Delegates Lab

Programming in C# for Visual Studio .NET Platform II

Objective

This exercise examines delegates in C#. We've looked at and used primitive ***type***s such as `System.Int32, System.String, System.Boolean` that are pre-defined by the FCL. We've also looked at how you can declare types of your own design that encapsulate data and behaviors using classes (or structs for value types). Delegates are just a specific reference type that is tailored for referencing methods that have specific signature. Delegate types are classes derived from the `System.MulticastDelegate` type in the Framework Class Library, where the Delegate type is assigned a type name and declared in such a way that it can only work with methods that match a specific signature. For example, the declaration:

```
delegate int ProcessString(string s);
```

creates a new type named `ProcessString` that is tailored for working with any method that accepts a single string argument and returns an integer. You can think of the above syntax as shorthand for creating a class derived from `System.MulticastDelegate` whose class declaration would start like this:

```
public class ProcessString : System.MulticastDelegate
{
…
}
```

We can then declare and initialize a variable of the type `ProcessString` much like we would declare any other reference type:

```
ProcessString myDelegate;
myDelegate  = new ProcessString(MyMethod);
```

where `MyMethod` is the name of a method with the appropriate signature. The delegate inherits an Invoke method that has a signature matching the methods it is designed to encapsulate, and calls the method it references, which we can call like this:

```
int result = myDelegate.Invoke("Hello");
```

The entire point of using a delegate is to provide a level of indirection between the code that needs to use the method and the method itself, to avoid "hard-wiring" the method name into the code that needs to use the method.

In this exercise, we look first at the mechanics of creating a delegate type and using it with a simplistic example, and then proceed to look at a few different applications that illustrate the key features of delegates.

Delegate Example 0: Simplest Delegate Example

Unlike previous labs, in this lab you download a solution that I have created (https://github.com/entrotech/Lab05-Delegates), extract it to your c:\UclaExt directory and run various parts of the existing solution.

To get started with delegates, we will illustrate with a trivial example that just shows the steps involved in defining and using a delegate. Set project DelegateEx00Console to be the startup project. In Solution Explorer the start-up project is shown in bold text. To specify the start-up project, select the project's node in Solution Explorer, right-click and select "Set as StartUp Project" from the context menu. This project's Program class should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateEx00Console
{
    // Step 1: Define a delegate type with the right "shape"
    // (i.e., parameter list and return type)
    public delegate double BinaryOperation(double operand1, double operand2);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Delegate Example 0 - A degenerate usage: ");
            double a = 24;
            double b = 11;
            double c;

            // Step 2: Declare a reference variable for a
            // Binary Operation delegate
            BinaryOperation binOp;

            // Step 3: Instantiate a delegate and
            // assign to variable
            binOp = new BinaryOperation(BinaryOperations.Add);
            // binOp = BinaryOperations.Add;  // Short Form

            // Step 4: Invoke the delegate to use it.
            c = binOp.Invoke(a, b);
            //c = binOp(a, b); // Short Form

            Console.WriteLine(a + " + "
                + b + " = " + c);

            // Programmatically switch methods by
            // re-assigning the delegate to Subtract
            binOp = BinaryOperations.Subtract;

            // Delegate subtracts
            c = binOp(a, b);

            Console.WriteLine(a + " - "
                + b + " = " + c);

            Console.WriteLine("Press <Enter> to quit the program:");
            Console.ReadLine();
        }
    }

    public static class BinaryOperations
    {
        public static double Add(double op1, double op2)
```

```
        {
            return op1 + op2;
        }

        public static double Subtract(double op1, double op2)
        {
            return op1 - op2;
        }
    }
}
```

The steps involved in defining and using a delegate are:

1. Define a delegate type.
2. Declare a reference variable of your delegate type.
3. Create an instance of the delegate and assign to the delegate variable.
4. Invoke the delegate to execute the method that the delegate references.

The point of the delegate is to provide a level of indirection when invoking (calling) a method, so you can programmatically change the delegate reference variable to point to a different method.

Defining a delegate type:

