Covariance and Contravariance in Generics

As you've seen throughout this chapter, when you create an instance of a generic type, the compiler takes the generic type declaration and the type arguments and creates a constructed type. A mistake that people commonly make, however, is to assume that you can assign a delegate of a derived type to a variable of a delegate of a base type. In the following sections, we'll look at this topic, which is called variance. There are three types of variance—*covariance*, *contravariance*, and *invariance*.

We'll start by reviewing something you've already learned: every variable has a type assigned to it, and you can assign an object of a more derived type to a variable of one of its base types. This is called *assignment compatibility*. The following code demonstrates assignment compatibility with a base class `Animal` and a class `Dog` derived from `Animal`. In `Main`, you can see that the code creates an object of type `Dog` and assigns it to variable `a2` of type `Animal`.

```
class Animal
{
   public int NumberOfLegs = 4;
}

class Dog : Animal
{
}

class Program
{
   static void Main( )
   {
      Animal a1 = new Animal( );
      Animal a2 = new Dog( );

      Console.WriteLine( "Number of dog legs: {0}", a2.NumberOfLegs );
   }
}
```

Figure 19-12 illustrates assignment compatibility. In this figure, the boxes showing the `Dog` and `Animal` objects also show their base classes.

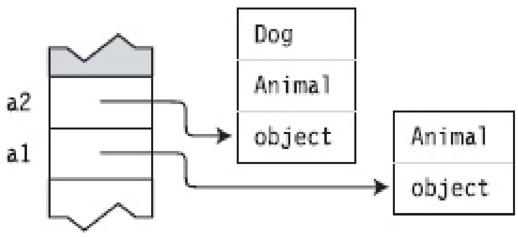

Figure 19-12. *Assignment compatibility means that you can assign a reference of a more derived type to a variable of a less derived type.*

Now let's look at a more interesting case by expanding the code in the following ways as shown following:

- This code adds a generic delegate named `Factory`, which takes a single type parameter `T`, takes no method parameters, and returns an object of type `T`.
- I've added a method named `MakeDog` that takes no parameters and returns a `Dog` object. This method, therefore, matches delegate `Factory` if we use `Dog` as the type parameter.
- The first line of `Main` creates a delegate object whose type is `delegate Factory<Dog>` and assigns its reference to variable `dogMaker`, of the same type.
- The second line attempts to assign a delegate of type `delegate Factory<Dog>` to a delegate type variable named `animalMaker` of type `delegate Factory<Animal>`.

This second line in `Main`, however, causes a problem, and the compiler produces an error message saying that it can't implicitly convert the type on the right to the type on the left.

```
class Animal       { public int Legs = 4; }  // Base class
class Dog : Animal { }                       // Derived class

delegate T Factory<T>( );          ← delegate Factory

class Program
{
   static Dog MakeDog( )           ← Method that matches delegate Factory
   {
      return new Dog( );
   }

   static void Main( )
   {
      Factory<Dog>    dogMaker    = MakeDog;    ← Create delegate object
      Factory<Animal> animalMaker = dogMaker;   ← Attempt to assign delegate object

      Console.WriteLine( animalMaker( ).Legs.ToString( ) );
   }
}
```

It seems to make sense that a delegate constructed with the base type should be able to hold a delegate constructed with the derived type. So why does the compiler give an error message? Doesn't the principle of assignment compatibility hold?

The principle *does* hold, but it doesn't apply in this situation! The problem is that although `Dog` derives from `Animal`, delegate `Factory<Dog>` does *not* derive from delegate `Factory<Animal>`. Instead, both delegate objects are peers, deriving from type d `elegate`, which derives from type `object`, as shown in Figure 19-13. Neither delegate is derived from the other, so assignment compatibility doesn't apply.

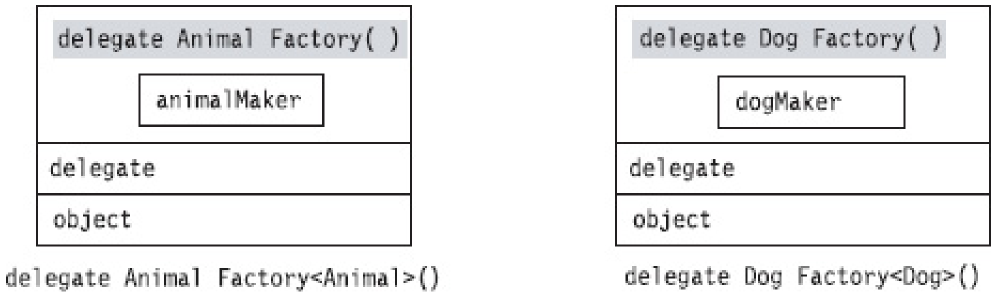

Figure 19-13. *Assignment compatibility doesn't apply because the two delegates are unrelated by inheritance.*

