Understanding Delegates

By Bill Wagner
January 2011

Delegates, Lambdas and Conversions, oh my

One of the most common questions I get from developers is why code like the following does not compile (taken from a WPF sample):

```
void stopwatch_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
{
   string newContent;
   TimeSpan span = finished - DateTime.Now;
   if (span.TotalSeconds() > 0)
      newContent = span.ToString();
   else
      newContent = "Finished";

   if (System.Threading.Thread.CurrentThread != label1.Dispatcher.Thread)
      label1.Dispatcher.Invoke(() => label1.Content = newContent);
   else
      label1.Content = newContent;
}
```

The highlighted line generated CS1660, "Could not convert lambda expression type to type System.Delegate because it is not a delegate type." Developers are often confused about why the compiler cannot convert a lambda expression to a delegate. After all, the compiler converts lambdas to delegates in every LINQ query expression, in APIs used with the Task Parallel Library, and many other locations throughout the framework. In this article, I'll explain why this doesn't work, what you can do about it, and why it works in many other libraries.

.NET and C# are 10 years old, but continue to get new features, new libraries and new ways to express our designs. That gives us challenges when we work with APIs that were designed for earlier versions of the C# language. The API used above was designed when C# did not have support for lambda expressions, or anonymous functions. Some APIs you work with predate support for generics. Anonymous functions can be hard to use with older APIs whose parameters are the abstract delegate type. The most common location for this error is WPF and Windows Forms event handlers, as shown above. Remember that Windows controls can be updated only from the UI thread, not from a background thread. In WPF, you use a control's Dispatcher property to determine if code is executing on the UI thread or not. If not, you must use the Invoke() method on the dispatcher object to marshal that call to the UI thread.

The Invoke method's signature looks like this:

```
public Object Invoke(
        Delegate method,
        params Object[] args
)
```

The important key for this article is that the first parameter's type is System.Delegate. That means it could be any valid delegate type. We want to write is code like the routine shown above. CS1660 occurs whenever you use an anonymous function block or a lambda expression with an API whose parameter is System.Delegate. You get the same error if you use this assignment:

```
Delegate foo = () => label1.Content = newContent;
```

Let's examine exactly why the compiler doesn't do what you want. There are two ways to create a delegate in C#: a delegate creation expression, or a conversion to a delegate type. The above assignment does not qualify as either.

A delegate creation expression involves the new operator. You new up some type that represents a concrete delegate type. You can create an Action, because it is a concrete type:

```
Delegate foo = new Action(() => label1.Content = newContent);
```

You cannot new up a System.Delegate, because it is an abstract type:

```
Delegate foo = new Delegate(() => label1.Content = newContent); // fails
```

You can use a delegate creation expression to fix the original problem that caused CS1660:

```
label1.Dispatcher.Invoke(new Action(() => label1.Content = newContent));
```

I prefer using delegate conversions rather than delegate creation expressions in most cases. Let's examine why the original code is not a valid conversion to a delegate type and how to fix that. After all, that lambda expression is a valid Action. This is a valid delegate conversion expression:

```
Action updateLabel = () => label1.Content = newContent;
```

You may have already guessed that the following delegate conversion expression fails:

```
System.Delegate updateLabel = () => label1.Content = newContent; // fails
```

An anonymous function block (or lambda expression) does not have a type, or a value. It is convertible to a compatible delegate type, or expression tree. In other words, "() => label1.Content = newContent" is not an Action, it is an untyped lambda expression. However, it is convertible to an Action. Section 6.5.1 of the C# spec covers the rules that make an anonymous function convertible to a delegate type. Section 15.2 covers what makes a method compatible with a particular delegate. It's also why an implicitly typed local variable (var) can't be used to declare a variable that is initialized to a lambda expression. This declaration also causes CS1660:

```
var updateLabel = () => label1.Content = newContent;
```

Because a lambda expression does not have a type, the compiler cannot infer the type of the local variable “updateLabel”. If you cast the lambda expression to an Action, you can use it with Dispatcher.Invoke, just like declaring the variable’s type explicitly did above:

```
label1.Dispatcher.Invoke((Action)(() => label1.Content = newContent));
```

There are two different conversions going on in that method call. First, you’ve got an explicit cast from a lambda expression to an Action. Second, there is an implicit conversion from an Action to a System.Delegate.

