Visual Studio 2010

**Creating Variant Generic Interfaces (C# and Visual Basic)**

You can declare generic type parameters in interfaces as covariant or contravariant. *Covariance* allows interface methods to have more derived return types than that defined by the generic type parameters. *Contravariance* allows interface methods to have argument types that are less derived than that specified by the generic parameters. A generic interface that has covariant or contravariant generic type parameters is called *variant*.

|  |
| --- |
| **Note** |
| .NET Framework 4 introduces variance support for several existing generic interfaces. For the list of the variant interfaces in the .NET Framework, see [Variance in Generic Interfaces (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd233059.aspx). |

Declaring Variant Generic Interfaces

You can declare variant generic interfaces by using the **in** and **out** keywords for generic type parameters.

|  |
| --- |
| **Important** |
| **ByRef** parameters in Visual Basic and **ref** and **out** parameters in C# cannot be variant. Value types also do not support variance. |

You can declare a generic type parameter covariant by using the **out** keyword. The covariant type must satisfy the following conditions:

* The type is used only as a return type of interface methods and not used as a type of method arguments. This is illustrated in the following example, in which the type R is declared covariant.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl01_code');)

Interface ICovariant(Of Out R)

Function GetSomething() As R

' The following statement generates a compiler error.

' Sub SetSomething(ByVal sampleArg As R)

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl02_code');)

interface ICovariant<out R>

{

R GetSomething();

// The following statement generates a compiler error.

// void SetSometing(R sampleArg);

}

There is one exception to this rule. If you have a contravariant generic delegate as a method parameter, you can use the type as a generic type parameter for the delegate. This is illustrated by the type R in the following example. For more information, see [Variance in Delegates (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd233060.aspx) and [Using Variance for Func and Action Generic Delegates (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/dd465122.aspx).

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl05_code');)

Interface ICovariant(Of Out R)

Sub DoSomething(ByVal callback As Action(Of R))

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl06_code');)

interface ICovariant<out R>

{

void DoSomething(Action<R> callback);

}

* The type is not used as a generic constraint for the interface methods. This is illustrated in the following code.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl07_code');)

Interface ICovariant(Of Out R)

' The following statement generates a compiler error

' because you can use only contravariant or invariant types

' in generic contstraints.

' Sub DoSomething(Of T As R)()

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl08_code');)

interface ICovariant<out R>

{

// The following statement generates a compiler error

// because you can use only contravariant or invariant types

// in generic contstraints.

// void DoSomething<T>() where T : R;

}

You can declare a generic type parameter contravariant by using the **in** keyword. The contravariant type can be used only as a type of method arguments and not as a return type of interface methods. The contravariant type can also be used for generic constraints. The following code shows how to declare a contravariant interface and use a generic constraint for one of its methods.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl09_code');)

Interface IContravariant(Of In A)

Sub SetSomething(ByVal sampleArg As A)

Sub DoSomething(Of T As A)()

' The following statement generates a compiler error.

' Function GetSomething() As A

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl10_code');)

interface IContravariant<in A>

{

void SetSomething(A sampleArg);

void DoSomething<T>() where T : A;

// The following statement generates a compiler error.

// A GetSomething();

}

It is also possible to support both covariance and contravariance in the same interface, but for different type parameters, as shown in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl11_code');)

Interface IVariant(Of Out R, In A)

Function GetSomething() As R

Sub SetSomething(ByVal sampleArg As A)

Function GetSetSomething(ByVal sampleArg As A) As R

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl12_code');)

interface IVariant<out R, in A>

{

R GetSomething();

void SetSomething(A sampleArg);

R GetSetSometings(A sampleArg);

}

In Visual Basic, you cannot declare events in variant interfaces without specifying the delegate type. Also, a variant interface cannot have nested classes, enums, or structures, but it can have nested interfaces. This is illustrated in the following code.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl13_code');)

Interface ICovariant(Of Out R)

' The following statement generates a compiler error.

' Event SampleEvent()

' The following statement specifies the delegate type and

' does not generate an error.

Event AnotherEvent As EventHandler

' The following statements generate compiler errors,

' because a variant interface cannot have

' nested enums, classes, or structures.

'Enum SampleEnum : test : End Enum

'Class SampleClass : End Class

'Structure SampleStructure : Dim value As Integer : End Structure

' Variant interfaces can have nested interfaces.

Interface INested : End Interface

End Interface

Implementing Variant Generic Interfaces

You implement variant generic interfaces in classes by using the same syntax that is used for invariant interfaces. The following code example shows how to implement a covariant interface in a generic class.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl00_code');)

Interface ICovariant(Of Out R)

Function GetSomething() As R

End Interface

Class SampleImplementation(Of R)

Implements ICovariant(Of R)

Public Function GetSomething() As R \_

Implements ICovariant(Of R).GetSomething

' Some code.

End Function

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl01_code');)

interface ICovariant<out R>

{

R GetSomething();

}

class SampleImplementation<R> : ICovariant<R>

{

public R GetSomething()

{

// Some code.

return default(R);

}

}

Classes that implement variant interfaces are invariant. For example, consider the following code.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl02_code');)

' The interface is covariant.