```
// Step 1: Declare a delegate type named BinaryOperation
public delegate double BinaryOperation(double op1, double op2);
```

Defining a delegate defines a specialized class derived from `System.MulticastDelegate` which "delegates" an operation at runtime to a method with a specific "shape" (i.e., parameter list and return type). In defining a delegate, you give your delegate a name and specify a return type and parameter list much like you would define an abstract method in a class but add the keyword `delegate` before the return type. In the example, the `BinaryOperation` delegate type will be able to delegate an operation to any method that takes two double parameters and returns a double parameter. A delegate is a class and can be defined anywhere you would normally define a class – note how the example definition is not defined within a class at all, and could just have well have been put in a source code file of its own.

Declaring a delegate instance variable. In the Main method, the line

```
BinaryOperation binOp;
```

declares a reference variable that can hold a reference to an instance of type `BinaryOperation`.

Create an instance of the delegate that references a method. A delegate has a single "constructor" that takes one parameter which consists of the name of a static method (`BinaryOperations.Add` in the example) or an object instance variable followed by a period and an instance method name. Note that this special parameter isn't really a normal datatype, but a specialized type called a ***method group,*** which is an expression that the compiler matches

to any method of the same name. If the method is an instance method, the delegate also keeps a reference to the object instance.

```
binOp = new BinaryOperation(BinaryOperations.Add);
```

To reiterate, this code calls a "constructor" for BinaryOperation that creates a new delegate and sets it to point to the static Add method of the BinaryOperations class. This looks like a normal class constructor except that `BinaryOperations.Add` is not a normal .Net variable argument. The compiler, however will translate this into two internal bits of information: a reference to the object instance (this will be null if the method is a static method like this example), and a pointer to the Add method within the class. Alternatively, the compiler will accept an abbreviated version of this syntax like this:

```
binOp = BinaryOperation.Add;
```

since the compiler implicitly knows that an assignment to a delegate should create an instance of the delegate referencing the specified method. This abbreviated syntax is a bit more obscure, but it is more concise once you get used to it, and virtually everybody uses it, so we will as well. (The point of showing the longer form is to just emphasize that the statement is constructing a delegate instance.)

Invoke the delegate. At this point, we have an instance of a delegate and it is set up to delegate the operation to a specific method. We are finally in a position to use the delegate to call the designated method. The delegate implements an Invoke method that has the same signature and return type as the method it is defined to support. When called, the Invoke method will execute the method to which the delegate refers. In this example, calling

```
double c = binOp.Invoke(a, b);
```

will execute the delegate's method. Since the designated method is the Add method, the result will be the sum of the two parameters. Since calling the Invoke method is what we want to do with a delegate 99% of the time, the compiler also supports an abbreviated syntax for invoking a delegate of the form:

```
double c = binOp(a, b);
```

The compiler translates this into the same call to the Invoke method, so this variation just makes the source code more concise – and actually makes the call look more like a normal method call where the delegate name is substituted where we would normally specify the method name.

The short form for invoking a delegate is generally preferred and we will use it most of the time.

Take a bit of time to review this example and make sure you follow how it works, since the rest of the examples in this lab exercise build on these basic concepts. We could write this particular example without delegates in a more straightforward fashion, but the point is to illustrate how a delegate provides a level of indirection when calling methods, and the subsequent examples will illustrate less trivial situations where the indirection that a delegate provides is required.

As the code demonstrates, we can re-assign the delegate reference variable to a different method (`BinaryOperations.Subtract`), and then invoke the delegate again, and it will perform a different operation.

Delegate Example 1: Delegates as a Plug-In Method

Let's look at an example of a situation where we need to apply a different method depending on circumstances that aren't known until the code executes at runtime. What we need here is a way to "plug-in" different methods at a particular place in our code, depending upon factors that are subject to change at runtime.

Set the start –up project to DelegateEx01Console. Open the `Program.cs` source code file from this project in the code editor. The Main method looks like this:

```
static void Main(string[] args)
{
    AdjustDelegate d;

