Faculty of Science, Engineering and Technology

Object Oriented Programming

Pass Task 1.1: Object Oriented Hello World

Overview

As always, "Hello World" is the first program you should write in a new language or with a new set of tools. In this tasks you will create an object oriented version of this classic program.

Purpose: Demonstrate that you have got started with Xamarin and C#.

Task: Create a hello world program and extend it to output custom messages for different user names.

Time: This task should be completed before the start of week 2.

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 Doubtfire:

- C# code files of the classes created.
- Screenshot of output.
- Screenshot of the setup of the project within Xamarin.

Instructions

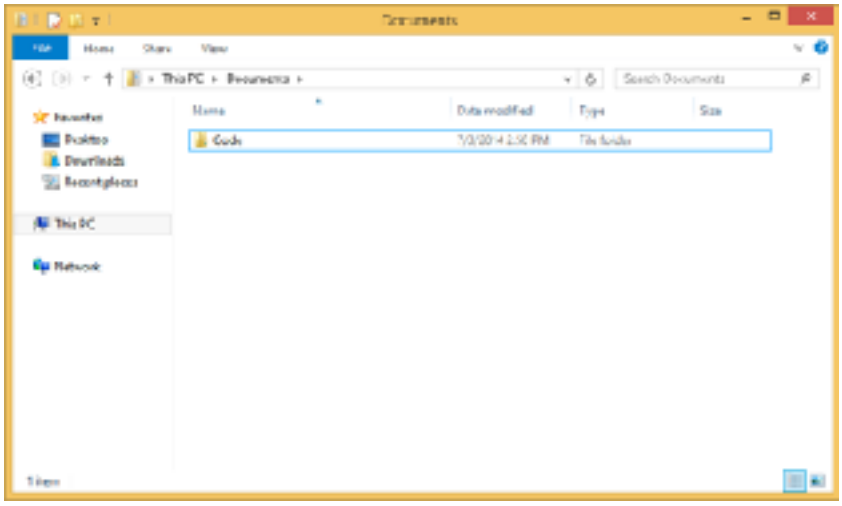
The first task includes the steps needed for you to install the tools you will need in this unit. You will then use these tools to create the classic '*Hello World*' program.

1. Install the tools you need to get started.
   - For **Mac** and **Windows** operating systems:
     - Install Visual Studio for Windows (https://www.visualstudio.com/downloads/) and Visual Studio Community for Mac (https://docs.microsoft.com/en-us/visualstudio/mac/) using the unified installer. Note that you do not need the Android or iOS packages for this unit (they take up a lot of space!).
   - 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: `sudo apt-get install fpc monodevelop monodevelop-nunit build-essentials`

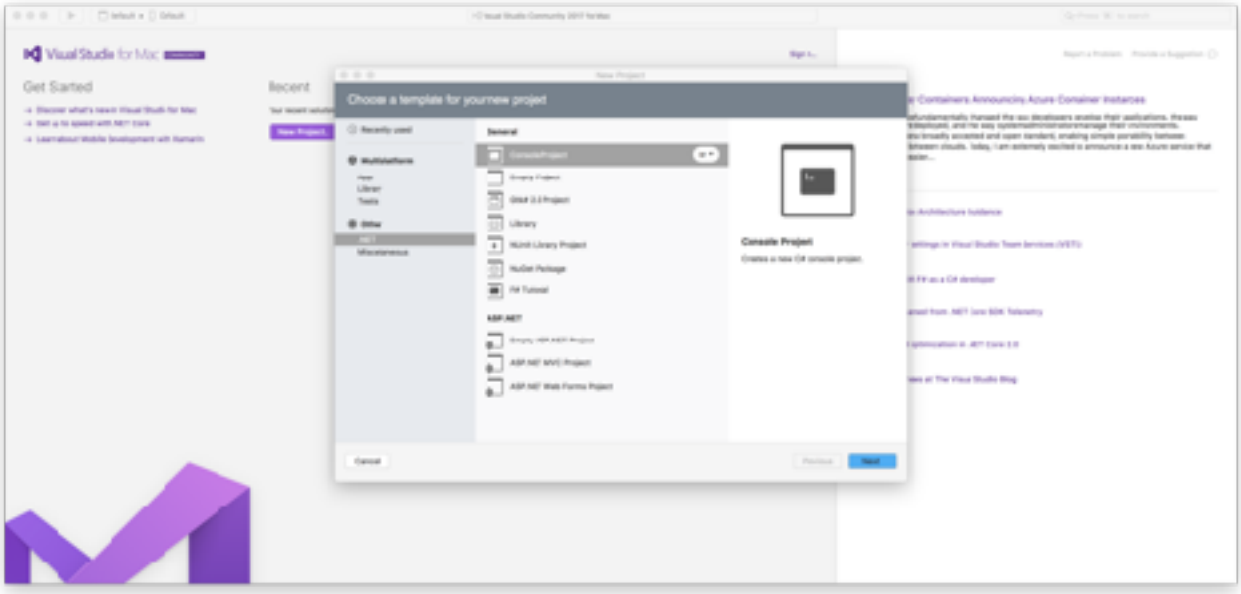
Note: If using the computers (Macs) in labs, Visual Studio Community has been installed.