Although the mismatch of delegate types doesn't allow assigning one type to the variable of another type, it's too bad in this situation, because in the example code, any time we would execute delegate `animalMaker`, the calling code would expect to have a reference to an `Animal` object returned. If it returned a reference to a `Dog` object instead, that would be perfectly fine since a reference to a `Dog` is a reference to an `Animal`, by assignment compatibility.

Looking at the situation more carefully, we can see that for any generic delegate, if a type parameter is used *only as an output value*, then the same situation applies. In all such situations, you would be able to use a constructed delegate type created with a derived class, and it would work fine, since the invoking code would always be expecting a reference to the base class—which is exactly what it would get.

This constant *relation* between the use of a derived type only as an output value, and the validity of the constructed delegate, is called *covariance*, and is now explicitly allowed in C# 4.0. To let the compiler know that this is what you intend, you must mark the type parameter in the delegate declaration with the `out` keyword.

For example, if we change the delegate declaration in the example by adding the `out` keyword, as shown here, the code compiles and works fine.

```
delegate T Factory<out T>( );
                   ↑
         Keyword specifying covariance
            of the type parameter
```

Figure 19-14 illustrates the components of covariance in this example:

- The variable on the stack on the left is of type delegate `T Factory<out T>()`, where type variable `T` is of class `Animal`.
- The actual constructed delegate in the heap, on the right, was declared with a type variable of class `Dog`, which is derived from class `Animal`.
- This is acceptable because when the delegate is called, the calling code receives an object of type `Dog`, instead of the expected object of type `Animal`. The calling code can freely operate on the `Animal` part of the object as it expects to do.

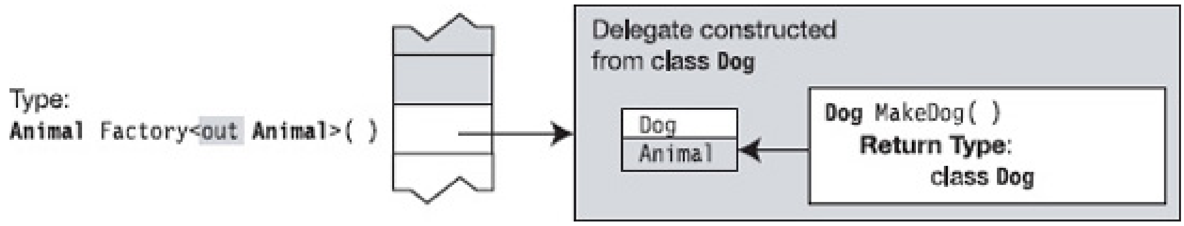

Figure 19-14. *The covariant relation allows a more derived type to be in return and out positions.*

The following code illustrates a related situation. In this example, there's a delegate, named `Action1` , which takes a single type parameter, and a single method parameter whose type is that of the type parameter, and it returns no value.

The code also contains a method called `ActOnAnimal` , whose signature and `void` return type match the delegate declaration.

The first line in `Main` creates a constructed delegate using type `Animal` and method `ActOnAnimal` , whose signature and void return type match the delegate declaration. In the second line, however, the code attempts to assign the reference to this delegate to a stack variable named `dog1` , of type `delegate Action1<Dog>` .

```
class Animal { public int NumberOfLegs = 4; }
class Dog : Animal { }

class Program           Keyword for Contravariance
{                              ↓
   delegate void Action1<in T>( T a );

   static void ActOnAnimal( Animal a ) { Console.WriteLine( a.NumberOfLegs ); }

   static void Main( )
   {
      Action1<Animal> act1 = ActOnAnimal;
      Action1<Dog>    dog1 = act1;
      dog1( new Dog() );
   }
}
```

This code produces the following output:

```
4
```

Like the previous situation, by default, you can't assign the two incompatible types. But also like the previous situation, there are situations where the assignment would work perfectly fine.

As a matter of fact, this is true whenever the type parameter is used *only as an input parameter* to the method in the delegate. The reason for this is that even though the invoking code passes in a reference to a more derived class, the method in the delegate is only expecting a reference to a less derived class—which of course it receives and knows how to manipulate.

