Faculty of Science, Engineering and Technology

Object Oriented Programming

Topic 1: Objects and Encapsulation

Resources

The following resources can help you with this topic:

- 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

Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the due date (check Doubtfire for the submission deadline).

Remember to submit your progress, even if you haven't finished everything.

After you have **discussed** your work with your tutor and **corrected any issues**, it will be signed off as complete.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Pass Task 1 - Hello World

The first task includes the steps needed for you to install the tools you will need in this unit. You will then use these tools to create the classic '*Hello World*' program.

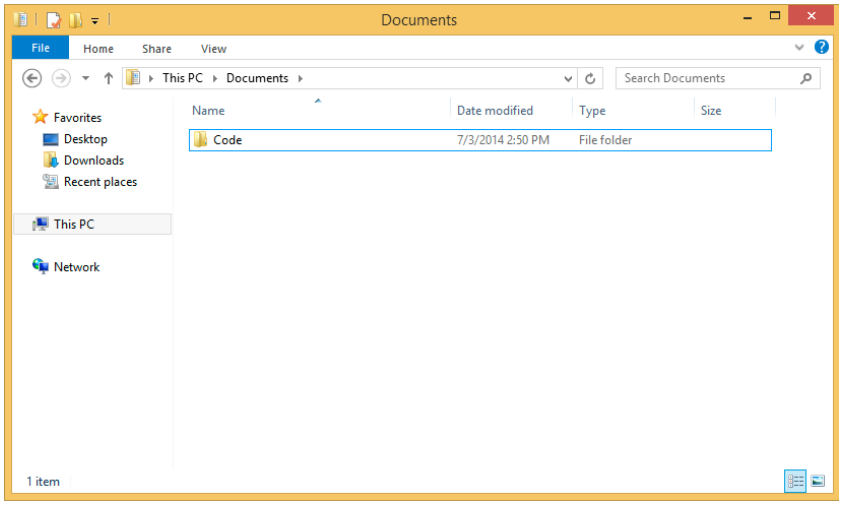
1. Install the tools you need to get started.
   - For **Linux** operating systems:
     - Install Mono MDK and GTK# via apt-get or from go-mono.com
     - Install MonoDevelop via apt-get or from monodevelop.com

> **Hint**: From the command line: `apt-get install monodevelop`

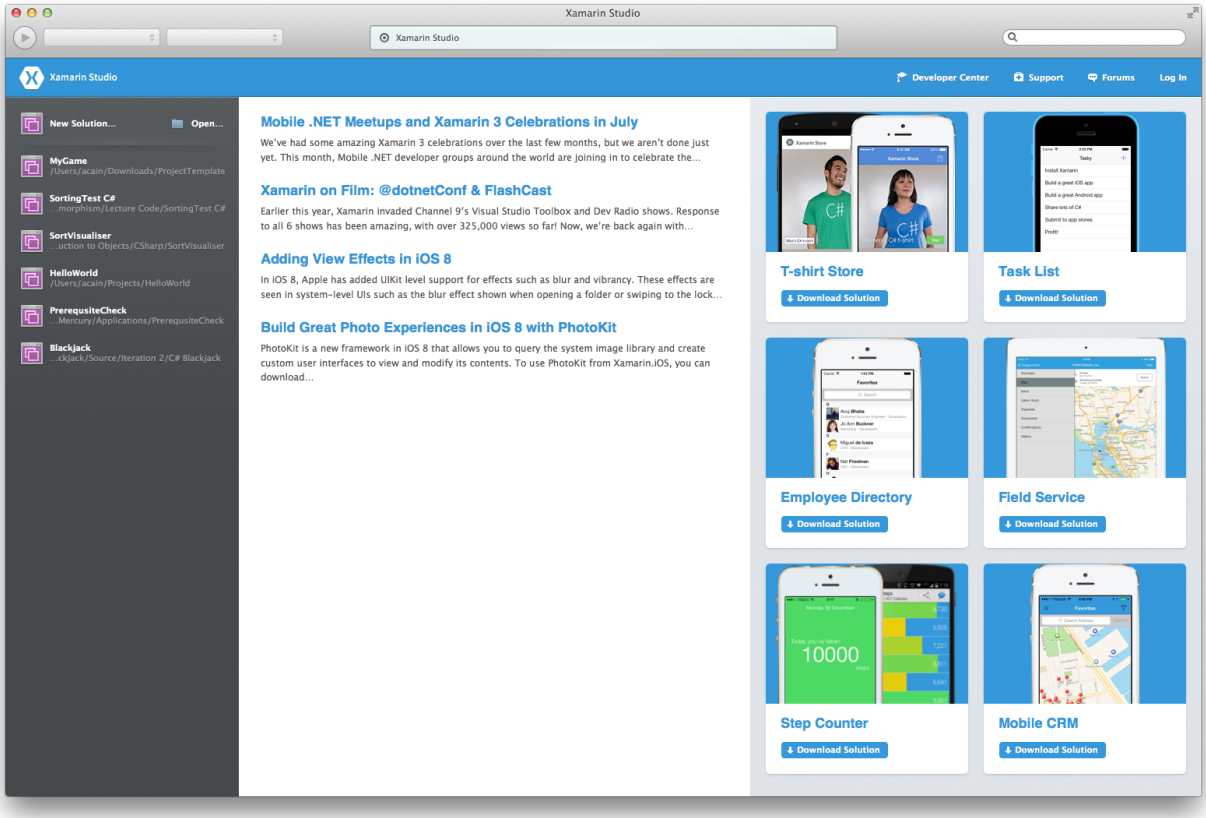
   - For **Mac** operating systems:
     - Install Mono MDK and GTK# from go-mono.com
     - Install MonoDevelop from monodevelop.com
   - For **Windows** systems:
     - Install GTK# for .NET from xamarin.com
     - Install .NET Framework 4.0 (if required) from microsoft.com
     - Install MonoDevelop from monodevelop.com