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**

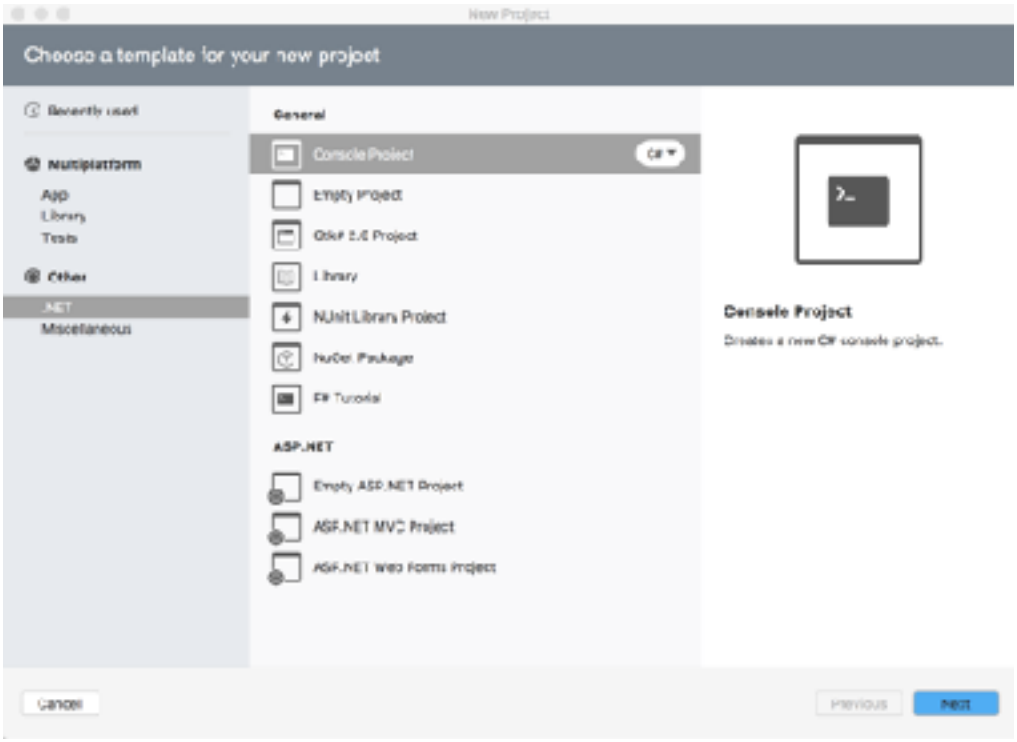


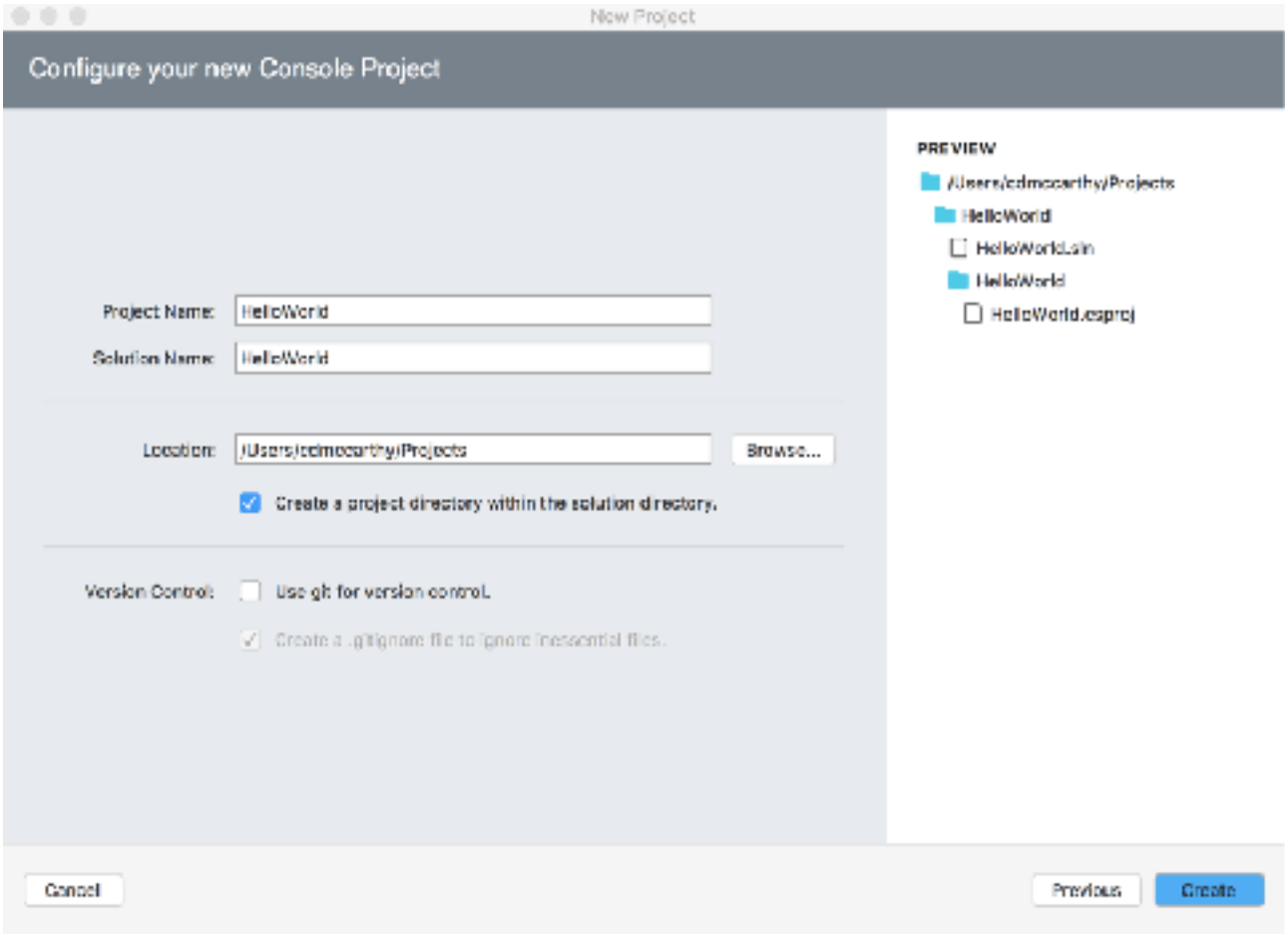
3. Open **Visual Studio**

This is an **Integrated Development Environment.