**Username:** Pralay Patoria **Book:** Effective C# (Covers C# 4.0): 50 Specific Ways to Improve Your C#, Second Edition, Video Enhanced Edition. 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.

# Item 29: Support Generic Covariance and Contravariance

Complementary Video: Generics and Variance

Type variance, and specifically, covariance and contravariance define the conditions under which one type can be substituted for another type. Whenever possible, you should decorate generic interfaces and delegate definitions to support generic covariance and contravariance. Doing so will enable your APIs to be used in more different ways, and safely. If you cannot substitute one type for another, it is called invariant.

Type variance is one of those topics that many developers have encountered but not really understood. Covariance and contravariance are two different forms of type substitution. A return type is covariant if you can substitute a more derived type than the type declared. A parameter type is contravariant if you can substitute a more base parameter type than the type declared. Object-oriented languages generally support covariance of parameter types. You can pass an object of a derived type to any method that expects a more base type. For example, Console.WriteLine() has an overload that takes a System.Object parameter. You can pass an instance of any type that derives from object. When you override an instance of a method that returns a System.Object, you can return anything that is derived from System.Object.

That common behavior led many developers to believe that generics would follow the same rules. You should be able to use an IEnumerable<MyDerived Type> with a method that has a parameter of IEnumerable<Object>. You would expect that if a method returns an IEnumerable<MyDerivedType>, you could assign that to a variable of type IEnumerable<object>. No so. Prior to C# 4.0, all generic types were invariant. That meant there were many times when you would reasonably expect covariance or contravariance with generics only to be told by the compiler that your code was invalid. Arrays were treated covariantly. However, Arrays do not support safe covariance. As of C# 4.0, new keywords are available to enable you to use generics covariantly and contravariantly. That makes generics much more useful, especially if you remember to include the in and out parameters where possible on generic interfaces and delegates.

Let's begin by understanding the problems with array covariance. Consider this small class hierarchy:

```
abstract public class CelestialBody
{
    public double Mass { get; set; }
    public string Name { get; set; }
    // elided
}

public class Planet : CelestialBody
{
    // elided
```

```
}

public class Moon : CelestialBody
{
    // elided
}
public class Asteroid : CelestialBody
{
    // elided
}
```

This method treats arrays of CelestialBody objects covariantly, and does so safely:

```
public static void CoVariantArray(CelestialBody[] baseItems)
{
    foreach (var thing in baseItems)
        Console.WriteLine("{0} has a mass of {1} Kg",
        thing.Name, thing.Mass);
}
```

This method also treats arrays of CelestialBody objects covariantly, but it is not safe. The assignment statement will throw an exception.

```
public static void UnsafeVariantArray(
    CelestialBody[] baseItems)
{
    baseItems[0] = new Asteroid
        { Name = "Hygiea", Mass = 8.85e19 };
}
```

You can have the same problem simply by assigning an array of a derived class to a variable that is an array of a base type:

```
CelestialBody[] spaceJunk = new Asteroid[5];
spaceJunk[0] = new Planet();
```

Treating collections as covariant means that when there is an inheritance relationship between two types, you can imagine there is a similar inheritance relationship between arrays of those two types. This isn't a strict definition, but it's a useful picture to keep in your mind. A Planet can be passed to any method that expects CelestialBody. That's because Planet is derived from CelestialBody. Similarly, you can pass a Planet[] to any method that expects a CelestialBody[]. But, as the above example shows, that doesn't always work the way you'd expect.

When generics were first introduced, this issue was dealt with in a rather draconian fashion. Generics were always treated invariantly. Generic types had to have an exact match. However, in C# 4.0, you can now decorate generic interfaces such that they can be treated covariantly, or contravariantly. Let's discuss generic covariance first, and then we'll move on to contravariance.

This method can be called with a List<Planet>:

```
public static void CoVariantGeneric(
    IEnumerable<CelestialBody> baseItems)
```

```
{
    foreach (var thing in baseItems)
        Console.WriteLine("{0} has a mass of {1} Kg",
        thing.Name, thing.Mass);
}
```

