The Generic Action<> and Func<> Delegates

Over the course , you have seen that when you want to use delegates to enable callbacks in

your applications, you typically follow the steps shown here:

Define a custom delegate that matches the format of the method being pointed to.

Create an instance of your custom delegate, passing in a method name as a

constructor argument.

Invoke the method indirectly, via a call to Invoke() on the delegate object.

In many cases, you

simply want “some delegate” that takes a set of arguments and possibly has a return value other than

void.

In these cases, you can make use of the framework’s built-in Action<> and Func<> delegates

The generic **Action<>** delegate is defined in the System namespaces of mscorlib.dll and

System.Core.dll assemblies.

You can use this generic delegate to “point to” a method **that takes up to *16***

***arguments*** (that ought to be enough!) and **returns void.**

Now recall, because Action<> is a generic

delegate, you will need to specify the underlying types of each parameter as well.

// This is a target for the Action<> delegate.

static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)

{

// Set color of console text.

ConsoleColor previous = Console.ForegroundColor;

Console.ForegroundColor = txtColor;

for (int i = 0; i < printCount; i++)

{

Console.WriteLine(msg);

}

// Restore color.

Console.ForegroundColor = previous;

}

Now, rather than building a custom delegate manually to pass the program’s flow to the

DisplayMessage() method, we can use the out-of-the-box Action<> delegate, as so:

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\*\* Fun with Action and Func \*\*\*\*\*");

// Use the Action<> delegate to point to DisplayMessage.

Action<string, ConsoleColor, int> actionTarget =

new Action<string, ConsoleColor, int>(DisplayMessage);

actionTarget("Action Message!", ConsoleColor.Yellow, 5);

Console.ReadLine();

}

As you can see, using the Action<> delegate saves you the bother of defining a custom delegate.

However, recall that the **Action<>** delegate can point only to methods that take a **void return value.**

If you

want to point to a method that does have a return value (and don’t want to bother writing the custom

delegate yourself), you can use Func<>.

The generic **Func<> delegate** can point to methods that (like Action<>) take up to

16 parameters and

a **custom return value.**

To illustrate, add the following new method to the Program class:

// Target for the Func<> delegate.

static int Add(int x, int y)

{

return x + y;

}

Earlier in the, I had you build a custom BinaryOp delegate to “point to” addition and

subtraction methods. However, we can simplify our efforts using a version of Func<> that takes a total of

three type parameters. Be aware that the *final* type parameter of Func<> is *always* the return value of the

method. Just to solidify that point, assume the Program class also defines the following method:

static string SumToString(int x, int y)

{

return (x + y).ToString();

}

Now, our Main() method can call each of these methods, as so:

Func<int, int, int> funcTarget = new Func<int, int, int>(Add);

int result = funcTarget.Invoke(40, 40);

Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int, string>(SumToString);

string sum = funcTarget2(90, 300);

Console.WriteLine(sum);

So, given that Action<> and Func<> can save you the step of manually defining a custom delegate,

you might be wondering if you should use them all the time. The answer, like so many aspects of

programming is “it depends.” In many cases, Action<> and Func<> will be the preferred course of action

(no pun intended). However, if you need a delegate that has a custom name that you feel helps better

capture your problem domain, building a custom delegate is as simple as single code statement