Covariance and Contravariance

Covariance and contravariance are big terms that describe how conversions are performed between types.[6]

Consider the following:

```
class Auto
{
}
class Sedan: Auto
{
}
void ReferenceCovariance()
{
    Sedan dodgeDart = new Sedan();
    Auto currentCar = dodgeDart;
}
```

This works exactly as you would expect; because `Sedan` is derived from `Auto`, every `Sedan` is an `Auto`, and therefore you can safely make this assignment.

When you extend this to arrays of reference types, it gets more interesting.

```
void ArrayCovariance()
{
    Sedan[] sedans = new Sedan[1];
    sedans[0] = new Sedan();
    Auto[] autos = sedans;
    autos[0] = new Roadster();
}
```

It is useful to be able to assign an array of `Sedan` instances to an array of `Auto` instances; this allows you to write methods that take an array of `Auto` instances as a parameter. Unfortunately, it isn't typesafe; the last statement in the method assigns a `Roadster` instance to the `autos` array. That would be fine if the `autos` array was actually of type `Auto[]`, but it is in fact of type `Sedan[]`, and the assignment fails at runtime.[7]

This behavior is a bit unfortunate. It would be nice if generics provided a better solution. Consider the following example:

```
interface IFirstItem <T>
{
    T GetFirstItem();
}
class MyFirstList <T> : List <T>, IFirstItem <T>
{
    public MyFirstList () { }
    public T GetFirstItem()
    {
       return this[0];
    }
}
```

Here you define an interface named `IFirstItem < T >` and a list class that implements it. You then write some code to use it.

```
void TestService()
{
    MyFirstList <Sedan> sedans = new MyFirstList <Sedan> ();
    sedans.Add(new Sedan());
    PerformService(sedans);
```

```
}
void PerformService(IFirstItem <Auto> autos)
{
}
```

You are passing an `IFirstItem <Sedan>` to a function that takes an `IFirstItem <Auto>`, and that's not allowed. The compiler is worried that `PerformService()` will lose the fact that the `Auto` is really a `Sedan` and try to do something that will generate an exception. It's the same situation you had with the array.

If you examine the `IFirstItem <T>` interface, you will realize that there is no issue; the only thing that it does is pull an instance of type `T` out, and there is no way for that to cause an issue. What you need is a way to tell the compiler that the type parameter `T` is used only as output.

You can do that through the following:

```
interface IFirstItem <out T>
{
    T GetFirstItem();
}
```

The code now works. This is an example of generic covariance; the compiler now knows that it is safe to convert from the type of `T` to a less-derived type, so it allows you to do the conversion.

You now try to extend the interface by adding an additional method.

```
interface IFirstItem <out T>
{
    T GetFirstItem();
    void NotLegal(T parameter);
}
```

This generates an error.[8] You said that you are going to use the generic parameter `T` only for output, but the `NotLegal()` method uses it for input.

Contravariance

Contravariance applies in a different case. Consider the following:

```
interface IEqual <in T>
{
    bool IsEqual(T x, T y);
}
class Comparer : IEqual <object>
{
    public bool IsEqual(object x, object y)
    {
       return true;
    }
}
class GenericContravariance
{
    void Example()
    {
       Comparer comparer = new Comparer();
       TestEquality(comparer);
    }
    void TestEquality(IEqual <Auto> equalizer)
    {
    }
}
```

In this case, instances of type `T` flow only into the interface and are never visible outside of the interface. That allows you to do something that seems a bit surprising; you can pass an `IEqual <object>` for use as an `IEqual <Auto>`. That just seems wrong. However, if you look a bit closer, you will figure out that if you have an `IEqual <Auto>`, you will want to use it in code such as this:

```
Auto auto1 = ...;
```

```
Auto auto2 = ...;
bool equal = equalizer.IsEqual(auto1, auto2);
```

In that situation, it is perfectly safe to use an `IEqual <object>`, since you can safely convert the `Auto` arguments into `object` arguments. You indicate this situation by adding the `in` keyword to the type parameter.