Covariance and Contravariance in Generic Type Parameters

In Chapter 15, covariance and contravariance were described as they relate to non-generic delegates. That form of contravariance and covariance is still fully supported by C# 4.0 and is quite useful. However, C# 4.0 expands the covariance and contravariance features to include generic type parameters that are used by generic interfaces and generic delegates. One of their principal uses is to streamline certain types of situations encountered when using generic interfaces and delegates defined by the .NET Framework, and some interfaces and delegates defined by the library have been upgraded to use type parameter covariance and contravariance. Of course, they can also be beneficial in interfaces and delegates that you create.

This section explains the generic type parameter covariance and contravariance mechanisms and shows examples of both.

Using Covariance in a Generic Interface

As it applies to a generic interface, covariance is the feature that enables a method to return a type that is derived from the class specified by a type parameter. In the past, because of the strict type-checking applied to generics, the return type had to match the type parameter precisely. Covariance relaxes this rule in a type-safe way. A covariant type parameter is declared by preceding its name with the keyword **out**.

To understand the implications of covariance, it is helpful to work through an example. First, here is a very simple interface called **IMyCoVarGenIF** that uses covariance:

```
// This generic interface supports covariance.
public interface IMyCoVarGenIF<out T> {
  T GetObject();
}
```

Pay special attention to the way that the type parameter **T** is declared. It is preceded by the keyword **out**. When used in this context, **out** specifies that **T** is covariant. Because **T** is covariant, **GetObject( )** can return a reference of type **T** or a reference of any class derived from **T**.

Although covariant on **T**, **IMyCoVarGenIF** is implemented just like any other generic interface. For example, here it is implemented by **MyClass:**

```
// Implement the IMyCoVarGenIF interface.
class MyClass<T> : IMyCoVarGenIF<T> {
  T obj;

  public MyClass(T v) { obj = v; }

  public T GetObject() { return obj; }
}
```

Notice that **out** is not specified again in the interface clause of **MyClass**. Not only is it not needed,

but also it would be an error to attempt to specify it again.

Now, assume the following simple class hierarchy:

```
// Create a simple class hierarchy.
class Alpha {
  string name;
  public Alpha(string n) { name = n; }

  public string GetName() { return name; }
  // ...
}

class Beta : Alpha {
  public Beta(string n) : base(n) { }
  // ...
}
```

Notice that **Beta** is derived from **Alpha**.

Given the foregoing, the following sequence is legal:

```
// Create a IMyCoVarGenIF reference to a MyClass<Alpha> object.
// This is legal with or without covariance.
IMyCoVarGenIF<Alpha> AlphaRef =
     new MyClass<Alpha>(new Alpha("Alpha #1"));

Console.WriteLine("Name of object referred to by AlphaRef is " +
                  AlphaRef.GetObject().GetName());

// Now create a MyClass<Beta> object and assign it to AlphaRef.
// *** This line is legal because of covariance. ***
AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));

Console.WriteLine("Name of object referred to by AlphaRef is now " +
                  AlphaRef.GetObject().GetName());
```

First, an **IMyCoVarGenIF<Alpha>** interface variable called **AlphaRef** is created and is assigned a reference to a **MyClass<Alpha>** object. This is legal because **MyClass** implements **IMyCoVarGenIF**, and both specify **Alpha** for a type argument. Next, the name of the object is displayed by calling **GetName( )** on the object returned by **GetObject( )**. Again, this works because the return type of **GetName( )** is **Alpha** (in this case) and **T** is of type **Alpha**. Next, **AlphaRef** is assigned a reference to an instance of **MyClass<Beta>**. This is legal because **Beta** is derived from **Alpha**, and **T** is covariant in **IMyCoVarGenIF**. If either of these were not the case, the statement would be illegal.

