**Covariance and Contravariance in Generics**

**.NET Framework 4.5**

[Other Versions](javascript:;)

http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [.NET Framework 4](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.100).aspx)

Covariance and contravariance are terms that refer to the ability to use a less derived or more derived type than originally specified. Generic type parameters support covariance and contravariance to provide greater flexibility in assigning and using generic types. When you are referring to a type system, covariance, contravariance, and invariance have the following definitions. The examples assume a base class named Base and a derived class named Derived.

* Covariance

Enables you to use a more specific type than originally specified.

You can assign an instance of IEnumerable<Derived> (IEnumerable(Of Derived) in Visual Basic) to a variable of type IEnumerable<Base>.

* Contravariance

Enables you to use a more generic (less derived) type than originally specified.

You can assign an instance of IEnumerable<Base> (IEnumerable(Of Base) in Visual Basic) to a variable of type IEnumerable<Derived>.

* Invariance

Means that you can use only the type originally specified; so an invariant generic type parameter is neither covariant nor contravariant.

You cannot assign an instance of IEnumerable<Base> (IEnumerable(Of Base) in Visual Basic) to a variable of type IEnumerable<Derived> or vice versa.

Covariant type parameters enable you to make assignments that look much like ordinary polymorphism, as shown in the following code.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

IEnumerable<Derived> d = new List<Derived>();

IEnumerable<Base> b = d;

The [List<T>](http://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx) class implements the [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx) interface, so List<Derived> (List(Of Derived) in Visual Basic) implements IEnumerable<Derived>. The covariant type parameter does the rest.

Contravariance, on the other hand, seems counterintuitive. The following example creates a delegate of type Action<Base> (Action(Of Base) in Visual Basic), and then assigns that delegate to a variable of type Action<Derived>.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };

Action<Derived> d = b;

d(new Derived());

This seems backward, but it is type-safe code that compiles and runs. The lambda expression matches the delegate it is assigned to, so it defines a method that takes one parameter of type Base and that has no return value. The resulting delegate can be assigned to a variable of type Action<Derived> because the type parameter T of the [Action<T>](http://msdn.microsoft.com/en-us/library/018hxwa8(v=vs.110).aspx) delegate is contravariant. The code is type-safe because T specifies a parameter type. When the delegate of type Action<Base> is invoked as if it were a delegate of type Action<Derived>, its argument must be of type Derived. This argument can always be passed safely to the underlying method, because the method's parameter is of type Base.

In general, a covariant type parameter can be used as the return type of a delegate, and contravariant type parameters can be used as parameter types. For an interface, covariant type parameters can be used as the return types of the interface's methods, and contravariant type parameters can be used as the parameter types of the interface's methods.

Covariance and contravariance are collectively referred to as variance. A generic type parameter that is not marked covariant or contravariant is referred to as invariant. A brief summary of facts about variance in the common language runtime:

* In the .NET Framework 4, variant type parameters are restricted to generic interface and generic delegate types.
* A generic interface or generic delegate type can have both covariant and contravariant type parameters.
* Variance applies only to reference types; if you specify a value type for a variant type parameter, that type parameter is invariant for the resulting constructed type.
* Variance does not apply to delegate combination. That is, given two delegates of types Action<Derived> and Action<Base> (Action(Of Derived) and Action(Of Base) in Visual Basic), you cannot combine the second delegate with the first although the result would be type safe. Variance allows the second delegate to be assigned to a variable of type Action<Derived>, but delegates can combine only if their types match exactly.

The following subsections describe covariant and contravariant type parameters in detail:

* [Generic Interfaces with Covariant Type Parameters](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx#InterfaceCovariantTypeParameters)
* [Generic Interfaces with Contravariant Generic Type Parameters](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx#InterfaceCovariantTypeParameters)
* [Generic Delegates with Variant Type Parameters](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx#DelegateVariantTypeParameters)
* [Defining Variant Generic Interfaces and Delegates](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx#DefiningVariantTypeParameters)
* [List of Variant Generic Interface and Delegate Types](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx#VariantList)

[Generic Interfaces with Covariant Type Parameters](javascript:void(0))

Starting with the .NET Framework 4, several generic interfaces have covariant type parameters; for example: [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx), [IEnumerator<T>](http://msdn.microsoft.com/en-us/library/78dfe2yb(v=vs.110).aspx), [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562(v=vs.110).aspx), and [IGrouping<TKey, TElement>](http://msdn.microsoft.com/en-us/library/bb344977(v=vs.110).aspx). All the type parameters of these interfaces are covariant, so the type parameters are used only for the return types of the members.