    Console.WriteLine("Delegate Example 1: Plug-In Method ");
    Console.WriteLine("Enter an adjustment percentage between -100 ");
    Console.WriteLine("and + 1000 or <Enter> for standard markup:");
    decimal percentage;
    if (decimal.TryParse(Console.ReadLine(), out percentage)
        && percentage >= -100.0M && percentage <= 1000.0M)
    {
        // To use an instance method, we need to create an instance
        // of an object that has the method.
        Adjuster adj = new Adjuster(percentage);
        // delegate operation to the object
        d = adj.Apply;
    }
    else
    {
        // To use a static method, we just use the class name to
        // reference the method.
        d = Adjuster.ApplyStandard;
    }

    while (true)
    {
        Console.WriteLine("Enter list price then "
            + "<Enter> or just <Enter> to quit:");
        decimal listPrice;
        if (decimal.TryParse(Console.ReadLine(), out listPrice))
        {
            Console.WriteLine("List Price: "
                + listPrice.ToString("c")
                + "\t Adj Price: "
                + d(listPrice).ToString("c"));
        }
        else
        {
            break;
        }
```

```
    }

    Console.WriteLine("Press <Enter> to quit the program:");
    Console.ReadLine();
}
```

For this example, the delegate definition is in a separate Class Library project, `DelegateLibrary` in the `AdjustDelegate.cs` file like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateLibrary
{
    public delegate decimal AdjustDelegate(decimal inputAmount);
}
```

The library also contains an Adjuster class that implements a few different methods we can use to adjust prices:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateLibrary
{
    public class Adjuster
    {
        decimal _percent;
        public const decimal StandardPercent = 50.0M;

        public Adjuster(decimal percent)
        {
            _percent = percent;
        }

        public decimal Apply(decimal amount)
        {
            return amount * (1.0M + _percent / 100);
        }

        public static decimal ApplyStandard(decimal amount)
        {
            return amount * (1.0M + StandardPercent / 100);
        }
    }
}
```

In this example, we prompt the user to specify what price adjustment rule to apply to a series of list prices that he will enter. The method to use is not determined until runtime based on user input.

This builds on Example 0 in a few respects:

1. The scenario is a more realistic usage of delegates, since the method is switched at runtime based on conditions that can only be determined at runtime.

2. Note that the method can be a static method as it was in Example 0, or we can create an instance (object) of an Adjuster (adjuster), and then have the delegate "delegate" the processing to an instance on a particular object.
3. I've located the delegate definition and referenced method(s) in a separate assembly, to emphasize the loose coupling between the code that uses the delegate and the code that implements the methods referenced.

Take a bit of time to review this example and make sure you follow how it works, since the rest of the examples in this lab exercise build on these basic concepts. Once again, we could write this particular example without delegates in a more straightforward fashion, but the point is to illustrate how a delegate provides a level of indirection when calling methods, and the subsequent examples will require the indirection that a delegate provides.
Run and step through the code to verify that this works as you would expect.

Delegate Example 2: Callbacks

Arguably the most common direct use of delegates is to provide a callback mechanism. A callback is a pattern where a delegate is used as a parameter when calling a method, so the called method can invoke the delegate at selected places in its processing (the called method "calls back" the calling code's method). The callback delegate will refer to a method available from the calling code that usually either performs some processing algorithm that the caller specifies or serves to inform the caller about the method's progress or status.

Set DelegateEx02Console to be the start-up project.

Then open the source code file DelegateEx2.cs in the code editor. The Main method looks like this:

```
static void Main(string[] args)
{
    Console.WriteLine("Delegate Example 2:");
    Console.WriteLine("Delegate as a Call-Back method");

    decimal[] prices = { 678.34m, 34.22m, 1293.11m };
    Adjuster adj = new Adjuster(-25.0M);
    AdjustDelegate adjustDelegate = adj.Apply;

    // Second method argument is a call-back delegate,
    // which the method "plugs into" its code at the
    // appropriate place(s).
    DecimalTransformer.Transform(prices, adjustDelegate);

    // Print adjusted prices
    foreach (var item in prices)
    {
        Console.WriteLine("Adjusted Price: "
            + item.ToString("c"));
    }