> **Note**: You can skip this step on the computers in the Swinburne lab as this will already be setup.

2. If you don't already have one, make a directory (i.e., a 'folder') to store your code (e.g., *Documents/Code/Lab1*). On a Swinburne computer you may wish to use a directory on your student drive or a USB storage device.
   - Navigate to your *Documents* directory in Finder or File Explorer
   - Right click in the *Documents* directory and select **New Folder**, name it **Code**



3. Open **Xamarin Studio** (Mono Develop)



Xamarin Studio is an **Integrated Development Environment**. It combines together the resources you need to develop programs using the C# programming language. This includes a syntax highlighting editor (like Sublime Text), with the compiler (like fpc and gcc), and a debugger. This helps make the process of building programs simpler.

Note: Why Xamarin Studio (Mono Develop) and not Visual Studio?

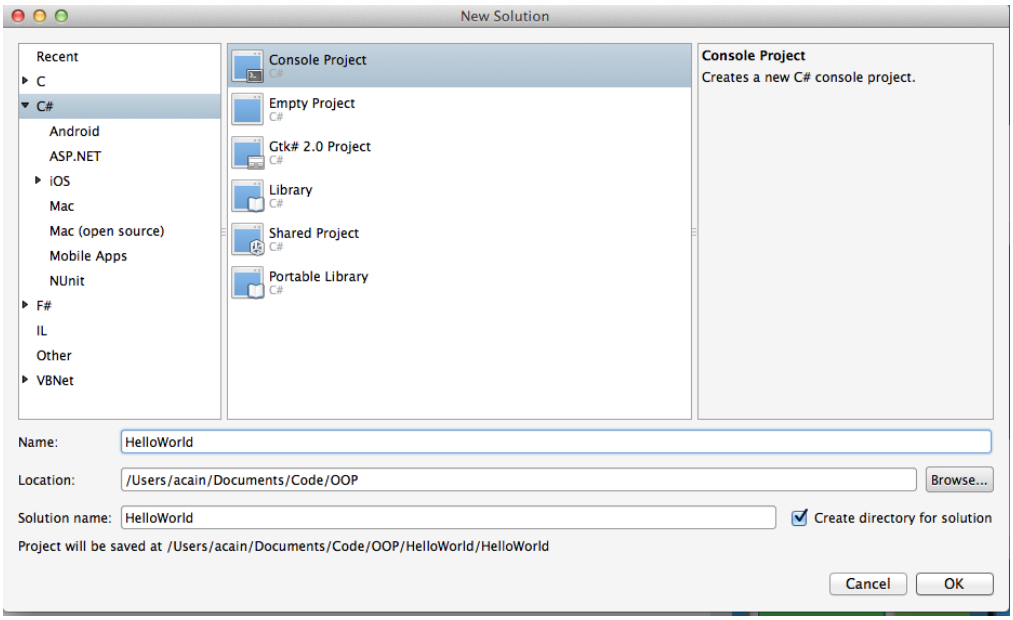
Xamarin Studio is faster, easier to use, and lighter weight than Visual Studio. Learning to use it will be easier than starting with Visual Studio, and the extra features of Visual Studio are not going to be used in this unit. Having said that, learning to use Xamarin Studio will help you understand how IDEs work in general, and make it easier for you to learn to use Visual Studio when its features are more appropriate.

As stated on the Xamarin Studio web site:

> "If Visual Studio is the Humvee of IDEs, Xamarin Studio is a Tesla."

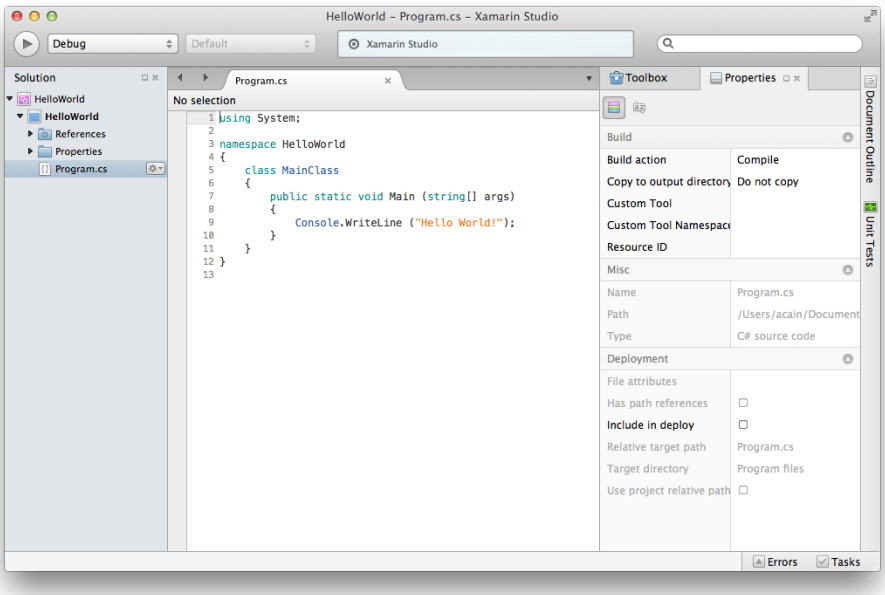
Lets start with a really simple HelloWorld to see that everything is setup correctly.

