *know* and *can do* things.