This relation, allowing a more derived object where a less derived object is expected, is called *contravariance* and is now explicitly allowed in C# 4.0. To use it, you must use the `in` keyword with the type parameter, as shown in the code.

Figure 19-15 illustrates the components of contravariance in line 2 of `Main` .

- The variable on the stack on the left is of type delegate `void Action1<in T>(T p)` , where the type variable is of class `Dog` .
- The actual constructed delegate, on the right, is declared with a type variable of class `Animal` , which is a base class of class `Dog` .
- This works fine because when the delegate is called, the calling code passes in an object of type `Dog` , to method `ActOnAnimal` , which is expecting an object of type `Animal` . The method can freely operate on the `Animal` part of the object as it expects to do.

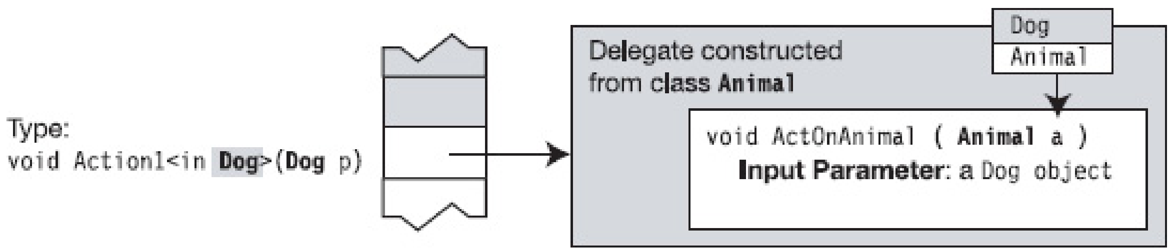

Figure 19-15. *The contravariant relation allows more derived types to be allowed as input parameters.*

Figure 19-16 summarizes the differences between covariance and contravariance in a generic delegate.

- The top figure illustrates covariance.
  - The variable on the stack on the left is of type delegate `F<out T>( )` where the type variable is of a class named `Base`.
  - The actual constructed delegate, on the right, was declared with a type variable of class `Derived`, which is derived from class `Base`.
  - This works fine because when the delegate is called, the method returns a reference to an object of the derived type, which is also a reference to the base class, which is exactly what the calling code is expecting.
- The bottom figure illustrates contravariance.
  - The variable on the stack on the left is of type delegate `void F<in T>(T p)`, where the type parameter is of class `Derived`.
  - The actual constructed delegate, on the right was declared with a type variable of class `Base`, which is a base class of class `Derived`.
  - This works fine because when the delegate is called, the calling code passes in an object of the derived type, to the method which is expecting an object of the base type. The method can operate freely on the base part of the object as it expects to do.