The lesson here is that you must explicitly convert the lambda expression (which does not have a type) to a concrete delegate type before you use that delegate as a parameter to a method call expecting a System.Delegate object. The second conversion, from a concrete delegate type to a System.Delegate, is implicit, so will succeed.

Why doesn’t this affect all APIs that use delegates?

You may already be wondering why you do not need to explicitly state a delegate type on other methods that take delegate types as parameters. LINQ APIs don’t need explicit delegate types, neither do the Task Parallel Library APIs. That’s because those APIs use concrete delegate types in their API signatures. The API signature for System.Linq.Enumerable.Where is:

```
public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, bool> predicate)
```

Func<TSource, bool> is a concrete delegate type. When you call a Where() method call using a lambda expression or an anonymous function block, the compiler tries to convert that expression (which does not have a type in and of itself) into a Func<TSource, bool>. As long as the anonymous function (or lambda expression) can be converted to Func<TSource, bool>, the code compiles.

APIs created with C# 3.0 generally follow the practice of using concrete delegate types for parameters, whereas earlier APIs will use parameters typed as System.Delegate. That’s the difference between the newer APIs and those APIs where you must use an explicit conversation. The .NET 3.5 framework contains several Action, Func, and Predicate types that make it easy to define a more strongly typed API. Earlier APIs used System.Delegate, along with a params object array so that you could use any method signature you needed. In a .NET world before generics, that made sense. In that world, you could not create anonymous functions or lambda expression, so you didn’t notice as much. Now, the language and the .NET library support a concise syntax to create the many overloads you might need for common method signatures.

Inferring types is not easy, and sometimes impossible

I’ve discussed how anonymous function expressions do not have types, and System.Delegate parameters can be satisfied by any concrete delegate type. While we’d like to have the compiler infer exactly what we meant, that’s actually much harder than it appears. After all, you know exactly what you want when you write a lambda expression in your code. Put yourself in the compiler’s place, and it looks much different. Suppose a developer writes this:

```
var add = (x, y) => x + y;
```

This should probably be a Func<T1, T2, TResult>, but what types should be used for T1, T2, and TResult? Any numeric type could be valid for T1 and T2. Addition is even defined between dissimilar numeric types, which increase the valid possible signatures. String is a possibility for T1 and T2. Any other type in scope for which a valid operator + is defined could also be a possibility. Other dissimilar types, like DateTime and TimeSpan would work. The compiler really does not know what the developer meant for the concrete type. That ambiguity is why a declaration like this is not legal C#. You cannot assign a lambda expression or an anonymous function declaration to an implicitly typed variable. (The Visual Basic .NET compiler and the F# compiler have greater support for type inference of delegates).

What surprises a lot of people is that even this generates the same error:

```
var add = (int x, int y) => (int)(x + y);
```

The thinking usually goes: I've defined an explicit type for the parameters, and the return type. Obviously, I mean Func<int, int, int>. Why can't the compiler figure that out?

The problem is that while you could mean Func<int, int, int>, that's not the only definition you could mean. You could define a different delegate type that has the same structure:

```
delegate int Sum(int, int);
```

Delegate types in C# are name equivalent, not structurally equivalent. A Sum is a different type than a Func<int, int, int>. The developer might have meant either or some other delegate type that is also structurally equivalent.

The only way you enable the compiler to convert an anonymous function into a delegate type is to explicitly state what concrete delegate type you want. You have two ways to do that: use a delegate creation expression, or assign the lambda to a concrete delegate type so that your code contains a valid conversion to a delegate type. You usually do not, however, need to define a new delegate type. You should choose one of the variations of System.Action (for methods that do not return a value), Predicate (for methods that test a condition and return a Boolean), or System.Func (for methods that return a single value).

© 2014 Microsoft