The following example illustrates covariant type parameters. The example defines two types: Base has a static method named PrintBases that takes an IEnumerable<Base> (IEnumerable(Of Base) in Visual Basic) and prints the elements. Derived inherits from Base. The example creates an empty List<Derived> (List(Of Derived) in Visual Basic) and demonstrates that this type can be passed to PrintBases and assigned to a variable of type IEnumerable<Base> without casting. [List<T>](http://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx) implements [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx), which has a single covariant type parameter. The covariant type parameter is the reason why an instance of IEnumerable<Derived> can be used instead of IEnumerable<Base>.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-3)

using System;

using System.Collections.Generic;

class Base

{

public static void PrintBases(IEnumerable<Base> bases)

{

foreach(Base b in bases)

{

Console.WriteLine(b);

}

}

}

class Derived : Base

{

public static void Main()

{

List<Derived> dlist = new List<Derived>();

Derived.PrintBases(dlist);

IEnumerable<Base> bIEnum = dlist;

}

}

[Generic Interfaces with Contravariant Generic Type Parameters](javascript:void(0))

Starting with the .NET Framework 4, several generic interfaces have contravariant type parameters; for example: [IComparer<T>](http://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx), [IComparable<T>](http://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx), and [IEqualityComparer<T>](http://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx). These interfaces have only contravariant type parameters, so the type parameters are used only as parameter types in the members of the interfaces.

The following example illustrates contravariant type parameters. The example defines an abstract (MustInherit in Visual Basic) Shape class with an Area property. The example also defines a ShapeAreaComparer class that implements IComparer<Shape> (IComparer(Of Shape) in Visual Basic). The implementation of the [IComparer<T>.Compare](http://msdn.microsoft.com/en-us/library/xh5ks3b3(v=vs.110).aspx) method is based on the value of the Area property, so ShapeAreaComparer can be used to sort Shape objects by area.

The Circle class inherits Shape and overrides Area. The example creates a [SortedSet<T>](http://msdn.microsoft.com/en-us/library/dd412070(v=vs.110).aspx) of Circle objects, using a constructor that takes an IComparer<Circle> (IComparer(Of Circle) in Visual Basic). However, instead of passing an IComparer<Circle>, the example passes a ShapeAreaComparer object, which implements IComparer<Shape>. The example can pass a comparer of a less derived type (Shape) when the code calls for a comparer of a more derived type (Circle), because the type parameter of the [IComparer<T>](http://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx) generic interface is contravariant.

When a new Circle object is added to the SortedSet<Circle>, the IComparer<Shape>.Compare method (IComparer(Of Shape).Compare method in Visual Basic) of the ShapeAreaComparer object is called each time the new element is compared to an existing element. The parameter type of the method (Shape) is less derived than the type that is being passed (Circle), so the call is type safe. Contravariance enables ShapeAreaComparer to sort a collection of any single type, as well as a mixed collection of types, that derive from Shape.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-4)

using System;

using System.Collections.Generic;

abstract class Shape

{

public virtual double Area { get { return 0; }}

}

class Circle : Shape

{

private double r;

public Circle(double radius) { r = radius; }

public double Radius { get { return r; }}

public override double Area { get { return Math.PI \* r \* r; }}

}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>

{

int IComparer<Shape>.Compare(Shape a, Shape b)

{

if (a == null) return b == null ? 0 : -1;

return b == null ? 1 : a.Area.CompareTo(b.Area);

}

}

class Program

{

static void Main()

{

// You can pass ShapeAreaComparer, which implements IComparer<Shape>,

// even though the constructor for SortedSet<Circle> expects

// IComparer<Circle>, because type parameter T of IComparer<T> is

// contravariant.

