Contravariance

We previously saw that, assuming that `A` allows an implicit reference conversion to `B`, a type `X` is covariant if `X<A>` allows a reference conversion to `X<B>`. A type is *contravariant* when you can convert in the reverse direction—from `X<B>` to `X<A>`. This is supported with generic interfaces—when the generic type parameter only appears in *input* positions, designated with the `in` modifier. Extending our previous example, if the `Stack<T>` class implements the following interface:

```
public interface IPushable<in T> { void Push (T obj); }
```

we can legally do this:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;    // Legal
bears.Push (new Bear());
```

No member in `IPushable` *outputs* a `T`, so we can't get into trouble by casting `animals` to `bears` (there's no way to `Pop`, for instance, through that interface).

> **NOTE**
>
> Our `Stack<T>` class can implement both `IPushable<T>` and `IPoppable<T>`—despite `T` having opposing variance annotations in the two interfaces! This works because you can exercise variance only through an interface; therefore, you must commit to the lens of either `IPoppable` or `IPushable` before performing a variant conversion. This lens then restricts you to the operations that are legal under the appropriate variance rules.
>
> This also illustrates why it would usually make no sense for *classes* (such as `Stack<T>`) to be variant: concrete implementations typically require data to flow in both directions.

To give another example, consider the following interface, defined as part of the .NET Framework:

```
public interface IComparer<in T>
{
  // Returns a value indicating the relative ordering of a and b
  int Compare (T a, T b);
}
```

Because the interface is contravariant, we can use an `IComparer<object>` to compare two *strings*:

```
var objectComparer = Comparer<object>.Default;
// objectComparer implements IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Brett", "Jemaine");
```

Mirroring covariance, the compiler will report an error if you try to use a contravariant parameter in an output position (e.g., as a return value, or in a readable property).