Published Apr 07, 2017 | Alfredo Ball

Understanding C# Delegates The Easy Way : Part 2 (Anonymous Methods)

Now that we have the understanding that a delegate is just a method signature definition, let's move on to an easier concept.

Remember, this is not complicated at all, it's super easy.

Instead of having to define a formally named method that can match a delegate signature, we can use a short hand syntax called an anonymous method.

The syntax format is called a lambda expression (λ). As far as I know a lambda expression is just a short hand way of detailing a function/method, such as its result and input arguments with both sides separated by a => symbol. All this means is that it doesn't have a name and the argument/parameter types don't need to be specified since they are already defined by the delegate definition. The C# compiler can infer the types.

So let's make a delegate definition/declaration like so:

```
delegate int MyDelegate(int someNumber, int someOtherNumber)
```

MyDelegate is the method signature definition that some other method has to match. But instead of using a formally defined method, we can use an anonymous method called a lambda expression.

A lambda expression allows us to take in the corresponding input arguments defined by a delegate but we only have to specify the input argument names (the names are up to us, they are placed positionally), not the types like in a normal method declaration.

A regular method that can be used to match MyDelegate would look like this:

```
public int MyMethod(int firstParameter, int secondParameter)
{


    return
}
```

**Note that the parameter names of the method don't have to be the same as the paramter names of the delegate definition/declaration, they just have to be in the same order.

However, an anonymous method only has to accept the same number of input arguments as named placeholders, so if a class has a property of type MyDelegate, we can assign an anonymous method that may add both input arguments together like this:

```
public MyDelegate MyDelegateProperty = (x, y) => x + y;
```

What this means is that the someNumber and someOtherNumber parameter input arguments of the MyDelegate declaration that could be satisfied by the firstParameter and secondParameter arguments of the MyMethod method reference are instead being substituted by the x and y symbols of an anonymous method.

X and Y are good enough, we don't need to specify the types of the arguments because the C# compiler is smart enough to know that X is type int and Y is type int, so it really doesn't need us to explicitly specify the type since the delegate definition already tells it what it should be.

The first part of the lambda expression is the input argument/parameter names, the right hand side of the => symbol is the actual code logic which can make use of the input argument/parameter names.

The most important lesson is that a delegate type can be satisfied by a formally defined method that corresponds to the same signature of a delgate, or we can also use an anonymous (throw away) method.

Whether one should use a formally defined method or an anonymous method depends primarily on reusability. If you reuse a formally defined method throughout an app, then by all means go that route, but if you need a quick method for a one time use, then go the anonymous route.

A delegate is used as a type, so it can be a type for a property or an input argument to a method, so a lambda expression can be used in either of these scenarios.

Viewed using Just Read