That's because IEnumerable<T> has been augmented to limit T to only output positions in its interface:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> :
    IDisposable, IEnumerator
{
    T Current { get; }
    // MoveNext(), Reset() inherited from IEnumerator
}
```

I included both the IEnumerable<T> and IEnumerator<T> definition here, because the IEnumerator<T> has the important restrictions. Notice that IEnumerator<T> now decorates the type parameter T with the out modifier. That forces the compiler to limit T to output positions. Output positions are limited to function return values, property get accessors, and certain delegate positions.

Therefore, using IEnumerable<out T>, the compiler knows that you will look at every T in the sequence, but never modify the contents of the source sequence. Treating every Planet as a CelestialBody in this case works.

IEnumerable<T> can be covariant only because IEnumerator<T> is also covariant. If IEnumerable<T> returned an interface that was not declared as covariant, the compiler would generate an error. Covariant types must return either the type parameter, or an interface on the type parameter that is also covariant.

However, the method that replaces the first item in the list will be invariant when using generics:

```
public static void InvariantGeneric(
    IList<CelestialBody> baseItems)
{
    baseItems[0] = new Asteroid
        { Name = "Hygiea", Mass = 8.85e19 };
}
```

Because IList<T> is neither decorated with the in or out modifier on T, you must use the exact type match.

Of course, you can create Contravariant generic interfaces and delegates as well. Substitute the in modifier for the out modifier. That instructs the compiler that the type parameter may only appear in input positions. The .NET Framework has added the in modifier to the IComparable<T> interface:

```
public interface IComparable<in T>
{
```

```
    int CompareTo(T other);
}
```

That means you could make CelestialBody implement IComparable<T>, using an object's mass. It would compare two Planets, a Planet and a Moon, a Moon and an Asteroid, or any other combination. By comparing the mass of the objects, that's a valid comparison.

You'll notice that IEquatable<T> is invariant. By definition, a Planet cannot be equal to a Moon. They are different types, so it makes no sense. It is necessary, if not sufficient, for two objects to be of the same type if they are equal (see Item 6).

Type parameters that are contravariant can only appear as method parameters, and some locations in delegate parameters.

By now, you've probably noticed that I've used the phrase "some locations in delegate parameters" twice. Delegate definitions can be covariant or contravariant as well. It's usually pretty simple: Method parameters are contravariant (in), and method return types are covariant (out). The BCL updated many of their delegate definitions to include variance:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, T2, out TResult>(T1 arg1,
    T2 arg2);
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, T3>(T1 arg1,
    T2 arg2, T3 arg3);
```

Again, this probably isn't too hard. But, when you mix them, things start to get to be mind benders. You already saw that you cannot return invariant interfaces from covariant interfaces. You can't use delegates to get around the covariant and contravariant restrictions, either.

Delegates have a tendency to "flip" the covariance and contravariance in an interface if you're not careful. Here are a couple examples:

```
public interface ICovariantDelegates<out T>
{
    T GetAnItem();
    Func<T> GetAnItemLater();
    void GiveAnItemLater(Action<T> whatToDo);
}

public interface IContravariantDelegates<in T>
{
    void ActOnAnItem(T item);
    void GetAnItemLater(Func<T> item);
    Action<T> ActOnAnItemLater();
}
```

I've named the methods in these interfaces specifically to show why it works the way it does. Look closely at the ICovariantDelegate interface definition. GetAnItemLater() is just a way to retrieve an item lazily. The caller can invoke the Func<T> returned by the method later to retrieve a value. T

still exists in the output position. That probably still makes sense. The GetAnItemLater() method probably is a bit more confusing. Here, you're method takes a delegate that will accept a T object whenever you call it. So, even though Action<in T> is covariant, its position in the ICovariantDelegate interface means that it is actually a means by which T objects are returned from an ICovariantDelegate<T> implementing object. It may look like it should be contravariant, but it is covariant with respect to the interface.

IContravariantDelegate<T> is similar but shows how you can use delegates in a contravariant interface. Again, the ActOnAnItem method should be obvious. The ActOnAnItemLater() method is just a little more complicated. You're returning a method that will accept a T object sometime later. That last method, once again, may cause some confusion. It's the same concept as with the other interface. The GetAnItemLater() method accepts a method that will return a T object sometime later. Even though Func<out T> is declared covariant, its use is to bring an input to the object implementing IContravariantDelegate. Its use is contravariant with respect to the IContravariantDelegate.

It certainly can get complicated describing exactly how covariance and contravariance work. Thankfully, now the language supports decorating generic interfaces and delegates with `in` (contravariant) and `out` (covariant) modifiers. You should decorate any interfaces and delegates you define with the `in` or `out` modifiers wherever possible. Then, the compiler can correct any possible misuses of the variance you've defined. The compiler will catch it both in your interface and delegate definitions, and it will detect any misuse of the types you've created.