4. From the **File** menu choose **New Solution**.
   - Choose **C#** and **Console Project**
   - Enter name: **HelloWorld**
   - Choose the **Location** where you want the project saved.



Note: Xamarin Studio uses Solutions and Projects to manage the files associated with your program. A **Project** is equivalent to a **Program**. The **Solution** may contain many Projects.

5. Press **OK** to create your project. You should see the IDE change to show you the details of the solution you have created.



6. Review the IDE and get familiar with where things are:
   - You should be able to see the **Solution**, **Project** in the Solution tab to the left.
   - In the solution tab you should be able to see the **files** in the Project.
   - In the main area you should be able to see your **code**.
   - In the toolbar you should see a large **Play** button

> **Tip**: You can get some more screen space by **Auto Hiding** the **Properties** and **Toolbox** tabs on the right. You wont be using these, so best hide them away. You can also Auto Hide the **Errors**, **Tasks**, and **Application Output** if they are showing. Hover over the tops at the top to see the Auto Hide button.

7. Run the program… click the **Play** button.

> **Note**: This will run the C# compiler for you. The options for the compiler are all provided in the Project's settings. It then runs the program for you, and the code will output Hello World.

8. The program will run, but the output is likely is disappear before you can read it… Alter the code to appear as follows:

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class MainClass
6      {
7          public static void Main (string[] args)
8          {
9              Console.WriteLine ("Hello World!");
10             Console.ReadLine ();
11         }
12     }
13 }
14
```

This program is using basic structured programming concepts, so you should be able to understand how it works in general.

- **Main** is a **method** (procedure) that is the entry point for the program, so the computer begins running the instructions here when the program starts.
- The code runs in **sequence** and this demonstrates two **method calls** (like procedure calls).
  - **Console.WriteLine** writes something to the Terminal - like WriteLn or printf
  - **Console.ReadLine** reads something from the Terminal - like ReadLn or scanf

Object oriented programs work a little differently to procedural programs. An object oriented program consists of **objects** that **know** and **can do things**. When creating an object oriented program you design the kinds of objects you want, the things they will know, and the things they can do. The program then coordinates the actions of these objects by **sending** them **messages** asking them to **do things** or to return you things they know.

While this code is a "Hello World" program, it is not very "object oriented". We should be able to create an object and have it output the message for us.

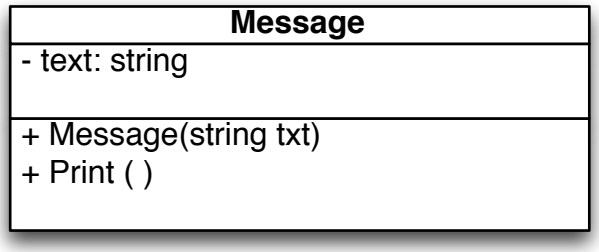
> **Note**: In the current code **Console** is an object that we are asking to WriteLine and ReadLine. Console is a class which is a special kind of object.

