Covariance and Contravariance (Visual Basic)

Visual Studio 2015

In Visual Basic, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

VB

```
' Assignment compatibility.
Dim str As String = "test"
' An object of a more derived type is assigned to an object of a less derived type.
Dim obj As Object = str

' Covariance.
Dim strings As IEnumerable(Of String) = New List(Of String)()
' An object that is instantiated with a more derived type argument
' is assigned to an object instantiated with a less derived type argument.
' Assignment compatibility is preserved.
Dim objects As IEnumerable(Of Object) = strings

' Contravariance.
' Assume that there is the following method in the class:
' Shared Sub SetObject(ByVal o As Object)
' End Sub
Dim actObject As Action(Of Object) = AddressOf SetObject

' An object that is instantiated with a less derived type argument
' is assigned to an object instantiated with a more derived type argument.
' Assignment compatibility is reversed.
Dim actString As Action(Of String) = actObject
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

VB

```
Dim array() As Object = New String(10) {}
' The following statement produces a run-time exception.
' array(0) = 10
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see Variance in Delegates (Visual Basic) and Using Variance in Delegates (Visual Basic).

The following code example shows covariance and contravariance support for method groups.

VB

```vb
Shared Function GetObject() As Object
    Return Nothing
End Function

Shared Sub SetObject(ByVal obj As Object)
End Sub

Shared Function GetString() As String
    Return ""
End Function

Shared Sub SetString(ByVal str As String)

End Sub

Shared Sub Test()
    ' Covariance. A delegate specifies a return type as object,
    ' but you can assign a method that returns a string.
    Dim del As Func(Of Object) = AddressOf GetString

    ' Contravariance. A delegate specifies a parameter type as string,
    ' but you can assign a method that takes an object.
    Dim del2 As Action(Of String) = AddressOf SetObject
End Sub
```

In .NET Framework 4 or later Visual Basic support covariance and contravariance in generic interfaces and delegates and allow for implicit conversion of generic type parameters. For more information, see Variance in Generic Interfaces (Visual Basic) and Variance in Delegates (Visual Basic).

The following code example shows implicit reference conversion for generic interfaces.

VB

```vb
Dim strings As IEnumerable(Of String) = New List(Of String)
Dim objects As IEnumerable(Of Object) = strings
```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. Visual Basic enables you to create your own variant interfaces and delegates. For more information, see Creating Variant Generic Interfaces (Visual Basic) and Variance in Delegates (Visual Basic).

Related Topics

Title	**Description**

Variance in Generic Interfaces (Visual Basic)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
Creating Variant Generic Interfaces (Visual Basic)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (Visual Basic)	Shows how covariance and contravariance support in the IEnumerable(Of T) and IComparable(Of T) interfaces can help you reuse code.
Variance in Delegates (Visual Basic)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.
Using Variance in Delegates (Visual Basic)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (Visual Basic)	Shows how covariance and contravariance support in the **Func** and **Action** delegates can help you reuse code.

Variance in Generic Interfaces (Visual Basic)

Visual Studio 2015

.NET Framework 4 introduced variance support for several existing generic interfaces. Variance support enables implicit conversion of classes that implement these interfaces. The following interfaces are now variant:

- IEnumerable(Of T) (T is covariant)
- IEnumerator(Of T) (T is covariant)
- IQueryable(Of T) (T is covariant)
- IGrouping(Of TKey, TElement) (*TKey* and *TElement* are covariant)
- IComparer(Of T) (T is contravariant)
- IEqualityComparer(Of T) (T is contravariant)
- IComparable(Of T) (T is contravariant)

Covariance permits a method to have a more derived return type than that defined by the generic type parameter of the interface. To illustrate the covariance feature, consider these generic interfaces: `IEnumerable(Of Object)` and `IEnumerable(Of String)`. The `IEnumerable(Of String)` interface does not inherit the `IEnumerable(Of Object)` interface. However, the `String` type does inherit the `Object` type, and in some cases you may want to assign objects of these interfaces to each other. This is shown in the following code example.

VB

```
Dim strings As IEnumerable(Of String) = New List(Of String)
Dim objects As IEnumerable(Of Object) = strings
```

In earlier versions of the .NET Framework, this code causes a compilation error in Visual Basic with `Option Strict On`. But now you can use `strings` instead of `objects`, as shown in the previous example, because the IEnumerable(Of T) interface is covariant.

Contravariance permits a method to have argument types that are less derived than that specified by the generic parameter of the interface. To illustrate contravariance, assume that you have created a `BaseComparer` class to compare instances of the `BaseClass` class. The `BaseComparer` class implements the `IEqualityComparer(Of BaseClass)` interface. Because the IEqualityComparer(Of T) interface is now contravariant, you can use `BaseComparer` to compare instances of classes that inherit the `BaseClass` class. This is shown in the following code example.