    Console.WriteLine("Press <Enter> to quit the program:");
    Console.ReadLine();
}
```

This example still uses the `AdjustDelegate` from the previous example, which is a delegate for methods that have one decimal parameter and return one decimal value. It also re-uses the

`Adjuster` class. However, it also makes use of a simple `DecimalTransformer` class with a single method that I added to the `DelegateLibrary` project:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateLibrary
{
    public class DecimalTransformer
    {
        public static void Transform(decimal[] values
            , AdjustDelegate adj)
        {
            for (int i = 0; i < values.Length; i++)
            {
                values[i] = adj(values[i]);
            }
        }
    }
}
```

If you look at the `DecimalTransformer` class, you will see that it has a static method `Transform` that takes two parameters, an array of decimals and an `AdjustDelegate` delegate. In a real application, the `DecimalTransformer` might be a class that holds general purpose utility methods for performing operations on arrays, and this class might be in a separate assembly that could be re-used by several projects. This method is quite general in nature, and can be used to apply an adjustment to each element in the array. The exact form of the adjustment can then be provided by the method caller, as long as it complies with the delegate's definition (i.e., it takes one decimal parameter and returns a decimal value).

Note the line

```
DecimalTransformer.Transform(prices, adjustDelegate);
```

in the Main method, which just uses the named delegate as a parameter. You can put a breakpoint on this line and then step through the code with the debugger, making sure to step into code (F11), so you can follow the code execution through the method call and then through the callback methods.

Once again, C# provides a more concise syntax in that we can omit the line

```
AdjustDelegate adjustDelegate = adj.Apply;
```

And modify the method call to

```
DecimalTransformer.Transform(prices, adj.Apply);
```

The compiler will work out that the second parameter should be an `AdjustDelegate` and implicitly call the delegate constructor for you. You can try this and verify that it works.

Callback delegates are pervasive in .Net programming as you will see in the following examples (and subsequent labs).

Delegate Example 3: Generic Delegates

We saw earlier how generic classes and interfaces can be used to create types that are re-usable by substituting various specific types for type parameters. You can also define generic delegates that allow you to create a family of related delegates that differ only in the type that gets substituted for the type parameter(s) when the delegate is instantiated.

For this example, set the start-up project to DelegateEx03Console. Then look at the `Transformers` class in the `DelegateLibrary` project:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateLibrary
{
    // Declare a generic delegate
    public delegate T Transformer<T>(T arg);

    public class Transformers
    {
        // Generic method operates on arrays of type
        // T, requiring Transformer delegate of same
        // type.
        public static void Transform<T>(T[] values
            , Transformer<T> trans)
        {
            for (int i = 0; i < values.Length; i++)
            {
                values[i] = trans(values[i]);
            }
        }

        public static void TransformFunc<T>(T[] values,
            Func<T, T> trans)
        {
            for (int i = 0; i < values.Length; i++)
            {
                values[i] = trans(values[i]);
            }
        }
    }
}
```

To see how this can be used, look at the `Program.cs` in the DelegateEx03Console project:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DelegateLibrary;

namespace DelegateEx03Console
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Delegate Example 3:");
            Console.WriteLine("Generic Delegate as a Call-Back method");
```

```
            double[] myDoubles = { 2, 1.41, 7.2 };
            int[] myIntegers = { 2, 3, 7 };
            decimal[] myDecimals = { 5.6M, 8.2M, -2.345M };

            // Because myDoubles is a double array,
            // compiler looks for method Square that
            // has type parameter of  double.
            Console.WriteLine("\r\nSquaring doubles with callback:");
            Transformers.Transform(myDoubles, Square);

            foreach (var item in myDoubles)
            {
                Console.WriteLine("Squared Double: "
                    + item);
            }

            // Because myIntegers is an integer array,
            // compiler looks for method Square that
            // has type parameter of int.
            Console.WriteLine("\r\nSquaring integers with callback:");
            Transformers.Transform(myIntegers, Square);

            foreach (var item in myIntegers)
            {
                Console.WriteLine("Squared Int: "
                    + item);
            }

            Console.WriteLine("\r\nUsing pre-defined Func delegate:");
            Transformers.TransformFunc(myDecimals, Square);

            foreach (var item in myDecimals)
            {
                Console.WriteLine("Squared Decimal: "
                    + item);
            }

            Console.WriteLine("Press <Enter> to quit the program:");
            Console.ReadLine();
        }

        static double Square(double d)
        {
            return d * d;
        }

        static int Square(int i)
        {
            return i * i;
        }

        static decimal Square(decimal d)
        {
            return d * d;
        }
    }
}
```