In C#, each object is created by a **class**. The class is a special kind of object which you can send the **new** message to, to get it to create and initialise a *new* object for you. The code within the class describes what objects created by that class looks like.

> **Tip**: You can think of a **class** as being an object blueprint. It defines the structure of objects it creates.

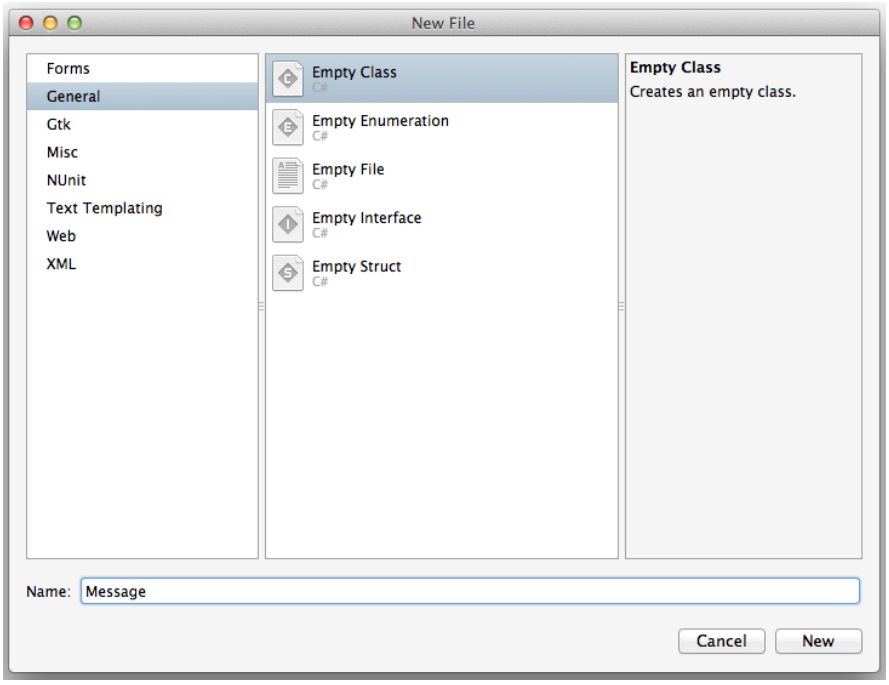
To create your own objects you first need to create a **class**, and then use that class to create an object for you.

9. Read the UML Class Diagrams Tutorial by Robert C. Martin and ensure that you fully understand the following **UML Class Diagram**. It describes a class and the features you need to implement for it.
   - The overall rectangle represents a **Message** class
   - The top part has the name of the class
   - The middle part contains the things the object *knows*. These become **data** within the object, much like the fields of a record or struct. So the message class has a **text** field that stores a reference to a **String** object.
   - The lower part contains the things the object *can do*. These become **methods** within the object, much like functions and procedures. So the Message has two methods, the first is a special **constructor** and the second is a **Print** method.

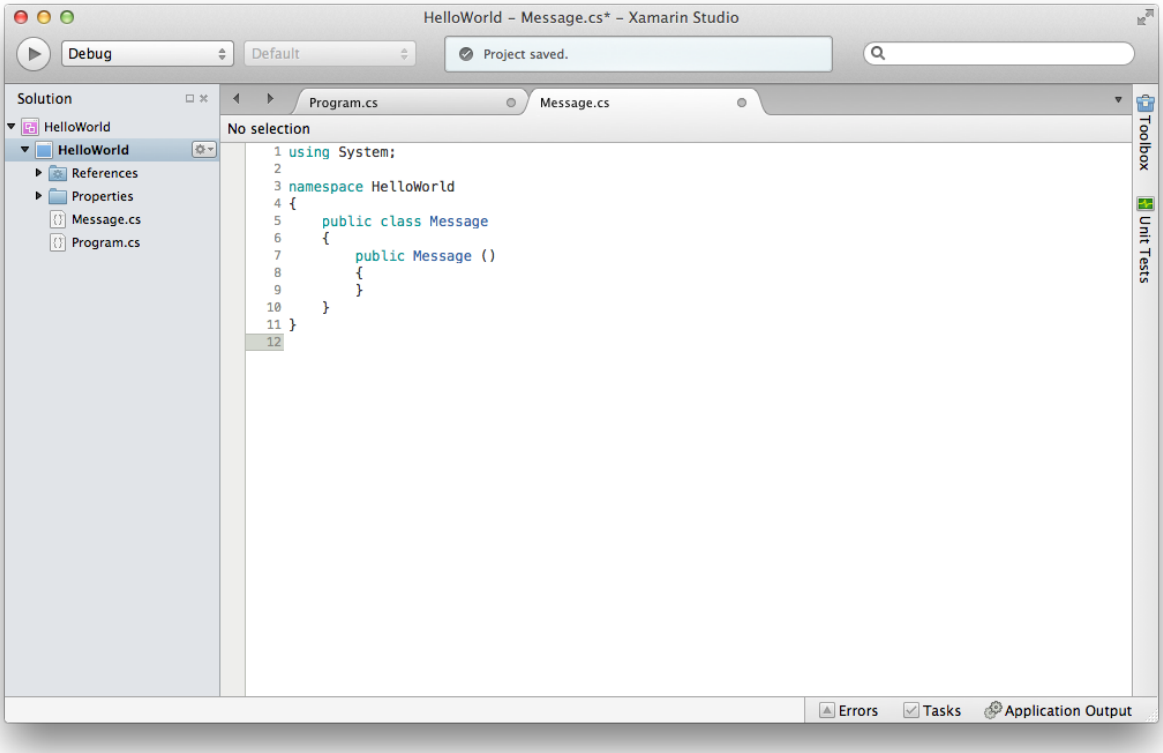


10. Create a new file for your C# class.
    - Right click the Project in the Solution tab, select New File

11. Choose an **Empty Class** and name it **Message**. Click **New** to create it.



12. You should now see a new file, and the start of the Message class' code.



Now we have the start of the class we need to add a **field** to store the **text** that the object "knows". A field is a variable declared within the class' scope - within its code.

> **Tip**: Store the things the object knows (its fields) at the top of the class. This helps match the UML, and means it is easy to locate this when you need it.

13. Add a **text** field to the **Message** class. It should appear as shown below in your code. This tells the class that objects of the Message type need to *know* a string they call "text":

```
public class Message
{
    private string text;

    public Message ()
    {
    }
}
```

> **Note**: Objects **encapsulate** the things they know and can do. You specify a scope modifier to indicate what things can see the fields and methods within a class. The **public** modifier means everyone can see it, **private** means only this class. All fields should be private.

The other code in the Message class is a special method called a **constructor**. The constructor is what **new** calls to initialise the object when it is created. The UML Diagram indicates that Message's constructor should have a string parameter. This parameter can then be used to initialise the object's text field.

14. Update the constructor to accept a **string** parameter named **txt**.
15. Assign the object's **text** field the value from the **txt** parameter. The code should appear as shown below.