** Click on New Project and you should see something like:



IDEs combine 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.



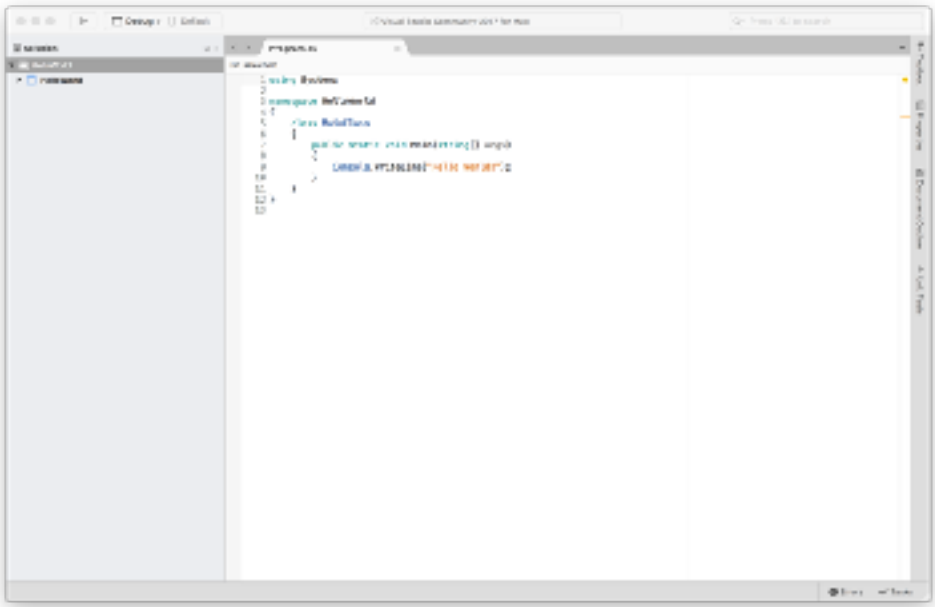


Lets start with a really simple HelloWorld to see that everything is setup correctly.