This is a variation on the previous example where we want to create a new version of the `DecimalTransformer` class which we will call `Transformers` that is more general in that it can work with arbitrary types, rather than just decimals. We can declare a generic delegate like this:

```
public delegate T Transformer<T>(T arg);
```

This is a delegate for a method that accepts a single parameter of type T and returns a result of the same type T. When the delegate is instantiated, the compiler will look at the method provided to determine what type T is and create an actual delegate for that specific type. We can then implement the `Transformers.Transform` static method as a generic method:

```
public static void Transform<T>(T[] values
    , Transformer<T> trans)
{
    for (int i = 0; i < values.Length; i++)
    {
        values[i] = trans(values[i]);
    }
}
```

So this method now takes an array of type T and then invokes a delegate of type `Transformer<T>` for each element in the array. The generic `Transform` method doesn't even care what types it is working with.

In the Main method, we can use the Transform method like this:

```
Transformers.Transform(myDoubles, Square);
```

Which the compiler will check to see that myDoubles is an array of `double`s, so it determines that T, in this case, must be `double`. Then it looks at the delegate argument and tries to find an overload of the `Square` method that is compatible with `Transformer<double>`. I just used a static `Square` method. Later, when we call

```
Transformers.Transform(myIntegers, Square);
```

The compiler will see that `myIntegers` is an `int` array, and then finds the overload of the `Square` method that works with `int`s for the delegate argument.

Pre-Defined Generic Delegates. Now that we see what generic delegates are, we might recall that a delegate type is specified solely by its return type and the number and types of its parameters.

We can then define a generic delegate where the return type and each of the parameters is parameterized. For example, a method that takes two parameters and returns a specific return type can be represented by a generic delegate like this:

```
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
```

where `TResult` is the type parameter for the return type, `T1` is the type parameter for the first parameter and `T2` is the type parameter for the second parameter. This is a really flexible set of method signatures, and could relieve us of the burden of ever creating a custom delegate definition that takes two arguments and return some return value.

In fact, in version 3.5 and later of the Framework Class Library, Microsoft added pre-defined generic delegates like this for you to use. In .Net 4.0, there are versions of `System.Func` for 0 parameters, 1 parameter, … up to 16 parameters. There is also another set of generic delegates

called System.Action which have no return type (for methods that have a void return type) with 0, 1, … 16 parameters. `Func` and `Action` delegates are used throughout the framework and can be used in your own code for many situations. This prevents a lot of needless duplication of delegates that are essentially equivalent except for name. (You may want to create a custom delegate anyway, as discussed in the Text [Griffiths] in the Section of Chapter 9 "Common Delegate Types".)

To illustrate how we can use `Func`s, I added a second method to `Transformers` called `TransformFunc` that uses the pre-defined `Func` for methods that take one parameter and return a value of the same type, then used it in the Run method like this:

```
Transformers.TransformFunc(myDecimals, Square);
```

The point of this example was to illustrate that the concept of generics applies to delegates as well as classes and interfaces by defining the delegate in terms of type parameters. At compile time, the compiler looks at code that instantiates generic delegates and determines which types should be substituted for the type parameters to create a delegate for the specified type(s), then creates the specific delegates that your program needs.

Delegate Example 4: Working with FCL methods that use pre-defined delegates

In the previous examples, we provide the delegate definition, implemented methods that comply with the delegate shape, and then invoked the delegate. Delegates are frequently used in a class library where the library may provide the delegate definition and then either implement methods that comply with the delegate or provide methods that take a callback parameter, and then the code consuming the library provides the missing "parts" of the whole delegate picture to interact with the library. In particular, the FCL uses delegates extensively.