```
public class Message
{
    private string text;

    public Message (string txt)
    {
        text = txt;
    }
}
```

> **Note**: Within the object's methods you can access the object's fields and other methods directly. Here **text** refers to the object's text field.

16. Now add a **Print** method to the **Message** class. It will use **Console.WriteLine** to output the object's text. The code should appear as follows:

```
public class Message
{
    private string text;

    public Message (string txt)
    {
        text = txt;
    }

    public void Print()
    {
        Console.WriteLine (text);
    }
}
```

Tip: Picture a Message **object** as a capsule that contains a **text** field and a **Print** method. When you ask it to print, the object run's the steps inside the Print method. Print is inside the capsule so it can access the object's text field.

At this point you have created the Message class. It can create objects for us that can print their messages to the Terminal.

17. Switch back to your **Program.cs** file.

Note: Notice the program is run from a **MainClass**. This is a class just like Message is. However, **Main** is a special method - a **static method**. This means that the method exists on the MainClass itself, rather than on objects created from the class. This allows C# to use this as the entry point. It asks the MainClass object to run its Main method.

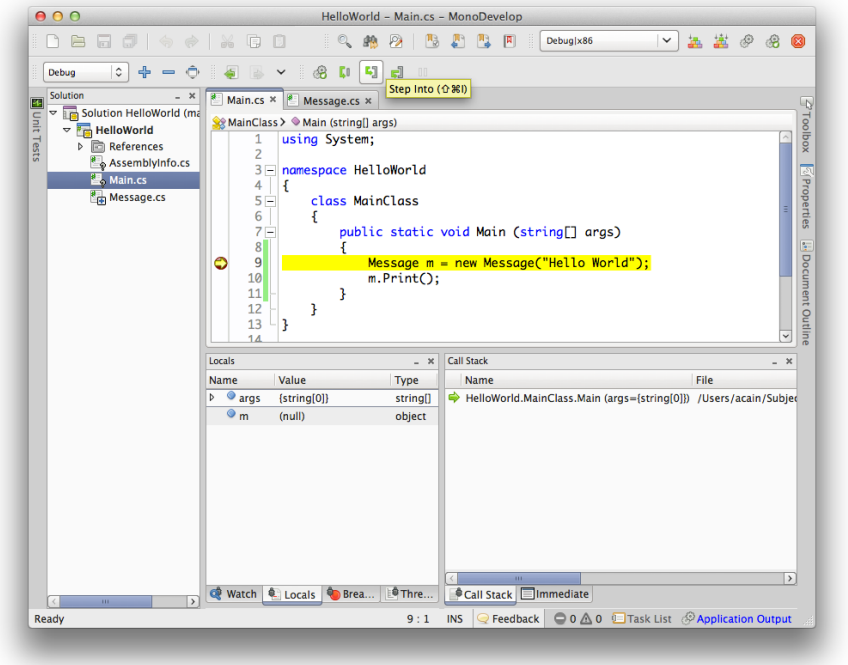
You could call Main yourself using MainClass.Main(…), this is how you can access the Console.WriteLine and Console.ReadLine methods. They are static methods of the Console class.

18. Inside the **Main** method, add a new **Message** local variable called **myMessage**.
19. Assign to **myMessage**, the result of asking **Message** for a **new** object with the text "Hello World - from Message Object".
20. Ask your myMessage object to **Print** itself out.
21. Delete the call in Main to Console.WriteLine(…). The code should appear as shown on the following page.

Note: You can create a Message object using **new Message("Hello")**.

```
class MainClass
{
    public static void Main (string[] args)
    {
        Message myMessage;
        myMessage = new Message ("Hello World - from Message Object");
        myMessage.Print ();