For your convenience, the entire sequence is assembled into the program shown here:

```
// Demonstrate generic interface covariance.
using System;

// This generic interface supports covariance.
public interface IMyCoVarGenIF<out T> {
  T GetObject();
```

```
}

// Implement the IMyCoVarGenIF interface.
class MyClass<T> : IMyCoVarGenIF<T> {
  T obj;

  public MyClass(T v) { obj = v; }
  public T GetObject() { return obj; }
}

// Create a simple class hierarchy.
class Alpha {
  string name;

  public Alpha(string n) { name = n; }

  public string GetName() { return name; }
  // ...
}

class Beta : Alpha {
  public Beta(string n) : base(n) { }
  // ...
}
class VarianceDemo {
  static void Main() {
    // Create a IMyCoVarGenIF reference to a MyClass<Alpha> object.
    // This is legal with or without covariance.
    IMyCoVarGenIF<Alpha> AlphaRef =
         new MyClass<Alpha>(new Alpha("Alpha #1"));

    Console.WriteLine("Name of object referred to by AlphaRef is " +
                      AlphaRef.GetObject().GetName());

      // Now create a MyClass<Beta> object and assign it to AlphaRef.
     // *** This line is legal because of covariance. ***
     AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));
    Console.WriteLine("Name of object referred to by AlphaRef is now " +
                      AlphaRef.GetObject().GetName());
  }
}
```

The output is

```
Name of object referred to by AlphaRef is Alpha #1
Name of object referred to by AlphaRef is now Beta #1
```

It is important to stress that **AlphaRef** can be assigned a reference to a **MyClass<Beta>** object only because **T** is covariant in **IMyCoVarGenIF**. To prove this, remove **out** from in **IMyCoVarGenIF's** declaration of **T**, and then attempt to recompile the program. The compilation will fail because the default strict type-checking will not allow the assignment.

It is possible for one generic interface to be inherited by another. In other words, a generic interface

with a covariant type parameter can be extended. For example,

```
public interface IMyCoVarGenIF2<out T> : IMyCoVarGenIF<T> {
  // ...
}
```

Notice that **out** is specified only in the extending interface's declaration. Specifying **out** in the base interface clause is not necessary or legal. One last point: it is legal for **IMyCoVarGenIF2** to not specify **T** as covariant. However, doing so eliminates the covariance that extending **IMyCoVarGetIF** could provide. Of course, making **IMyCoVarGenIF2** invariant may be required for some uses.

Here are some restrictions that apply to covariance. A covariant type parameter can be applied only to a method return type. Thus, **out** cannot be applied to a type parameter that is used to declare a method parameter. Covariance works only with reference types. A covariant type cannot be used as a constraint in an interface method. For example, this interface is illegal:

```
public interface IMyCoVarGenIF2<out T> {
  void M<V>() where V:T; // Error, covariant T cannot be used as constraint
}
```

Using Contravariance in a Generic Interface

As it applies to a generic interface, contravariance is the feature that lets a method use an argument whose type is a base class of the type specified by the type parameter for that parameter. In the past, because of the strict type-checking applied to generics, a method's argument type had to match the type parameter precisely. Contravariance relaxes this rule in a type-safe way. A contravariant type parameter is declared by preceding the type parameter with the keyword **in**.

To understand the effects of contravariance, we will again work through an example. To begin, here is a contravariant generic interface called **IMyContraVarGenIF**. Notice that its type parameter **T** is contravariant and it uses T in the declaration of a method called **Show( )**.

```
// This generic interface supports contravariance.
public interface IMyContraVarGenIF<in T> {
  void Show(T obj);
}
```

Notice that **T** is specified as contravariant by preceding it with **in**. Also, notice that the parameter type of **obj** is **T**.

Next, **MyClass** implements **IMyContraVarGenIF**, as shown here:

```
// Implement the IMyContraVarGenIF interface.
class MyClass<T> : IMyContraVarGenIF<T> {
    public void Show(T x) { Console.WriteLine(x); }
}
```

Here, **Show( )** simply displays the string representation of **x** (as obtained by **WriteLine( )'s** implicit call to **ToString( )**).