This example revisits the `Sort` method of `List<T>`. We have already seen how to implement custom sorting of your custom types by implementing the `IComparable<T>` interface on your type to support a "default" ordering rule, and we have looked an implementing `IComparer<T>` to provide one or more alternate ordering rules. There is an overload of the `List<T>.Sort` method that takes a single delegate argument. It uses the `System.Comparison<T>` delegate from the FCL. (Think we have enough ways to implement sorting? Just wait, there will be a few more in subsequent lessons.)

First set the start-up project to DelegateEx04Console. The Program.cs file looks like this:

```
using System;
using Talent.Domain;
using Talent.Domain.TestData;

namespace DelegateEx04Console
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Delegate Example 4:");
            Console.WriteLine("Using the FCL-defined Comparison<T> delegate as");
```

```
            Console.WriteLine(
                "an argument to the List<T>.Sort(Comparison<T> comp) method");

            DomainObjectStore store = new DomainObjectStore();

            // Use override of Sort method that takes a Comparison<T> delegate.
            // The point here is that the .Net Framework has a pre-defined delegate
            // Comparison<T> so we don't define a new delegate type, but use
            // an existing one. Then the List<T> class has a
            // Sort(Comparison<T> comp) method overload that can use the
            // delegate to sort. Compare this to the Sort(IComparer c)
            // and you can see that this is a bit simpler than creating a whole
            // new class that just implements the IComparer.Compare() method.
            store.People.Sort(FirstNameComparison);

            foreach (var item in store.People)
            {
                Console.WriteLine("\t" + item);
            }

            Console.WriteLine("Press <Enter> to quit the program:");
            Console.ReadLine();
        }

        /// <summary>
        /// Sort by FirstName then LastName.
        /// </summary>
        private static int FirstNameComparison(Person p1, Person p2)
        {
            return String.Compare(p1.FirstLastName, p2.FirstLastName, true);
        }
    }
}
```

This project also references a version of the Talent.Domain and Talent.Domain.TestData projects from Lab 4.

In this example, I want to be able to sort people by FirsLasttName. I first look at the documentation of `Comparison<T>` and see that it should take two parameters of type `T` and return a negative number if the first parameter is less than the second, 0 if they are equal, or a positive number if the first parameter is greater than the second. I then created the `EmployeeNumberComparison` method that implements this for the Employee type.

Using this with a List of employees is pretty simple:

```
emps.Sort(FirstNameComparison);
```

The compiler sees that `FirstNameComparison` is a method that is compliant with `Comparison<Person>`, creates a delegate type of `Comparison<Person>`, and creates an instance of the delegate that references the `FirstNameComparison` method, then uses that delegate as the callback method for the `Sort` method. The `Sort` method then implements the QuickSort algorithm to rearrange the list, calling the `FirstNameComparison` method whenever the sort algorithm needs to compare two employees.

I find this example pretty interesting, since there is very little code when you think about all that is going on. If you want, you can compare this example to the Generics lab where we implemented custom sorting using a generic interface.

Delegate Example 5: Multicast Delegates

So far, we have used a delegate to hold a reference to a single method and then invoke that method. In .Net a single delegate instance can actually hold a reference to zero, one, or many methods. In previous examples, when assigning a delegate reference, we used the normal = operator to perform the assignment, as in DelegateEx1:

```
d = new AdjustDelegate(adjuster.Apply);
```

or, in short form:

```
d = adjuster.Apply;
```

As with other reference variable assignments, this simply sets the reference variable `d` to reference the instantiated object, and in this case that object is a delegate whose constructor references the method supplied.

However, a delegate can keep an ***invocation list*** of compatible methods, in which case it will execute them in the order they were added. (This is an oversimplification of what really happens – the book **CLR via C#, 3rd Edition** by Jeffrey Richter has a very thorough explanation of what really happens if you are interested.) To add a method to the invocation list you can use the operator +=, which is overloaded for delegate for this purpose.

