Username: Pralay Patoria **Book:** C# in Depth, Third Edition: Foreword by Eric Lippert. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.3. Covariance and contravariance

We've already talked a lot about the concepts of covariance and contravariance in different contexts, usually bemoaning their absence, but delegate construction is the one area in which they're available in C# prior to version 4. If you want to refresh yourself about the meaning of the terms at a relatively detailed level, refer back to section 2.2.2, but the gist of the topic with respect to delegates is that if it would be valid (in a static typing sense) to call a method and use its return value everywhere that you could invoke an instance of a particular delegate type and use *its* return value, then that method can be used to create an instance of that delegate type. That's wordy—it's a lot simpler with examples.

Different types of variance in different versions

You may already be aware that C# 4 offers *generic* covariance and contravariance for delegates and interfaces. This is entirely different from the variance we're looking at here—we're only dealing with creating *new* instances of delegates at the moment. The generic variance in C# 4 uses *reference conversions*, which don't create new objects—they just view the existing object as a different type.

We'll look at contravariance first, and then covariance.

5.3.1. Contravariance for delegate parameters

Let's consider the event handlers in the little Windows Forms application in listing 5.1. The signatures of the three delegate types are as follows:[2]

[2] I've removed the *public delegate* part for reasons of space.

```
void EventHandler(object sender, EventArgs e)

void KeyPressEventHandler(object sender, KeyPressEventArgs e)

void MouseEventHandler(object sender, MouseEventArgs e)
```

Consider that `KeyPressEventArgs` and `MouseEventArgs` both derive from `EventArgs` (as do a lot of other types—MSDN lists 403 types that derive directly from `EventArgs` in .NET 4). If you have a method with an `EventArgs` parameter, you could always call it with a `KeyPressEventArgs` argument instead. It therefore makes sense to be able to use a method with the same signature as `EventHandler` to create an instance of `KeyPressEventHandler`, and that's exactly what C# 2 does. This is an example of contravariance of parameter types.

To see that in action, think back to listing 5.1 and suppose that you don't need to know which event was firing—you just want to write out the fact that an event has happened. Using method group conversions and contravariance, the code becomes a lot simpler, as shown in the following listing.