        Console.ReadLine ();
    }
}
```

22. Run your program…
23. Now try the following features of the debugger:
    - Add a **breakpoint**, click in the margin next to the code that creates your Message object in **MainClass**. You should see a red dot appear if you have clicked in the right location. Alternatively select the line of code and from the **Run** menu choose **Toggle Breakpoint**. A breakpoint tells the debugger to stop at this point and let you inspect the program.
    - Now run the program in the debugger using **Run > Start Debugging**. The program should stop when it gets to the breakpoint. You should be able to see the **Call Stack**, and the values of **Locals**. Watch the values of these change as the program runs. You can also hover over variables, or enter your own expressions to *Watch*.
    - Press the **Step Into** button (or choose from the **Run** menu). This will advance the program one statement at a time. You can also try stepping over and out of a method, and continuing when you no longer want to step.



You now have an object oriented "Hello World" program.

24. Extend the program to have it test user names - a Silly name testing program.
    - Create 4 message variables, and 4 different message objects.
    - Get the user to enter their name, and output one of the messages for that user. For **example** (use your own name and names of your friends, not these names):
        - "Andrew" gets the message "Welcome back oh great creator!"
        - "Fred" gets the message "What a lovely name"
        - "Wilma" gets "Great name"
        - anyone else gets "That is a silly name"

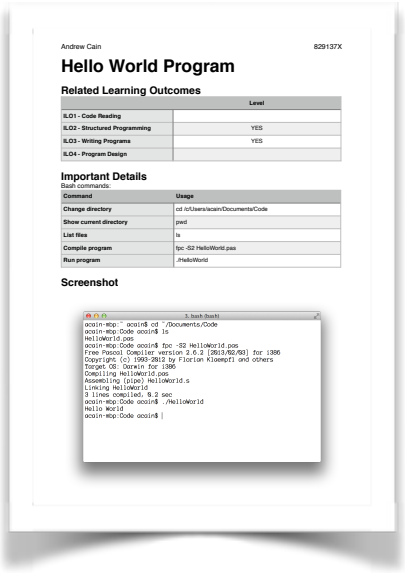
Tip: You can read a value into a string variable using Console.ReadLine(). Eg:

```
name = Console.ReadLine();
```

See the following pseudocode for the above example. Change the example to use your own names and messages.

```
Method: Main
--------------------
Local Variables:
 - myMessage: a reference to a Message object
 - messages: an array references to 4 Message objects
 - name: a reference to a String object
--------------------
Steps:
 1: Assign myMessage a new Message with text "Hello World…"
 2: Tell myMessage to Print
 3: Assign messages at index 0, a new Message with text "…"
 4: Assign messages at index 1, a new Message with text "…"
 5: …
 6: Tell Console to Write "Enter name: "
 7: Assign name, the result from asking Console to ReadLine
 8: If asking name ToLower returns "andrew" then
 9:     Tell messages[0] to Print
10: Else if asking name ToLower returns "willem" then
11:     Tell messages[1] to Print
12: …
```

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


Andrew Cain 829137X

Hello World Program

Related Learning Outcomes

	Level
ILO1 - Code Reading	
ILO2 - Structured Programming	YES
ILO3 - Writing Programs	YES
ILO4 - Program Design	

Important Details

Bash commands:

Command	Usage
Change directory	cd /c/Users/acain/Documents/Code
Show current directory	pwd
List files	ls
Compile program	fpc -S2 HelloWorld.pas
Run program	./HelloWorld

Screenshot

1. Download the Word or Pages document that will act as a header for this piece in your portfolio.
2. Add your name and student number to the file.
3. Update the Important Details section to outline the important aspects of what this piece demonstrates.
4. Use **Skitch** (or your preferred screenshot program) to take a screenshot of your work, and place it in the document.
5. Save the document and **backup** your work to multiple locations!
   - Once you get things working you **do not** want to lose them.
   - Work on your computer's storage device most of the time... but backup your work when you finish each task.
   - Use **Dropbox** or a similar online storage provider, as well as other locations.
   - A USB keys and portable hard drives are good secondary backups... but can be lost/damaged (do not rely upon them).

You now have your first portfolio piece. This will help demonstrate your learning from the unit.

Note: This is one of the tasks you need to **submit to Blackboard**. Check the assessment criteria for the important aspect your tutor will check.

Pass Task 1 - Assessment Criteria

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

- Your program prints hello world, and custom messages for at least 4 people. (funny = bonus)
- Code layout - match the example for indentation and use of case.
  - Classes and methods are PascalCase
  - Fields, variables, and parameters are camelCase (fields may be prefixed with a _ to make them easier to identify - eg: _text rather than just text)
  - Constants are UPPER_CASE
- The code must compile and the screenshot show it working on your machine.

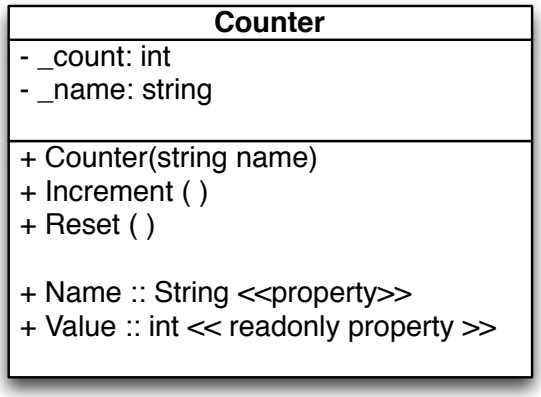
Pass Task 2 - Counter

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 start 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 by one.
- Be able to reset itself - a **reset** method that resets the count 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.



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

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.
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 adds one to the _count field - and stores the result back in the count field.
6. Add a **Reset** method that assigns 0 to the *_count* field.

At this point you can create Counter objects that will be able to remember and work with their counts, but there is no way to get the data out of the Counter. This is a part of the idea of **encapsulation**. Encapsulation means "to place within a capsule". Here you can picture your object as a capsule with aspects that are hidden inside.

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 (capsule). 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 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 not be public as others should not be able to directly change this value. 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. From the outside (outside the capsule) these *look* and *feel* like data, but inside they are actually methods. This provides a convenient way of giving other objects access to an objects data, without them actually having direct access to the fields themselves.

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.

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

Now to create the Counter class use some 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
--------------------
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.

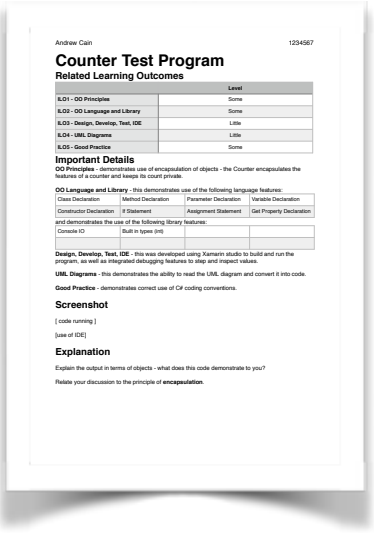
```
Static Method: Main
--------------------
Local Variables:
 - myCounters: an array of 3 references to Counter objects
 - i: is an integer
--------------------
Steps:
 1: Assign myCounters[0] a new Counter with name "Counter 1"
 2: Assign myCounters[1] a new Counter with name "Counter 2"
 3: Assign 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.
13. Explain the output of your program in terms of objects, and relate your discussion to the principle of **encapsulation**. Think about the following and include your conclusions in the discussion.
    - 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 "Counter 1".
    - Explain what happened when myCounter[2] was reset.