To check this out, set DelegateEx05Console as the start-up project, then open the Program.cs file:

```
using System;
using Talent.Domain;
using Talent.Domain.TestData;

namespace DelegateEx05Console
{
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("Delegate Example 5:");
            Console.WriteLine("Multicast Delegates");

            DomainObjectStore store = new DomainObjectStore();

           var people = store.People;

            Action<Person> act1 = null;

            act1 += PrintName;
            act1 += PrintAge;

            Console.WriteLine("\r\nPrintName and Age:");
            act1(people[0]);

            Console.WriteLine("\r\nSame method can be invoked multiple times:");
            act1 += PrintName;
            act1 += PrintAge;
            act1(people[1]);

            Console.WriteLine("\r\nCan remove method from invocation list:");
            act1 -= PrintName;
```

```
            act1(people[0]);

            Console.WriteLine("\r\nWhen a method in the invocation list fails:");
            Action<Person> act2 = null;
            act2 += PrintName;
            act2 += BadMethod;
            act2 += PrintAge;
            try
            {
                act2(people[1]);
            }
            catch (ApplicationException)
            {
                Console.WriteLine("When exception is thrown, "
                + "delegate stops processing methods on the invocation list!");
            }

            Console.WriteLine("\r\nManually iterating through invocation list:");
            Console.WriteLine("Allows handling exceptions or retrieving return values");
            foreach (Action<Person> m in act2.GetInvocationList())
            {
                try
                {
                    m(people[1]);
                }
                catch( ApplicationException ae)
                {
                    Console.WriteLine(ae.Message);
                }
            }

            Console.WriteLine("\r\nA Delegate with an empty invocation list");
            Console.WriteLine("throws an exception if invoked.");
            act2 -= PrintName;
            act2 -= BadMethod;
            act2 -= PrintAge;
            // act2(people[0]);  // this would throw an exception, since we
            // need to check for empty invocation list
            // if it might be empty.
            if (act2 != null)
            {
                act2(people[0]);
            }

            Console.WriteLine("\r\nPress <Enter> to quit the program:");
            Console.ReadLine();
        }

        private static void PrintName(Person emp)
        {
            Console.WriteLine("Name: " + emp.LastFirstName);
        }

        private static void PrintAge(Person emp)
        {
            Console.WriteLine("Age: " + emp.Age);
        }

        private static void BadMethod(Person emp)
        {
            throw new ApplicationException("BadMethod failed!");
        }
    }
}
```

In this code we create an instance of the generic `Action<Person>` delegate named `act1`. This delegate is compatible with any method that takes a single parameter of type `Person`, and has

a void return type. I created a few static methods near the end of the class that fit this description for testing.

Then the following code uses the += operator to add two methods to the invocation list of `act1` and invokes the delegate, which iterates through the invocation list and call the `PrintName` method and then the `PrintAge` method in sequence.

```
Action<Person> act1 = null;

act1 += PrintName;
act1 += PrintAge;

Console.WriteLine("\r\nPrintName and Age:");
act1(e1);
```

This is pretty straightforward, but the subsequent code illustrates a few implications to consider:

- If you add the same method to a delegate's invocation list multiple times, it will be executed that many times.
- You can remove a method from the invocation list using the -= operator, in which case the last occurrence of the method in the invocation list will be removed.
- When an invocation list contains multiple methods they are executed sequentially and if one of the methods throws an exception, the subsequent methods will not be called. If you want to have a different behavior, you can write code to retrieve the invocation list and execute each method, handling exceptions as shown in the example. In a similar vein, if the methods have a return value, the return values from all but the last method in the invocation list will be discarded.
- The invocation list can be empty, which will throw a `NullReferenceException` if you attempt to invoke the delegate. If there is a possibility that the delegate could be null, then you should test that the delegate is not null before attempting to invoke it.

Summary

This lab exercise covered quite a lot of material that all revolved around methods and ways that we can hook up methods dynamically at runtime. Delegates are the underlying technology that allows this level of indirection when calling methods. In this lab, we looked at a few applications of delegates, the most important of which is the use of delegates as callbacks. In the next lesson, we will explore how delegates are the basis for events.

I know I found delegates and events pretty difficult to learn, but studying them is well worth your while and will help you better understand all the remaining course material.

Delegates are covered well in Chapter 9 of [Griffiths].