SortedSet<Circle> circlesByArea =

new SortedSet<Circle>(new ShapeAreaComparer())

{ new Circle(7.2), new Circle(100), null, new Circle(.01) };

foreach (Circle c in circlesByArea)

{

Console.WriteLine(c == null ? "null" : "Circle with area " + c.Area);

}

}

}

/\* This code example produces the following output:

null

Circle with area 0.000314159265358979

Circle with area 162.860163162095

Circle with area 31415.9265358979

\*/

[Generic Delegates with Variant Type Parameters](javascript:void(0))

In the .NET Framework 4, the Func generic delegates, such as [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx), have covariant return types and contravariant parameter types. The Action generic delegates, such as [Action<T1, T2>](http://msdn.microsoft.com/en-us/library/bb549311(v=vs.110).aspx), have contravariant parameter types. This means that the delegates can be assigned to variables that have more derived parameter types and (in the case of the Func generic delegates) less derived return types.

|  |
| --- |
| **NoteNote** |
| The last generic type parameter of the Func generic delegates specifies the type of the return value in the delegate signature. It is covariant (out keyword), whereas the other generic type parameters are contravariant (in keyword). |

The following code illustrates this. The first piece of code defines a class named Base, a class named Derived that inherits Base, and another class with a static method (Shared in Visual Basic) named MyMethod. The method takes an instance of Base and returns an instance of Derived. (If the argument is an instance of Derived, MyMethod returns it; if the argument is an instance of Base, MyMethod returns a new instance of Derived.) In Main(), the example creates an instance of Func<Base, Derived> (Func(Of Base, Derived) in Visual Basic) that represents MyMethod, and stores it in the variable f1.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-5)

public class Base {}

public class Derived : Base {}

public class Program

{

public static Derived MyMethod(Base b)

{

return b as Derived ?? new Derived();

}

static void Main()

{

Func<Base, Derived> f1 = MyMethod;

The second piece of code shows that the delegate can be assigned to a variable of type Func<Base, Base> (Func(Of Base, Base) in Visual Basic), because the return type is covariant.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-6)

// Covariant return type.

Func<Base, Base> f2 = f1;

Base b2 = f2(new Base());

The third piece of code shows that the delegate can be assigned to a variable of type Func<Derived, Derived> (Func(Of Derived, Derived) in Visual Basic), because the parameter type is contravariant.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-7)

// Contravariant parameter type.

Func<Derived, Derived> f3 = f1;

Derived d3 = f3(new Derived());

The final piece of code shows that the delegate can be assigned to a variable of type Func<Derived, Base> (Func(Of Derived, Base) in Visual Basic), combining the effects of the contravariant parameter type and the covariant return type.

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-8)

// Covariant return type and contravariant parameter type.

Func<Derived, Base> f4 = f1;

Base b4 = f4(new Derived());

[Variance in Generic and Non-Generic Delegates](javascript:void(0))

In the preceding code, the signature of MyMethod exactly matches the signature of the constructed generic delegate: Func<Base, Derived> (Func(Of Base, Derived) in Visual Basic). The example shows that this generic delegate can be stored in variables or method parameters that have more derived parameter types and less derived return types, as long as all the delegate types are constructed from the generic delegate type [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx).

This is an important point. The effects of covariance and contravariance in the type parameters of generic delegates are similar to the effects of covariance and contravariance in ordinary delegate binding (see [Variance in Delegates (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd233060(v=vs.110).aspx)). However, variance in delegate binding works with all delegate types, not just with generic delegate types that have variant type parameters. Furthermore, variance in delegate binding enables a method to be bound to any delegate that has more restrictive parameter types and a less restrictive return type, whereas the assignment of generic delegates works only if both delegate types are constructed from the same generic type definition.

The following example shows the combined effects of variance in delegate binding and variance in generic type parameters. The example defines a type hierarchy that includes three types, from least derived (Type1) to most derived (Type3). Variance in ordinary delegate binding is used to bind a method with a parameter type of Type1 and a return type of Type3 to a generic delegate with a parameter type of Type2 and a return type of Type2. The resulting generic delegate is then assigned to another variable whose generic delegate type has a parameter of type Type3 and a return type of Type1, using the covariance and contravariance of generic type parameters. The second assignment requires both the variable type and the delegate type to be constructed from the same generic type definition, in this case, [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx).

C#

[VB](http://msdn.microsoft.com/en-us/library/dd799517(d=printer,v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-9)

using System;

public class Type1 {}

public class Type2 : Type1 {}

public class Type3 : Type2 {}

public class Program

{

public static Type3 MyMethod(Type1 t)

{

return t as Type3 ?? new Type3();

}

static void Main()

{

Func<Type2, Type2> f1 = MyMethod;

// Covariant return type and contravariant parameter type.

