Covariance

Assuming `A` is convertible to `B`, `X` is covariant if `X<A>` is convertible to `X<B>`.

> **NOTE**
>
> With C#'s notion of covariance (and contravariance), "convertible" means convertible via an *implicit reference conversion*—such as `A` *subclassing* `B`, or `A` *implementing* `B`. Numeric conversions, boxing conversions, and custom conversions are not included.

For instance, type `IFoo<T>` is covariant for `T` if the following is legal:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

From C# 4.0, generic interfaces permit covariance for (as do generic delegates—see Chapter 4), but generic classes do not. Arrays also support covariance (`A[]` can be converted to `B[]` if `A` has an implicit reference conversion to `B`), and are discussed here for comparison.

> **NOTE**
>
> Covariance and contravariance (or simply "variance") are advanced concepts. The motivation behind introducing and enhancing variance in C# was to allow generic interface and generic types (in particular, those defined in the Framework, such as `IEnumerable<T>`) to work *more as you'd expect*. You can benefit from this without understanding the details behind covariance and contravariance.

Classes

Generic classes are not covariant, to ensure static type safety. Consider the following:

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T>   // A simple Stack implementation
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)   { data[position++] = obj;  }
  public T Pop()             { return data[--position]; }
}
```

The following fails to compile:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;            // Compile-time error
```

That restriction prevents the possibility of runtime failure with the following code:

```
animals.Push (new Camel());      // Trying to add Camel to bears
```

Lack of covariance, however, can hinder reusability. Suppose, for instance, we wanted to write a method to `Wash` a stack of animals:

```
public class ZooCleaner
{
  public static void Wash (Stack<Animal> animals) {...}
}
```

Calling `Wash` with a stack of bears would generate a compile-time error. One workaround is to redefine the `Wash` method with a constraint:

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
}
```

We can now call `Wash` as follows:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Another solution is to have `Stack<T>` implement a covariant generic interface, as we'll see shortly.

Arrays

For historical reasons, array types are covariant. This means that `B[]` can be cast to `A[]` if `B` subclasses `A` (and both are reference types). For example:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;     // OK
```

The downside of this reusability is that element assignments can fail at runtime:

```
animals[0] = new Camel();     // Runtime error
```

Interfaces

As of C# 4.0, generic interfaces support covariance for type parameters marked with the `out` modifier. This modifier ensures that, unlike with arrays, covariance with interfaces is fully type-safe. To illustrate, suppose that our `Stack` class implements the following interface:

```
public interface IPoppable<out T> { T Pop(); }
```

The `out` modifier on `T` indicates that `T` is used only in *output positions* (e.g., return types for methods). The `out` modifier flags the interface as *covariant* and allows us to do this:

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears implements IPoppable<Bear>. We can convert to IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Legal
Animal a = animals.Pop();
```

The cast from `bears` to `animals` is permitted by the compiler—by virtue of the interface being covariant. This is type-safe because the case the compiler is trying to avoid—pushing a `Camel` onto the stack—can't occur as there's no way to feed a `Camel` *in*to an interface where `T` can appear only in *out*put positions.

> **NOTE**
>
> Covariance (and contravariance) in interfaces is something that you typically *consume*: it's less common that you need to *write* variant interfaces. Curiously, method parameters marked as `out` are not eligible for covariance, due to a limitation in the CLR.

We can leverage the ability to cast covariantly to solve the reusability problem described earlier:

```
public class ZooCleaner
{
  public static void Wash (IPoppable<Animal> animals) { ... }
}
```

> **NOTE**
>
> The `IEnumerator<T>` and `IEnumerable<T>` interfaces described in Chapter 7 are marked as covariant. This allows you to cast `IEnumerable<string>` to `IEnumerable<object>`, for instance.

The compiler will generate an error if you use a covariant type parameter in an *input* position (e.g., a parameter to a method or a writable property).

NOTE

With both generic types and arrays, covariance (and contravariance) is valid only for elements with *reference conversions*—not *boxing conversions*. So, if you wrote a method that accepted a parameter of type `IPoppable<object>`, you could call it with `IPoppable<string>`, but not `IPoppable<int>`.