4. From the **File** menu choose **New Solution**.
   - Choose **.Net** under **Other,** and **Console Project (with C#).** Then click Next.
   - Name your project: **HelloWorld**
   - Choose the **Location** where you want the project saved. (see above image)

> **Note**: Visual Studio (and many other IDEs) 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 **Create** 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**: If not already hidden, 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 may 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.

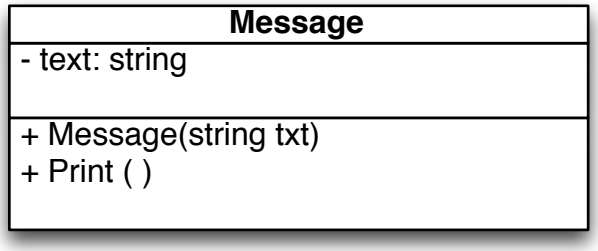
> **Note**: In the current code **Console** is an object that we are asking to WriteLine and ReadLine. Console is a class which is a special kind of object.

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.

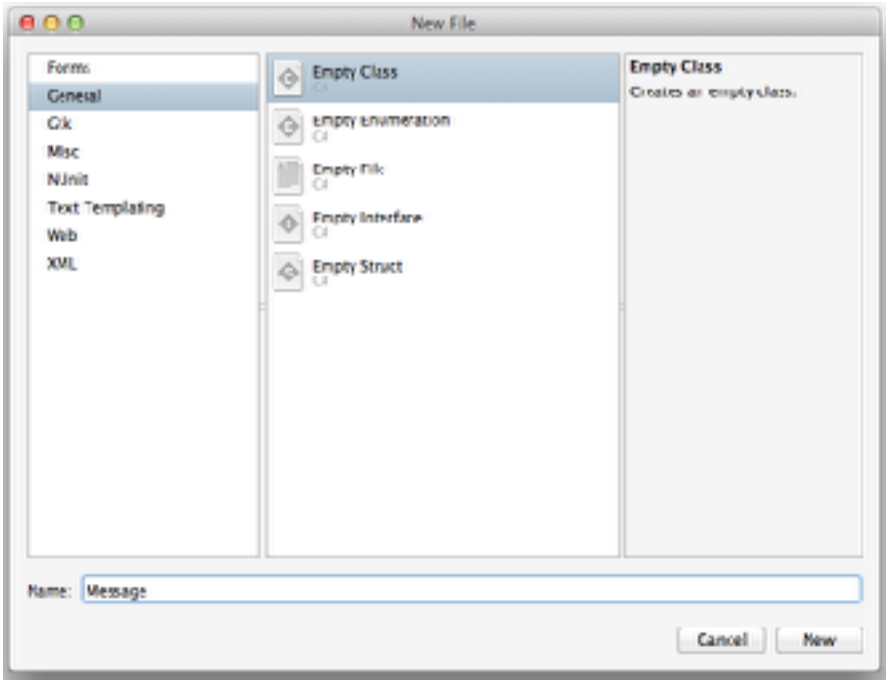
To create your own objects you first need to create a **class**, and then use that class to create an object for you.

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.

