Covariance and Contravariance

A common question asked by new users of generic types is why an expression of type `List<string>` may not be assigned to a variable of type `List<object>` —if a `string` may be converted to type `object`, surely a list of strings is similarly compatible with a list of objects. But this is not, generally speaking, either type-safe or legal. If you declare two variables with different type parameters using the same generic class, the variables are not type-compatible even if they are assigning from a more specific type to a more generic type—in other words, they are not **covariant.**

Covariant is a technical term from category theory, but the idea is straightforward: Suppose two types `X` and `Y` have a special relationship, namely that every value of the type `X` may be converted to the type `Y`. If the types `I<X>` and `I<Y>` always also have that same special relationship, we say, " `I<T>` is covariant in `T`." When dealing with simple generic types with only one type parameter, the type parameter can be understood and we simply say, " `I<T>` is covariant." The conversion from `I<X>` to `I<Y>` is called a **covariant conversion.**

For example, instances of a generic class, `Pair<Contact>` and `Pair<PdaItem>`, are not type-compatible even when the type arguments are themselves compatible. In other words, the compiler prevents converting (implicitly or explicitly) `Pair<Contact>` to `Pair<PdaItem>`, even though `Contact` derives from `PdaItem`. Similarly, converting `Pair<Contact>` to the interface type `IPair<PdaItem>` will also fail. See Listing 11.40 for an example.

Listing 11.40. Conversion between Generics with Different Type Parameters

```
// ...
// Error: Cannot convert type ...
Pair<PdaItem> pair = (Pair<PdaItem>) new Pair<Contact>();
IPair<PdaItem> duple = (IPair<PdaItem>) new Pair<Contact>();
```

But why is this not legal? Why are `List<T>` and `Pair<T>` not covariant? Listing 11.41 shows what would happen if the C# language allowed unrestricted generic covariance.

Listing 11.41. Preventing Covariance Maintains Homogeneity

```
//...
Contact contact1 = new Contact("Princess Buttercup"),
Contact contact2 = new Contact("Inigo Montoya");
Pair<Contact> contacts = new Pair<Contact>(contact1, contact2);


// This gives an error: Cannot convert type ...
// But suppose it did not.
// IPair<PdaItem> pdaPair = (IPair<PdaItem>) contacts;
// This is perfectly legal, but not type safe.
// pdaPair.First = new Address("123 Sesame Street");

...
```

An `IPair<PdaItem>` can contain an address, but the object is really a `Pair<Contact>` that can only contain contacts, not addresses. Type safety is completely violated if unrestricted generic covariance is allowed

Now it should also be clear why a list of strings may not be used as a list of objects; you cannot insert an integer into a list of strings, but you can insert an integer into a list of objects, so it must be illegal to cast a list of strings to a list of objects so that this error can be prevented by the compiler.

Enabling Covariance with the `out` Type Parameter Modifier in C# 4.0

You might have noticed that both of the problems described above as consequences of unrestricted covariance arise because the generic pair and the generic list allow their contents to be written. Suppose we eliminated this possibility by making a read-only `IReadOnlyPair<T>` interface that only exposes `T` as coming "out" of the interface (that is, used as the return type of a method or read-only property) and never "into" it (that is, used as a formal parameter or writeable property type). If we restricted ourselves to an "out only" interface with respect to `T`, the covariance problem just described would not occur (see Listing 11.42).

Listing 11.42. Potentially Possible Covariance

```
interface IReadOnlyPair<T>
{
  T First { get; }
  T Second { get; }
}
```

```
interface IPair<T>
{
  T First { get; set; }
  T Second { get; set; }
}
```

```
public struct Pair<T> : IPair<T>, IReadOnlyPair<T>
{
  // ...
}
```

```
class Program
{
  static void Main()
  {
      // Error: Only theoretically possible without
      // the out type parameter modifier
     Pair<Contact> contacts =
         new Pair<Contact>(
             new Contact("Princess Buttercupt"),
             new Contact("Inigo Montoya") );
     IReadOnlyPair<PdaItem> pair = contacts;
     PdaItem pdaItem1 = pair.First;
     PdaItem pdaItem2 = pair.Second;

  }
}
```

By restricting the generic type declaration to only expose data out of the interface, there is no reason for the compiler to prevent covariance. All operations on an `IReadOnlyPair<PdaItem>` instance would convert `Contact`s (from the original `Pair<Contact>` object) up to the base class `PdaItem`—a perfectly valid conversion. There is no way to "write" an address into the object that is really a pair of contacts, because the interface does not expose any writeable properties.

