**Delegates**

**Introduction**

This tutorial focuses on using delegates. Delegates are special objects that can be used to encapsulate a method and allow it to be remotely invoked. Delegates are reference types and an instance of a delegate can be treated the same as any other reference type.

There are three principal steps to creating and using a delegate:

1. Declaring the delegate – This tells the program the name of the delegate, the arguments the method takes and any return type.

The following code defines a delegate called “**FunctionCallBack**” that takes an integer as an input argument and returns a double. Note it could have zero arguments and/or return nothing (in which case the **double** keyword would be replaced with **void** or no arguments would be specified).

public delegate double FunctionCallBack(int arg);

1. Create an instance of a delegate – This is where you create an instance of a delegate type. When creating an instance of a delegate you must provide as an argument the name of a function that matches the signature of the delegate.

e.g.

static double MultiplyIntByFive(int number)

{

return number\*5;

}

static void Main()

{

FunctionCallBack callBack = new FunctionCallBack(MultiplyIntByFive);

}

Note that the brackets “()” aren’t specified at the end of the method passed to the delegate.

The above example shows a static method being set as a delegate function. An instance method would be used as follows:

public class Multiply

{

public double MultiplyIntByFive(int number)

{

return number\*5;

}

}

static void Main()

{

Multiply mult = new Multiply();

FunctionCallBack callback = new FunctionCallBack(mult.MultiplyIntByFive);

}

Notice in the above example an instance of the Multiply class had to be created.

1. The final step is invoking the delegate (i.e. calling the method the delegate was passed on creation). This is simply invoked as if it’s a normal function. E.g.

static void Main()

{

Multiply mult = new Multiply();

FunctionCallBack fruitProcessor = new FunctionCallBack(mult.MultiplyIntByFive);

double res = callback(6);

}

Note that a delegate is called the same irrespective of whether the method was static or an instance type, this is as once passed to the delegate the method is anonymous. Nothing is known or needs to be known about it to invoke it.

There are three questions in this tutorial, the first two get you to experiment with declaring, creating instances of and invoking delegates. The third get you to create a class that can handle multiple delegates registered to it. This is analogous to the **Observer** pattern which is a very important pattern in computer science and event driven programming.

All question in today’s tutorial will use a winform’s project rather than a console project and each question builds on the previous.

**Questions**

**Question 1.**

In this question you will create a form with a text box, a button and a label. You will use a delegate to transfer text from the text box to the label on a click of the button. If you are happy to give this a go based on the notes and examples form the lecture carry on, otherwise follow the more detailed instruction below. Make sure you use a delegate in your solution.

**Instructions:**

Create a winform project and add a button, textbox and label, give them all appropriate names.

Create an event for the button click (this can be done by double clicking on the button), we will use this method later but for now we will add more code to the “.cs” file we’ve opened.

In the main class for your form (i.e. inside where it says public partial class YourFormName: Form) add a delegate that accepts a string as an input argument and is void return. Give it an appropriate name of you choosing.

Create a method called **UpdateLabel** that accepts a string as an argumentand is void return. The method should update the label added to the form with the given string. This method will not be static.

Add a member variable to the form of your delegate type (e.g. where it says public partial class YourFormName: Form) and default initialise it to null.

In the form constructor set this member variable to a new instance of the delegate passing in the **UpdateLabel** method.

In your button click event invoke the delegate passing in the contents of the text box.

When you press the button the contents of your textbox should be used to update the button.

**Observations:**

Note that we can reuse the same delegate again and again to update the label.

Although we were using an instance method we didn’t have to specify the object the instance method was on (as is the case in the lecture notes), this is as the delegate was inside the form class, so this was implicitly specified by the compiler (it would look inside the class first).

What’s the point? Good question! Rather than use a delegate you could have just written in your click event:

lblYourLabel.txt = txtYourTextBox.txt;

and we wouldn’t have to worry about delegates at all. This is true, you just have to trust that there will be benefits in the future!!

**Question 2.**

Add another label to the above form, give it an appropriate name. Create a delegate for it on the form class and on the press of the button get it to be updated with the contents of the textbox but appended with the word “I say “. Again you will need to create an “**UpdateLabel**” method (albeit with a different name) and must use a delegate. Follow the steps from number 1 if you are unsure what to do.

**Question 3.**

In order to implement question 2 we had to add an extra member variable to the form of type delegate and we had to add an extra delegate invocation to the button click event. This could make the form seem overly complicated and difficult to debug and maintain. In this question we will implement the observer pattern which will act as an event trigger so that much of this clutter can be reduced.

To do this you will create a local (to the form) class **Observer** that will contain a private list of the delegate objects you declared in the previous example.

The class will have a public method called **RegisterDelegate** that will accept as an argument an instance of your delegate type. This method will then add that delegate to the list.

It will also contain another public method called **Notify** that will accept as an argument a string, it will then loop through each delegate in the list, invoking it using the given string as the function argument.

The form will then contain an instance of the **Observer** class as a member variable called **OnButtonClick** (remember to new it before first use), this will replace the two member variables delegates.

In the form constructor you should register the delegates for each method to the **Observer** object using the **RegisterDelegate** method. In the button click method you can then make a single call to the **Notify** method.

**Hints:**

A list can be declared in your class as follows (note this will create an instance of a list by default without you having to explicitly create a constructor):

class Observer

{

private List<YourDelegate> delegates = new List<YourDelegate>();

}

Items can be added to a list using the .**Add** method.

You can iterate over a list as follows:

foreach(YourDelegate relevantVarName in delegates)

{

// your code here does something with relevantVarName

}

**Observations:**

If you now want to add a new component to be altered on the button click event you only have to register it to the **Observer** class, you do not have to make any other code changes (e.g. to the button click event or adding a new member variable). This is a very powerful technique and is one of the principals of event driven programming. The button event does not need to have any knowledge about what other components it is affecting on the form (or possibly outside the form).

This has number of benefits:

1. The button event function is well protected from accidental modification as it should never have to be modified. It will only ever call the **OnButtonClick.Notify()** method.
2. If a developer wants to add a component that needs to be modified by the button being pressed they can so without impacting any other methods registered to the Observer class, the developer does not even need to know what else is registered. This isolates the code making it easier to debug, test and maintain. It prevents **Regression**, where once working code is regressed to a state where it is no longer correct.