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.


Andrew Cain 1234567

Counter Test Program

Related Learning Outcomes

	Level
ILO1 - OO Principles	Some
ILO2 - OO Language and Library	Some
ILO3 - Design, Develop, Test, IDE	Little
ILO4 - UML Diagrams	Little
ILO5 - Good Practice	Some

Important Details

OO Principles - demonstrates use of encapsulation of objects - the Counter encapsulates the features of a counter and keeps its count private.

OO Language and Library - this demonstrates use of the following language features:

Class Declaration	Method Declaration	Parameter Declaration	Variable Declaration
Constructor Declaration	If Statement	Assignment Statement	Get Property Declaration

and demonstrates the use of the following library features:

Console IO	Built in types (int)		

Design, Develop, Test, IDE - this was developed using Xamarin studio to build and run the program, as well as integrated debugging features to step and inspect values.

UML Diagrams - this demonstrates the ability to read the UML diagram and convert it into code.

Good Practice - demonstrates correct use of C# coding conventions.

Screenshot

[code running]

[use of IDE]

Explanation

Explain the output in terms of objects - what does this code demonstrate to you?

Relate your discussion to the principle of **encapsulation**.

1. Download the Word or Pages document that will act as a header for this piece in your portfolio.
2. Add your name and student number to the file.
3. Update the Important Details section to outline the important aspects of what this piece demonstrates. You want to relate this to the unit's learning outcomes.
4. Use **Skitch** (or your preferred screenshot program) to take a screenshot of your work, and place it in the document.
5. 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 **Blackboard** once it is complete. The assessment criteria give you a list of things to check before you submit.

Pass Task 2 - 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.

Pass Task 3 - Spells

This task is the first in a small Wizard simulator program that you will create over the next few weeks. In this task you will create a Spell class and a Spell Kind enumeration.

Each Spell will.

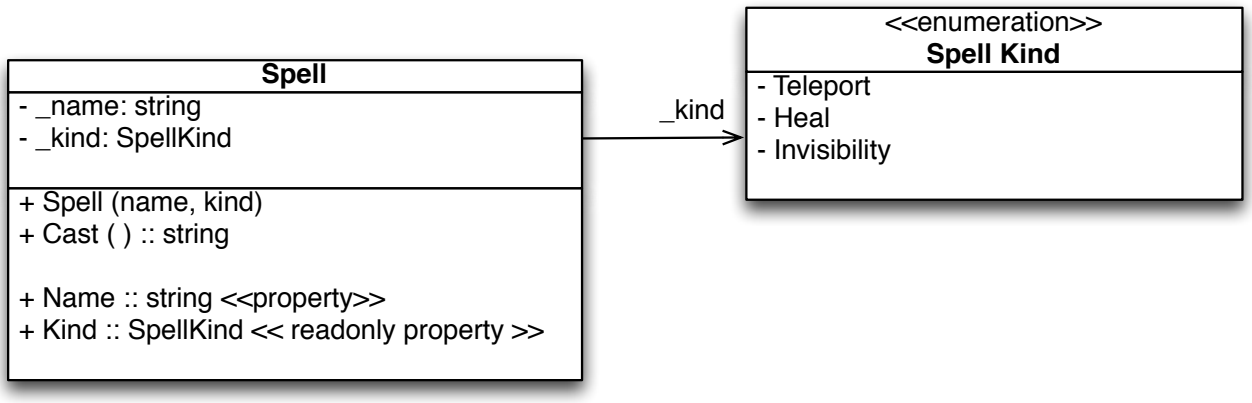
- Know what *kind* of spell it is
- Know its *name*
- Can you **constructed** with a given name and kind.
- Can be **cast**, which returns a message indicating what happened.