Next, a class hierarchy is declared:

```
// Create a simple class hierarchy.
class Alpha {
  public override string ToString() {
    return "This is an Alpha object.";
  }
  // ...
}
class Beta : Alpha {
  public override string ToString() {
    return "This is a Beta object.";
  }
  // ...
}
```

Notice that these versions of **Alpha** and **Beta** differ from the previous example for the sake of illustration. Also notice that **ToString( )** is overridden to return the type of object.

Given the foregoing, the following sequence is legal:

```
// Create an IMyContraVarGenIF<Alpha> reference to a
// MyClass<Alpha> object.
// This is legal with or without contravariance.
IMyContraVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>();

// Create an IMyContraVarGenIF<Beta> reference to a
// MyClass<Beta> object.
// This is legal with or without contravariance.
IMyContraVarGenIF<beta> BetaRef = new MyClass<Beta>();

// Create an IMyContraVarGenIF<beta> reference to
// a MyClass<Alpha> object.
// *** This is legal because of contravariance. ***
IMyContraVarGenIF<Beta> BetaRef2 = new MyClass<Alpha>();

// This call is legal with or without contravariance.
BetaRef.Show(new Beta());

// Assign AlphaRef to BetaRef.
//  *** This is legal because of contravariance. ***
BetaRef = AlphaRef;

BetaRef.Show(new Beta());
```

First, notice that two **IMyContraVarGenIF** reference variables are created and are assigned references to **MyClass** objects whose type parameters match that of the interface references. The first uses **Alpha**. The second uses **Beta**. These declarations do not require contravariance and are legal in all cases.

Next, an **IMyContraVarGenIF<Beta>** reference is created, but it is assigned a reference to a **MyClass<Alpha>** object. This is legal only because **T** is contravariant.

As you would expect, the next line, which calls **BetaRef.Show( )** with a **Beta** argument, is legal

because **T** in **MyClass<Beta>** is **Beta**, and the argument to **Show( )** is **Beta**.

The next line assigns **AlphaRef** to **BetaRef**. This is legal only because of contravariance. In this case, **BetaRef** is of type **MyClass<Beta>**, but **AlphaRef** is of type **MyClass<Alpha>**. Because **Alpha** is a base class of **Beta**, contravariance makes this conversion legal. To prove to yourself that contravariance is required in the program, try removing **in** from the declaration of **T** in **IMyContraVarGenIF**. Then attempt to recompile the program. As you will see, errors will result.

For your convenience, all the pieces are assembled into the following program:

```
// Demonstrate generic interface contravariance.
using System;

// This generic interface supports contravariance.
public interface IMyContraVarGenIF<in T> {
  void Show(T obj);
}
// Implement the IMyContraVarGenIF interface.
class MyClass<T> : IMyContraVarGenIF<T> {
  public void Show(T x) { Console.WriteLine(x); }
}
// Create a simple class hierarchy.
class Alpha {
  public override string ToString() {
    return "This is an Alpha object.";
  }
  // ...
}
class Beta : Alpha {
  public override string ToString() {
    return "This is a Beta object.";
  }
  // ...
}

class VarianceDemo {
  static void Main() {
    // Create an IMyContraVarGenIF<Alpha> reference to a
    // MyClass<Alpha> object.
    // This is legal with or without contravariance.

    IMyContraVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>();
    // Create an IMyContraVarGenIF<beta> reference to a
    // MyClass<Beta> object.
    // This is legal with or without contravariance.
    IMyContraVarGenIF<Beta> BetaRef = new MyClass<Beta>();

    // Create an IMyContraVarGenIF<beta> reference to
    // a MyClass<Alpha> object.
    // *** This is legal because of contravariance. ***

    IMyContraVarGenIF<Beta> BetaRef2 = new MyClass<Alpha>();
    // This call is legal with or without contravariance.
    BetaRef.Show(new Beta());
```

```
    // Assign AlphaRef to BetaRef.
    //  *** This is legal because of contravariance. ***
    BetaRef = AlphaRef;
    BetaRef.Show(new Beta());
  }
}
```