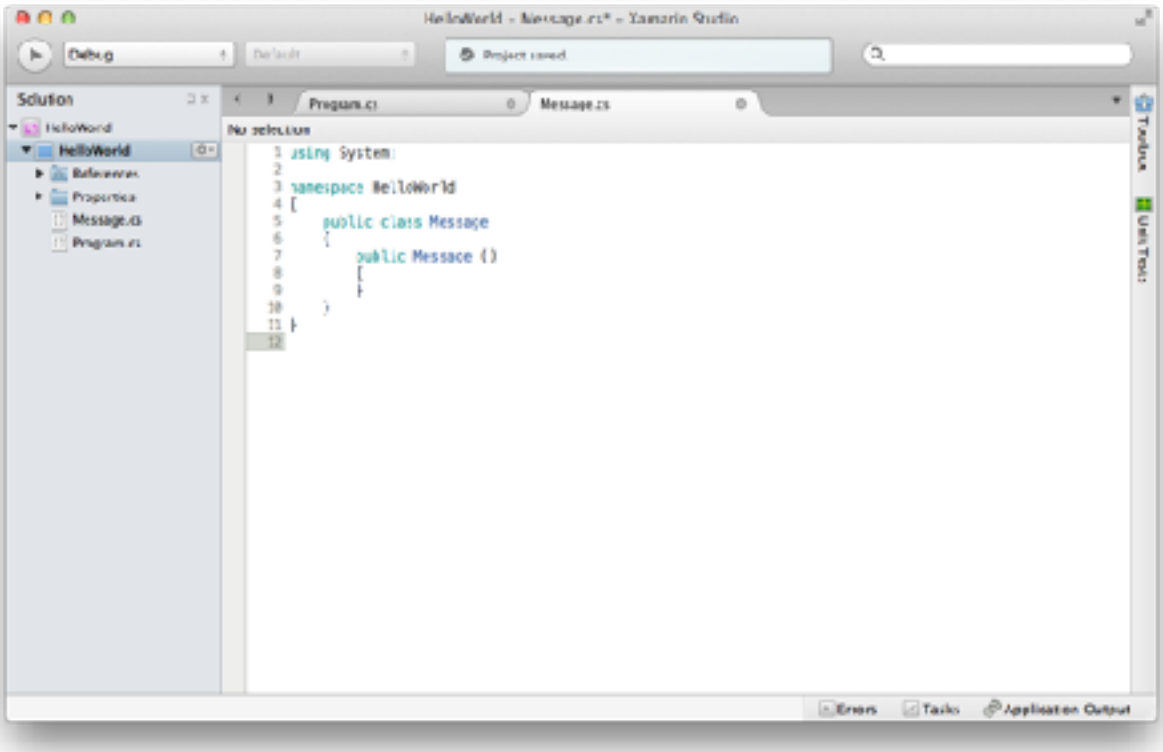


10. Create a new file for your C# class.
    - Right click the Project in the Solution tab, select New File

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 class to run its Main method.

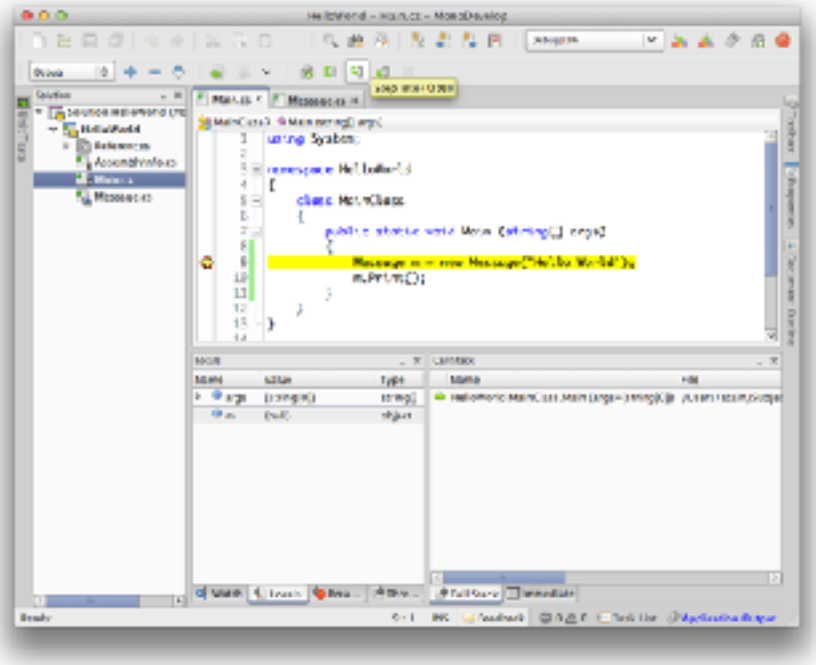
You could call Main yourself using MainClass.Main(…), this is how you can access the Console.WriteLine and Console.ReadLine methods. They are static methods of the Console class.

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):
        - "Chris" gets the message "Welcome back oh great educator!"
        - "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 "chris" then
 9:     Tell messages[0] to Print
10: Else if asking name ToLower returns "andrew" 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.

1. Review your code and ensure it is formatted correctly.
2. Run the program and use **Skitch** (or your preferred screenshot program) to take a screenshot of the Terminal showing the program's output.
3. Insert a breakpoint within your program and run the debugger. Take a screenshot of the IDE showing the call stack and the code paused within the **Print** method of the Message class.
4. Save 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.
   - USB keys and portable hard drives are good secondary backups... but can be lost/damaged (do not rely upon them).

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

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.