The code above still does not compile. However, support for safe covariance was added to C# 4. To indicate that a generic interface is intended to be covariant in one of its type parameters, declare the type parameter with the `out` type parameter modifier. Listing 11.43 shows how to modify the interface declaration to indicate that it should be allowed to be covariant.

Listing 11.43. Covariance Using the `out` Type Parameter Modifier

```
...
interface IReadOnlyPair<out T>

{
  T First { get; }
  T Second { get; }
}
```

Modifying the type parameter on the `IReadOnlyPair<out T>` interface with `out` will cause the compiler to verify that indeed `T` is used only for "outputs"—method return types and read-only property return types—and never for formal parameters or property setters. From then on, the compiler will allow any covariant conversions involving the interface to succeed. With this modification made to the code in Listing 11.42, the code will now compile and execute successfully.

There are a number of important restrictions on covariant conversions.

- Only generic interfaces and generic delegates (described in Chapter 12) may be covariant. Generic classes and structs are never covariant.
- The varying type arguments of both the "source" and "target" generic types must be reference types, not value types. That is, an `IReadOnlyPair<string>` may be converted covariantly to `IReadOnlyPair<object>` because both

`string` and `IReadOnlyPair<object>` are reference types. An `IReadOnlyPair<int>` may not be converted to `IReadOnlyPair<object>` because `int` is not a reference type.

- The interface or delegate must be declared as supporting covariance, and the compiler must be able to verify that the annotated type parameters are in fact used in only "output" positions.

Enabling Contravariance with the `in` Type Parameter Modifier in C# 4.0

Covariance that "goes backward" is called **contravariance.** Again, suppose two types `X` and `Y` are related such that every value of the type `X` may be converted to the type `Y`. If the types `I<X>` and `I<Y>` always have that same special relationship "backward"—that is, every value of the type `I<Y>` can be converted to the type `I<X>`—we say "`I<T>` is contravariant in `T`."

Most people find that contravariance is much harder to comprehend than covariance is. The canonical example of contravariance is a comparer. Suppose you have a derived type, `Apple`, and a base type, `Fruit`. Clearly, they have the special relationship: Every value of type `Apple` may be converted to `Fruit`.

Now suppose you have an interface `ICompareThings<T>` that has a method `bool FirstIsBetter(T t1, T t2)` that takes two `T`s, and returns a `bool` saying whether the first one is better than the second one.

What happens when we provide type arguments? An `ICompareThings<Apple>` has a method that takes two `Apple`s and compares them. An `ICompareThings<Fruit>` has a method that takes two `Fruit`s and compares them. But since every `Apple` is a `Fruit`, clearly a value of type `ICompareThings<Fruit>` can be safely used anywhere that an `ICompareThings<Apple>` is needed. The "direction" of the convertibility has been "reversed"; hence the name "contra-variance."

Perhaps unsurprisingly, the opposite of the restrictions on a covariant interface are necessary to ensure safe contravariance. An interface that is contravariant in one of its type parameters must use that type parameter only in "input" positions such as formal parameters. (Or in the types of write-only properties, which are extremely rare.) You can mark an interface as being contravariant by declaring the type parameter with the `in` modifier, as shown in listing 11.44.

Listing 11.44. Contravariance Using the `in` Type Parameter Modifier

```
class Fruit {}
class Apple : Fruit {}
class Orange : Fruit {}
```

```
interface ICompareThings<in T>
{
  bool FirstIsBetter(T t1, T t2);

}
```

```
class Program
{
  class FruitComparer : ICompareThings<Fruit>
  { ... }
  static void Main()
  {
      // Allowed in C# 4.0
      ICompareThings<Fruit> fc = new FruitComparer();
      Apple apple1 = new Apple();
      Apple apple2 = new Apple();
      Orange orange = new Orange();
      // A fruit comparer can compare apples and oranges:
      bool b1 = fc.FirstIsBetter(apple1, orange);
      // or apples and apples:
      bool b2 = fc.FirstIsBetter(apple1, apple2);
      // This is legal because the interface is
      // contravariant:
      ICompareThings<Apple> ac = fc;
      // This is really a fruit comparer, so it can
      // compare two apples still.
      bool b3 = ac.FirstIsBetter(apple1, apple2);
  }
}
```

Notice that similar to covariance support, contravariance uses a type parameter modifier: `in`, on the interface's type parameter declaration. This instructs the compiler to check that `T` never appears on a property getter or as the return type of a method, thus enabling contravariant conversions for this interface.