Func<Type3, Type1> f2 = f1;

Type1 t1 = f2(new Type3());

}

}

[Defining Variant Generic Interfaces and Delegates](javascript:void(0))

Starting with the .NET Framework 4, Visual Basic and C# have keywords that enable you to mark the generic type parameters of interfaces and delegates as covariant or contravariant.

|  |
| --- |
| **NoteNote** |
| Starting with the .NET Framework version 2.0, the common language runtime supports variance annotations on generic type parameters. Prior to the .NET Framework 4, the only way to define a generic class that has these annotations is to use Microsoft intermediate language (MSIL), either by compiling the class with [Ilasm.exe (MSIL Assembler)](http://msdn.microsoft.com/en-us/library/496e4ekx(v=vs.110).aspx) or by emitting it in a dynamic assembly. |

A covariant type parameter is marked with the out keyword (Out keyword in Visual Basic, + for the [MSIL Assembler](http://msdn.microsoft.com/en-us/library/496e4ekx(v=vs.110).aspx)). You can use a covariant type parameter as the return value of a method that belongs to an interface, or as the return type of a delegate. You cannot use a covariant type parameter as a generic type constraint for interface methods.

|  |
| --- |
| **NoteNote** |
| If a method of an interface has a parameter that is a generic delegate type, a covariant type parameter of the interface type can be used to specify a contravariant type parameter of the delegate type. |

A contravariant type parameter is marked with the in keyword (In keyword in Visual Basic, - for the [MSIL Assembler](http://msdn.microsoft.com/en-us/library/496e4ekx(v=vs.110).aspx)). You can use a contravariant type parameter as the type of a parameter of a method that belongs to an interface, or as the type of a parameter of a delegate. You can use a contravariant type parameter as a generic type constraint for an interface method.

Only interface types and delegate types can have variant type parameters. An interface or delegate type can have both covariant and contravariant type parameters.

Visual Basic and C# do not allow you to violate the rules for using covariant and contravariant type parameters, or to add covariance and contravariance annotations to the type parameters of types other than interfaces and delegates. The [MSIL Assembler](http://msdn.microsoft.com/en-us/library/496e4ekx(v=vs.110).aspx) does not perform such checks, but a [TypeLoadException](http://msdn.microsoft.com/en-us/library/system.typeloadexception(v=vs.110).aspx) is thrown if you try to load a type that violates the rules.

For information and example code, see [Variance in Generic Interfaces (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd233059(v=vs.110).aspx).

[List of Variant Generic Interface and Delegate Types](javascript:void(0))

In the .NET Framework 4, the following interface and delegate types have covariant and/or contravariant type parameters.

|  |  |  |
| --- | --- | --- |
| **Type** | **Covariant type parameters** | **Contravariant type parameters** |
| [Action<T>](http://msdn.microsoft.com/en-us/library/018hxwa8(v=vs.110).aspx) to [Action<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16>](http://msdn.microsoft.com/en-us/library/dd402872(v=vs.110).aspx) |  | Yes |
| [Comparison<T>](http://msdn.microsoft.com/en-us/library/tfakywbh(v=vs.110).aspx) |  | Yes |
| [Converter<TInput, TOutput>](http://msdn.microsoft.com/en-us/library/kt456a2y(v=vs.110).aspx) | Yes | Yes |
| [Func<TResult>](http://msdn.microsoft.com/en-us/library/bb534960(v=vs.110).aspx) | Yes |  |
| [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx) to [Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult>](http://msdn.microsoft.com/en-us/library/dd402862(v=vs.110).aspx) | Yes | Yes |
| [IComparable<T>](http://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx) |  | Yes |
| [Predicate<T>](http://msdn.microsoft.com/en-us/library/bfcke1bz(v=vs.110).aspx) |  | Yes |
| [IComparer<T>](http://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx) |  | Yes |
| [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx) | Yes |  |
| [IEnumerator<T>](http://msdn.microsoft.com/en-us/library/78dfe2yb(v=vs.110).aspx) | Yes |  |
| [IEqualityComparer<T>](http://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx) |  | Yes |
| [IGrouping<TKey, TElement>](http://msdn.microsoft.com/en-us/library/bb344977(v=vs.110).aspx) | Yes |  |
| [IOrderedEnumerable<TElement>](http://msdn.microsoft.com/en-us/library/bb534852(v=vs.110).aspx) | Yes |  |
| [IOrderedQueryable<T>](http://msdn.microsoft.com/en-us/library/bb340178(v=vs.110).aspx) | Yes |  |
| [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562(v=vs.110).aspx) | Yes |  |