Dim ibutton As ICovariant(Of Button) =

New SampleImplementation(Of Button)

Dim iobj As ICovariant(Of Object) = ibutton

' The class is invariant.

Dim button As SampleImplementation(Of Button) =

New SampleImplementation(Of Button)

' The following statement generates a compiler error

' because classes are invariant.

' Dim obj As SampleImplementation(Of Object) = button

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl03_code');)

// The interface is covariant.

ICovariant<Button> ibutton = new SampleImplementation<Button>();

ICovariant<Object> iobj = ibutton;

// The class is invariant.

SampleImplementation<Button> button = new SampleImplementation<Button>();

// The following statement generates a compiler error

// because classes are invariant.

// SampleImplementation<Object> obj = button;

Extending Variant Generic Interfaces

When you extend a variant generic interface, you have to use the **in** and **out** keywords to explicitly specify whether the derived interface supports variance. The compiler does not infer the variance from the interface that is being extended. For example, consider the following interfaces.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl00_code');)

Interface ICovariant(Of Out T)

End Interface

Interface IInvariant(Of T)

Inherits ICovariant(Of T)

End Interface

Interface IExtCovariant(Of Out T)

Inherits ICovariant(Of T)

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl01_code');)

interface ICovariant<out T> { }

interface IInvariant<T> : ICovariant<T> { }

interface IExtCovariant<out T> : ICovariant<T> { }

In the IInvariant<T> (Invariant(Of T) in Visual Basic) interface, the generic type parameter T is invariant, whereas in IExtCovariant<out T> (IExtCovariant (Of Out T) in Visual Basic) the type parameter is covariant, although both interfaces extend the same interface. The same rule is applied to contravariant generic type parameters.

You can create an interface that extends both the interface where the generic type parameter T is covariant and the interface where it is contravariant if in the extending interface the generic type parameter T is invariant. This is illustrated in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl02_code');)

Interface ICovariant(Of Out T)

End Interface

Interface IContravariant(Of In T)

End Interface

Interface IInvariant(Of T)

Inherits ICovariant(Of T), IContravariant(Of T)

End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl03_code');)

interface ICovariant<out T> { }

interface IContravariant<in T> { }

interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }

However, if a generic type parameter T is declared covariant in one interface, you cannot declare it contravariant in the extending interface, or vice versa. This is illustrated in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl04_code');)

Interface ICovariant(Of Out T)

End Interface

' The following statements generate a compiler error.

' Interface ICoContraVariant(Of In T)

' Inherits ICovariant(Of T)

' End Interface

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl05_code');)

interface ICovariant<out T> { }

// The following statement generates a compiler error.

// interface ICoContraVariant<in T> : ICovariant<T> { }

**Avoiding Ambiguity**

When you implement variant generic interfaces, variance can sometimes lead to ambiguity. This should be avoided.

For example, if you explicitly implement the same variant generic interface with different generic type parameters in one class, it can create ambiguity. The compiler does not produce an error in this case, but it is not specified which interface implementation will be chosen at runtime. This could lead to subtle bugs in your code. Consider the following code example.

|  |
| --- |
| **Note** |
| With **Option Strict Off**, Visual Basic generates a compiler warning when there is an ambiguous interface implementation. With **Option Strict On**, Visual Basic generates a compiler error. |

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl07_code');)

' Simple class hierarchy.

Class Animal

End Class

Class Cat

Inherits Animal

End Class

Class Dog

Inherits Animal

End Class

' This class introduces ambiguity

' because IEnumerable(Of Out T) is covariant.

Class Pets

Implements IEnumerable(Of Cat), IEnumerable(Of Dog)

Public Function GetEnumerator() As IEnumerator(Of Cat) \_

Implements IEnumerable(Of Cat).GetEnumerator

Console.WriteLine("Cat")

' Some code.

End Function

Public Function GetEnumerator1() As IEnumerator(Of Dog) \_

Implements IEnumerable(Of Dog).GetEnumerator

Console.WriteLine("Dog")

' Some code.

End Function

Public Function GetEnumerator2() As IEnumerator \_

Implements IEnumerable.GetEnumerator

' Some code.

End Function

End Class

Sub Main()

Dim pets As IEnumerable(Of Animal) = New Pets()

pets.GetEnumerator()

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl08_code');)

// Simple class hierarchy.

class Animal { }

class Cat : Animal { }

class Dog : Animal { }

// This class introduces ambiguity

// because IEnumerable<out T> is covariant.

class Pets : IEnumerable<Cat>, IEnumerable<Dog>

{

IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()

{

Console.WriteLine("Cat");

// Some code.

return null;

}

IEnumerator IEnumerable.GetEnumerator()

{

// Some code.

return null;

}

IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()

{

Console.WriteLine("Dog");

// Some code.

return null;

}

}

class Program

{

public static void Test()

{

IEnumerable<Animal> pets = new Pets();

pets.GetEnumerator();

}

}

In this example, it is unspecified how the pets.GetEnumerator method chooses between Cat and Dog. This could cause problems in your code.