Contravariant conversions have all the analogous restrictions as described above for covariant conversions: They are only valid on generic interface and delegate types, the varying type arguments must be reference types, and the compiler must be able to verify that the interface is safe for the contravariant conversions.

An interface can be covariant in one type parameter and contravariant in the other. Imagine, for example, a device that can transform one thing into another described by the `ITransformer<in TSource, out TTarget>` interface defined in Listing 11.45.

Listing 11.45. Combining Covariance and Contravariance in a Single Generic Type

```
class Food {}
class Pizza : Food {}
class Salad : Food {}
class Document {}
class ComputerProgram : Document {}
interface ITransformer<in TSource, out TTarget>
{
  TTarget Transform(TSource source);
}
// A computer programmer is a device which transforms
// food into computer programs:

class Programmer : ITransformer<Food, ComputerProgram>
{
  public ComputerProgram Transform(Food f) { ... }
}
class Program
{
  static void Main()
  {
    var programmer = new Programmer();
    ComputerProgram cp = programmer.Transform(new Salad());
    // A computer programmer may be converted with
    // both co- and contra-variant conversions. Because
    // a programmer can turn any food into a computer
    // program, it can be used as a device that turns pizza
    // into documents.
    ITransformer<Pizza, Document> transformer = programmer;
    Document d = transformer.Transform(new Pizza());
  }
}
```

Lastly, note that the compiler will check validity of the covariance and contravariance type parameter modifiers throughout the source. Consider the `PairInitializer<in T>` interface in Listing 11.46.

Listing 11.46. Compiler Validation of Variance

```
// ERROR:  Invalid variance, the type parameter 'T' is not
//         invariantly valid
interface IPairInitializer<in T>
{
  void Initialize(IPair<T> pair);
}
```

```
// Suppose the code above were legal, and see what goes
// wrong:
class FruitPairInitializer : IPairInitializer<Fruit>
{
  // Let's initiaize our pair of fruit with an
  // apple and an orange.
  public void Initialize(IPair<Fruit> pair)
  {
    pair.First = new Orange();
    pair.Second = new Apple();
  }
}
```

```
  // ... later ...
  var f = new FruitPairInitializer();
  // This would be legal if contravariance were legal:
  IPairInitializer<Apple> a = f;
  // And now we write an orange into a pair of apples:
```

```
a.Initialize(new Pair<Apple>());
```

A casual observer may be tempted to think that since `IPair<T>` is used only as an "input" formal parameter, the contravariant `in` modifier on `IPairInitializer` is valid. However, the `IPair<T>` interface cannot safely vary, and therefore, it cannot be constructed with a type argument that can vary. As you can see, this would not be type-safe, and therefore, the compiler disallows the `IPairInitializer<T>` interface from being declared as contravariant in the first place.

Support for Unsafe Covariance in Arrays

So far we have described covariance and contravariance as being properties of generic types. Of all the nongeneric types, arrays are most like generics; just as we think of a generic "list of T" or a generic "pair of T" we can think of an "array of T" as being the same sort of pattern. Since arrays clearly support both reading and writing, given what you know about covariance and contravariance, you probably would suppose that arrays may be neither safely contravariant nor covariant; an array can only be safely covariant if it is never written to, and only safely contravariant if it is never read from; neither seems like a realistic restriction.

Unfortunately, C# does support array covariance, even though doing so is not type-safe. For example, `Fruit[] fruits = new Apple[10];` is perfectly legal in C#, and if you then say `fruits[0] = new Orange();`, the runtime will issue a type safety violation in the form of an exception. It is deeply disturbing that it is not always legal to assign an `Orange` into an array of `Fruit` because it might really be an array of `Apple`s, but that is the situation in not just C#, but all CLR languages that use the runtime's implementation of arrays.

Try to avoid using unsafe array covariance. Every array is convertible to the read-only (and therefore safely covariant) interface `IEnumerable<T>`; that is to say, `IEnumerable<Fruit> fruits = new Apple[10]` is both safe and legal because there is no way to insert an `Orange` into the array if all you have is the read-only interface.

Guidelines

AVOID unsafe array covariance. Instead, **CONSIDER** converting the array to the read-only interface `IEnumerable<T>`, which can be safely converted via covariant conversions.