Listing 5.2. Demonstration of method group conversions and delegate contravariance

```
static void LogPlainEvent(object sender, EventArgs e)
{
   Console.WriteLine("An event occurred");
}
...
Button button = new Button();
button.Text = "Click me";
button.Click += LogPlainEvent;

button.KeyPress += LogPlainEvent;
button.MouseClick += LogPlainEvent;

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

1 Handles all events

2 Uses method group conversion

3 Uses conversion and contravariance

The two handler methods that dealt specifically with key and mouse events have been completely removed, and you're now using one event handling method for everything ❶. Of course, this isn't terribly useful if you want to do different things for different types of events, but sometimes all you need to know is that an event occurred and, potentially, the source of the event. The subscription to the `Click` event ❷ only uses the implicit conversion we discussed in the previous section because it has a simple `EventArgs` parameter, but the other event subscriptions ❸ involve the conversion and contravariance due to their different parameter types.

I mentioned earlier that the .NET 1.0/1.1 event handler convention didn't make much sense when it was first introduced. This example shows exactly why the guidelines are more useful with C# 2. The convention dictates that event handlers should have a signature with two parameters, the first of which is of type `object` and is the origin of the event, and the second of which carries any extra information about the event in a type deriving from `EventArgs`. Before contravariance became available, this wasn't useful—there was no benefit to making the informational parameter derive from `EventArgs`, and sometimes there wasn't much use for the origin of the event. It was often more sensible to pass the relevant information directly in the form of normal parameters with appropriate types, just like any other method. Now you can use a method with the `EventHandler` signature as the action for *any* delegate type that honors the convention.

So far we've looked at the values entering a method or delegate—what about the value coming out?

5.3.2. Covariance of delegate return types

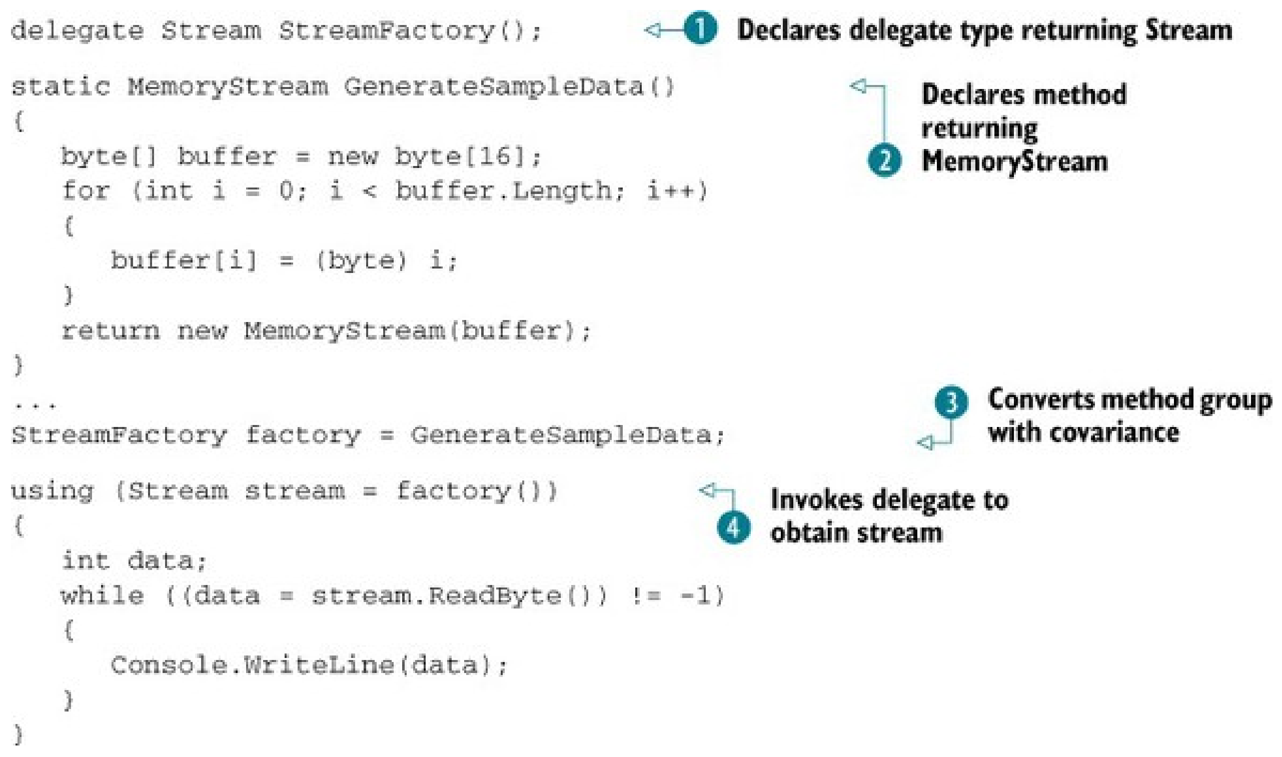
Demonstrating covariance is harder, as relatively few of the delegates available in .NET 2.0 are declared with a nonvoid return type, and those that are tend to return value types. There are some available, but it's easier to declare your own delegate type that uses `Stream` as its return type. For simplicity, we'll make it parameterless:[3]

[3] Return type covariance and parameter type contravariance can be used at the same time, but you're unlikely to come across situations where that would be useful.