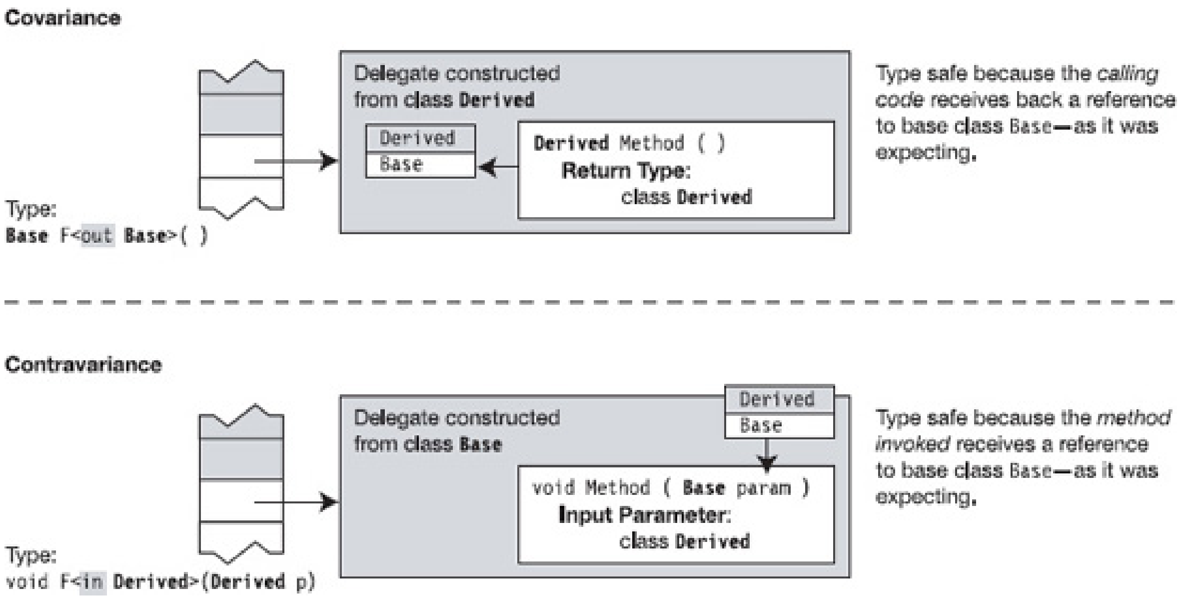

Figure 19-16. *A comparison of covariance and contravariance*

Covariance and Contravariance in Interfaces

You should now have an understanding of covariance and contravariance as it applies to delegates. The same principles apply to interfaces, including the syntax using the `out` and `in` keywords in the interface declaration.

The following code shows an example of using covariance with an interface. The things to note about the code are the following:

- The code declares a generic interface with type parameter `T`. The `out` keyword specifies that the type parameter is covariant.
- Generic class `SimpleReturn` implements the generic interface.
- Method `DoSomething` shows how a method can take an *interface* as a parameter. This method takes as its parameter a generic `IMyIfc` interface constructed with type `Animal`.

The code works in the following way:

- The first two lines of `Main` create and initialize a constructed instance of generic class `SimpleReturn`, using class `Dog`.
- The next line assigns that object to a variable on the stack that is declared of constructed interface type `IMyIfc<Animal>`. Notice several things about this declaration:
  - The type on the left of the assignment is an interface type—not a class.
  - Even though the interface types don't exactly match, the compiler allows them because of the covariant `out` specifier in the

interface declaration.

- Finally, the code calls method `DoSomething` with the constructed covariant class that implements the interface.

```
class Animal { public string Name; }
class Dog: Animal{ };
                 Keyword for Covariance

interface IMyIfc<out T>
{
    T GetFirst();
}

class SimpleReturn<T>: IMyIfc<T>
{
    public T[] items = new T[2];
    public T GetFirst() { return items[0]; }
}

class Program
{
    static void DoSomething(IMyIfc<Animal> returner)
    {
        Console.WriteLine(returner.GetFirst().Name);
    }

    static void Main( )
    {
        SimpleReturn<Dog> dogReturner = new SimpleReturn<Dog>();
        dogReturner.items[0] = new Dog() { Name = "Avonlea" };

        IMyIfc<Animal> animalReturner = dogReturner;

        DoSomething(dogReturner);
    }
}
```

This code produces the following output:

```
Avonlea
```

More About Variance

The previous two sections explained explicit covariance and contravariance. There is also a situation where the compiler automatically recognizes that a certain constructed delegate is covariant or contravariant and makes the type coercion automatically. That happens when the object hasn't yet had a type assigned to it. The following code shows an example.

The first line of `Main` creates a constructed delegate of type `Factory<Animal>` from a method where the return type is a `Dog` object, not an `Animal` object. In creating this delegate, the method name on the right side of the assignment operator doesn't yet have a type, and the compiler can determine that the method fits the type of the delegate except that its return type is of type `Dog` rather than type `Animal`. The compiler is smart enough to realize that this is a covariant relation and creates the constructed type and assigns it to the variable.

Compare that with the assignments in the third and fourth lines of `Main`. In these cases, the expressions on the right side of the equals sign already have a type and therefore need the `out` specifier in the delegate declaration to signal the compiler to allow them to be covariant.

```
class Animal { public int Legs = 4; }              // Base class
class Dog : Animal { }                             // Derived class

class Program
{
   delegate T Factory<out T>();

   static Dog MakeDog() { return new Dog(); }
   static void Main()
   {
      Factory<Animal> animalMaker1 = MakeDog;      // Coerced implicitly
```

```
        Factory<Dog>    dogMaker     = MakeDog;
        Factory<Animal> animalMaker2 = dogMaker;     // Requires the out specifier


        Factory<Animal> animalMaker3
                  = new Factory<Dog>(MakeDog);       // Requires the out specifier
    }
}
```

This implicit coercion implementing covariance and contravariance has been available without the `in` / `out` keywords since before C# 4.0.

Other important things you should know about variance are the following:

- As you've seen, variance deals with the issue of where it's safe to substitute a base type for a derived type, and vice versa. Variance, therefore, applies only to reference types, since value types can't be derived from.
- Explicit variance, using the `in` and `out` keywords applies only to delegates and interfaces—not classes, structs, or methods.
- Delegate and interface type parameters that don't include either the `in` or `out` keyword are called *invariant*. These types cannot be used covariantly or contravariantly.