The output is shown here:

```
This is a Beta object.
This is a Beta object.
```

A contravariant interface can be extended. The process is similar to that described for extending a covariant interface. To access the contravariant nature of the extended interface, the extending interface must specify **in** for the type parameter that corresponds to the contravarient type parameter in the base interface. For example,

```
public interface IMyContraVarGenIF2<in T> : IMyContraVarGenIF<T> {
// ...
}
```

Notice that **in** is not required (nor would it be legal) to be specified in the base interface clause. Furthermore, it is not required that **IMyContraVarGenIF2** be contravariant. In other words, it is not required that **IMyContraVarGenIF2** modify **T** with **in**. Of course, any benefits that could result from the contravariant **IMyContraVarGen** interface would be lost relative to the **IMyContraVarGenIF2** interface.

Contravariance works only with reference types, and a contravariant type parameter can be applied only to method arguments. Thus, **in** cannot be applied to a type parameter that is used for a return type.

Variant Delegates

As explained in Chapter 15, non-generic delegates already support covariance and contravariance as they relate to method return types and parameter types. Beginning with C# 4.0, these features are expanded for generic delegates to include type parameter covariance and contravariance. These features work in a fashion similar to that just described for interfaces.

Here is an example of a contravariant delegate:

```
// Declare a generic delegate that is contravariant on T.
delegate bool SomeOp<in T>(T obj);
```

This delegate can be assigned a method whose parameter is **T** or a class from which **T** is derived.

Here is an example of a covariant delegate:

```
// Declare a generic delegate that is covariant on T.
delegate T AnotherOp<out T, V>(V obj);
```

This delegate can be assigned a method whose return type is **T** or a class derived from **T**. In this

case, **V** is simply an invariant type parameter.

The following program puts these delegates into action:

```
// Demonstrate covariance and contravariance with a generic delegate.

using System;
// Declare a generic delegate that is contravariant on T.
delegate bool SomeOp<in T>(T obj);

// Declare a generic delegate that is covariant on T.
delegate T AnotherOp<out T, V>(V obj);

class Alpha {
  public int Val { get; set; }

  public Alpha(int v) { Val = v; }
}
class Beta : Alpha {
  public Beta(int v) : base(v) { }
}

class GenDelegateVarianceDemo {
  // Return true if obj.Val is even.
  static bool IsEven(Alpha obj) {
     if((obj.Val % 2) == 0) return true;
    return false;
   }

  static Beta ChangeIt(Alpha obj) {
    return new Beta(obj.Val +2);
  }

  static void Main() {
    Alpha objA = new Alpha(4);
    Beta objB = new Beta(9);

    // First demonstrate contravariance.

    // Declare a SomeOp<Alpha> delegate and set it to IsEven.
    SomeOp<Alpha> checkIt = IsEven;

    // Declare a SomeOp<Beta> delegate.
    SomeOp<Beta> checkIt2;

    // Now, assign the SomeOp<Alpha> delegate the SomeOp<Beta> delegate.
    // *** This is legal only because of contravariance. ***
    checkIt2 = checkIt;

    // Call through the delegate.
    Console.WriteLine(checkIt2(objB));

    // Now, demonstrate covariance.

    // First, declare two AnotherOp delegates.
```

```
    // Here, the return type is Beta and the parameter type is Alpha.
    // Notice that modifyIt is set to ChangeIt.
    AnotherOp<Beta, Alpha> modifyIt = ChangeIt;

    // Here, the return type is Alpha and the parameter type is Alpha.
    AnotherOp<Alpha, Alpha> modifyIt2;

    // Now, assign modifyIt to modifyIt2.
    // *** This statement is legal only because of covariance. ***
    modifyIt2 = modifyIt;

    // Actually call the method and display the results.
    objA = modifyIt2(objA);
    Console.WriteLine(objA.Val);
  }
}
```

The output is shown here:

```
False
6
```

The comments in the program explain each operation. It is important to stress, however, that for both **SomeOp** and **AnotherOp**, the use of **in** and **out**, respectively, is necessary for the program to compile. Without these modifiers, compilation errors would result at the indicated lines because no implicit conversions would be available.