There are the following Spell Kinds, and they return the indicated messages:

- Teleport
  - "Poof… you appear somewhere else"
- Heal
  - "Ahhh… you feel better"
- Invisibility
  - "Zippp… where am I?"

You will create a program to test that you can create and work with Spell objects.

The following UML diagram shows the Spell class and the Spell Kind enumeration. The arrow between the two indicates that the Spell **has a** _kind it knows that is a Spell Kind. You can also see this relationship due to the presence of the _kind field.



Note: Normally a UML diagrams would just show the relationship between Spell and Spell Kind and *not* include the field. You would still need to add the field when you wrote the code, but it saves time rather than duplicating this information on the diagram.

1. Create a new **Console Project** called **Swinwarts School of Magic**
2. Add a new file, but this time choose **Empty Enumeration**. Name the file, and its enumeration, **Spell Kind**.
3. Implement the Spell Kind enumeration using the following code. This lists the valid options for this enumeration.

```
public enum SpellKind
{
    Teleport,
    Heal,
    Invisibility
}
```

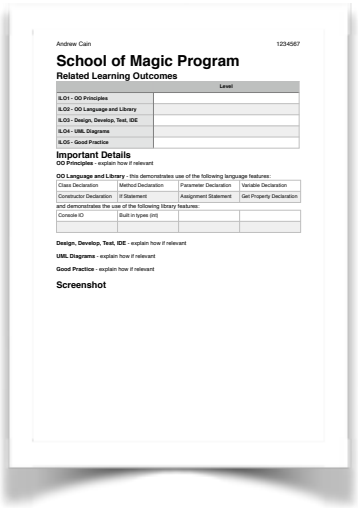
4. Add a new file for the **Spell** class.
5. Implement the **fields**, and the **constructor**.
6. Add **properties** for Name and Kind.
7. Implement the **Cast** method. It checks the Spell's kind and return the appropriate message.

At this point you should have created all of the code needed for the Spell class. So, now you need to test that it works in the MainClass.

8. Switch to the **Program.cs**
9. Add a **Cast All** static method that calls Cast on all of the elements in an array of spells that is passed to it. It should then use the **Console** class to write the name of the spell, and its effect (result of calling Cast) to the Terminal.
10. Add an array of 5 spells to **Main**.
11. Initialise these with 5 new Spell objects:
    - "Mitch's mighty mover" - Teleport
    - "Paul's potent poultice" - Heal
    - "David's dashing disappearance" - Invisibility
    - "Stan's stunning shifter" - Teleport
    - "Lachlan's lavish longevity" - Heal
12. Use **Cast All** to cast all of the spells.

Once your program is working correctly you can prepare it for your portfolio.

This time you will need to relate what you have learnt to the learning outcomes yourself. Add a few sentences for the learning outcomes you feel that you have demonstrated some aspect of in this work.



Andrew Cain 1234567

School of Magic Program

Related Learning Outcomes

	Level
ILO1 - OO Principles	
ILO2 - OO Language and Library	
ILO3 - Design, Develop, Test, IDE	
ILO4 - UML Diagrams	
ILO5 - Good Practice	

Important Details

OO Principles - explain how if relevant

OO Language and Library - this demonstrates use of the following language features:

Class Declaration	Method Declaration	Parameter Declaration	Variable Declaration
Constructor Declaration	If Statement	Assignment Statement	Get Property Declaration

and demonstrates the use of the following library features:

Console IO	Built in types (int)		

Design, Develop, Test, IDE - explain how if relevant

UML Diagrams - explain how if relevant

Good Practice - explain how if relevant

Screenshot

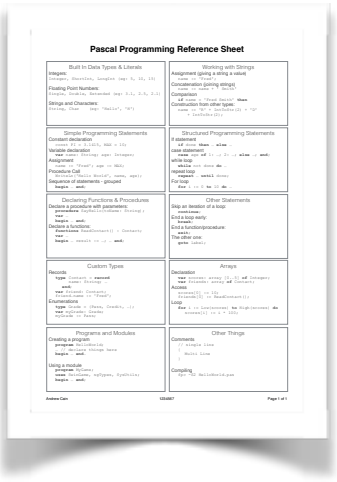
Pass Task 3 - Assessment Criteria

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

- The program is implemented correctly with Spell and Spell Kind.
- 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.

Pass Task 4 - C# Programming Reference Sheet

Object oriented languages like C# are built on top of the structured programming principles. This means that these languages share many features with languages like Pascal and C. To help you get started you need to learn the new syntax for the C# language. To do this you will need to create a **C# Programming Reference Sheet**. This will be a single page with most of the things you will need to gets started programming with C#.


Pascal Programming Reference Sheet

1. Download the Pascal Reference Sheet from Blackboard as an example to work from.
2. Download the start of the C# Reference Sheet that covers the same examples.

Make sure that your code examples follow the C# programming convention.

Once your reference sheet is completed, you can prepare it for your portfolio. Relate what you have learnt to the unit learning outcomes, and add a few sentences for each learning outcome you have demonstrated in this work.

Pass Task 4 - Assessment Criteria

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

- Your C# examples cover the same areas as the Pascal examples.
- Your C# examples use the C# coding convention