```
delegate Stream StreamFactory();
```

You can now use this with a method that's declared to return a specific type of stream, as shown in the following listing. You declare a method that always returns a `MemoryStream` with some sequential data (bytes 0, 1, 2, and so on up to 15), and then use that method as the action for a `StreamFactory` delegate instance.

Listing 5.3. Demonstration of covariance of return types for delegates



```
delegate Stream StreamFactory();

static MemoryStream GenerateSampleData()
{
   byte[] buffer = new byte[16];
   for (int i = 0; i < buffer.Length; i++)
   {
      buffer[i] = (byte) i;
   }
   return new MemoryStream(buffer);
}
...
StreamFactory factory = GenerateSampleData;

using (Stream stream = factory())
{
   int data;
   while ((data = stream.ReadByte()) != -1)
   {
      Console.WriteLine(data);
   }
}
```

1 Declares delegate type returning Stream
2 Declares method returning MemoryStream
3 Converts method group with covariance
4 Invokes delegate to obtain stream

The generation and display of the data in listing 5.3 is only present to give the code something to do. The important points are the annotated lines. You declare that the delegate type has a return type of `Stream` ❶, but the `GenerateSampleData` method has a return type of `MemoryStream` ②. The line creating the delegate instance ③ performs the conversion you saw earlier and uses covariance of return types to allow `GenerateSampleData` to be used as the action for `StreamFactory`. By the time you invoke the delegate instance ④, the compiler no longer knows that a `MemoryStream` will be returned—if you changed the type of the `stream` variable to `MemoryStream`, you'd get a compilation error.

Covariance and contravariance can also be used to construct one delegate instance from another. For instance, consider these two lines of code (which assume an appropriate `HandleEvent` method):

```
EventHandler general = new EventHandler(HandleEvent);

KeyPressEventHandler key = new KeyPressEventHandler(general);
```

The first line is valid in C# 1, but the second isn't—in order to construct one delegate from another in C# 1, the signatures of the two delegate types involved have to match. For instance, you could create a `MethodInvoker` from a `ThreadStart`, but you couldn't create a `KeyPressEventHandler` from an `EventHandler` as shown in the second line. You're using contravariance to create a new delegate instance from an existing one with a *compatible* delegate type signature, where compatibility is defined in a less restrictive manner in C# 2 than in C# 1.

All of this is positive, except for one small fly in the ointment.

5.3.3. A small risk of incompatibility

This new flexibility in C# 2 creates one of the few cases where existing valid C# 1 code may produce different results when compiled under C# 2. Suppose a derived class overloads a method declared in its base class, and you try to create an instance of a delegate using a method group conversion. A conversion that previously only matched the base class method could match the derived class method due to covariance or contravariance in C# 2, in which case that derived class method would be chosen by the compiler. The following listing gives an example of this.

Listing 5.4. Demonstration of breaking change between C# 1 and C# 2

```
delegate void SampleDelegate(string x);


public void CandidateAction(string x)

{

   Console.WriteLine("Snippet.CandidateAction");

}


public class Derived : Snippet

{

   public void CandidateAction(object o)

   {

      Console.WriteLine("Derived.CandidateAction");

   }

}

...

Derived x = new Derived();

SampleDelegate factory = new SampleDelegate(x.CandidateAction);

factory("test");
```

Remember that Snippy[4] will be generating all of this code within a class called `Snippet`, which the nested type derives from. Under C# 1, listing 5.4 would print `Snippet.CandidateAction` because the method taking an `object` parameter wasn't compatible with `SampleDelegate`. Under C# 2, the method *is* compatible, and it's the method chosen due to being declared in a more derived type, so the result is that `Derived.CandidateAction` is printed.

[4] In case you skipped the first chapter, Snippy is a tool I've built to create short but complete code samples. See section 1.8.1 for more details.

Fortunately, the C# 2 compiler knows that this is a breaking change and issues an appropriate warning. I've included this section because you ought to be aware of the possibility of such a problem, but I'm sure it's rarely encountered in real life.

Enough doom and gloom about potential breakage. We've still got to see the most important new feature regarding delegates: anonymous methods. They're a bit more complicated than the topics we've covered so far, but they're also *very* powerful—and a large step toward C# 3.