VB

```
' Simple hierarchy of classes.
Class BaseClass
End Class
```

```
Class DerivedClass
    Inherits BaseClass
End Class

' Comparer class.
Class BaseComparer
    Implements IEqualityComparer(Of BaseClass)

    Public Function Equals1(ByVal x As BaseClass,
                            ByVal y As BaseClass) As Boolean _
                            Implements IEqualityComparer(Of BaseClass).Equals
        Return (x.Equals(y))
    End Function

    Public Function GetHashCode1(ByVal obj As BaseClass) As Integer _
        Implements IEqualityComparer(Of BaseClass).GetHashCode
        Return obj.GetHashCode
    End Function
End Class
Sub Test()
    Dim baseComparer As IEqualityComparer(Of BaseClass) = New BaseComparer
    ' Implicit conversion of IEqualityComparer(Of BaseClass) to
    ' IEqualityComparer(Of DerivedClass).
    Dim childComparer As IEqualityComparer(Of DerivedClass) = baseComparer
End Sub
```

For more examples, see Using Variance in Interfaces for Generic Collections (Visual Basic).

Variance in generic interfaces is supported for reference types only. Value types do not support variance. For example, `IEnumerable(Of Integer)` cannot be implicitly converted to `IEnumerable(Of Object)`, because integers are represented by a value type.

VB

```
Dim integers As IEnumerable(Of Integer) = New List(Of Integer)
' The following statement generates a compiler error
' with Option Strict On, because Integer is a value type.
' Dim objects As IEnumerable(Of Object) = integers
```

It is also important to remember that classes that implement variant interfaces are still invariant. For example, although List(Of T) implements the covariant interface IEnumerable(Of T), you cannot implicitly convert `List(Of Object)` to `List(Of String)`. This is illustrated in the following code example.

VB

```
' The following statement generates a compiler error
' because classes are invariant.
' Dim list As List(Of Object) = New List(Of String)

' You can use the interface object instead.
Dim listObjects As IEnumerable(Of Object) = New List(Of String)
```

See Also

- Using Variance in Interfaces for Generic Collections (Visual Basic)
- Creating Variant Generic Interfaces (Visual Basic)
- Generic Interfaces
- Variance in Delegates (Visual Basic)

Creating Variant Generic Interfaces (Visual Basic)

Visual Studio 2015

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 introduced variance support for several existing generic interfaces. For the list of the variant interfaces in the .NET Framework, see Variance in Generic Interfaces (Visual Basic).

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 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.

VB

```
Interface ICovariant(Of Out R)
    Function GetSomething() As R
    ' The following statement generates a compiler error.
    ' Sub SetSomething(ByVal sampleArg As R)
End Interface
```

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 (Visual Basic) and Using Variance for Func and Action Generic

Delegates (Visual Basic).

VB

```
Interface ICovariant(Of Out R)
    Sub DoSomething(ByVal callback As Action(Of R))
End Interface
```

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

VB

```
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
```

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.

VB

```
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
```

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.

VB

```
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
```

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

VB

```
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.

VB

```
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
```

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

VB

```
 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
```

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.

VB

```
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
```

In the `Invariant(Of T)` interface, the generic type parameter `T` is invariant, whereas in `IExtCovariant (Of Out T)`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.

VB

```
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
```

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.

VB

```
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
```

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.

VB

```
' 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
```

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

See Also

Variance in Generic Interfaces (Visual Basic)
Using Variance for Func and Action Generic Delegates (Visual Basic)

Using Variance in Interfaces for Generic Collections (Visual Basic)

Visual Studio 2015

A covariant interface allows its methods to return more derived types than those specified in the interface. A contravariant interface allows its methods to accept parameters of less derived types than those specified in the interface.

In .NET Framework 4, several existing interfaces became covariant and contravariant. These include IEnumerable(Of T) and IComparable(Of T). This enables you to reuse methods that operate with generic collections of base types for collections of derived types.

For a list of variant interfaces in the .NET Framework, see Variance in Generic Interfaces (Visual Basic).

Converting Generic Collections

The following example illustrates the benefits of covariance support in the IEnumerable(Of T) interface. The `PrintFullName` method accepts a collection of the `IEnumerable(Of Person)` type as a parameter. However, you can reuse it for a collection of the `IEnumerable(Of Person)` type because `Employee` inherits `Person`.

VB

```
' Simple hierarchy of classes.
Public Class Person
    Public Property FirstName As String
    Public Property LastName As String
End Class

Public Class Employee
    Inherits Person
End Class

' The method has a parameter of the IEnumerable(Of Person) type.
Public Sub PrintFullName(ByVal persons As IEnumerable(Of Person))
    For Each person As Person In persons
        Console.WriteLine(
            "Name: " & person.FirstName & " " & person.LastName)
    Next
End Sub

Sub Main()
    Dim employees As IEnumerable(Of Employee) = New List(Of Employee)

    ' You can pass IEnumerable(Of Employee),
    ' although the method expects IEnumerable(Of Person).

    PrintFullName(employees)
    End Sub
```

Comparing Generic Collections

The following example illustrates the benefits of contravariance support in the IComparer(Of T) interface. The `PersonComparer` class implements the `IComparer(Of Person)` interface. However, you can reuse this class to compare a sequence of objects of the `Employee` type because `Employee` inherits `Person`.

VB

```
    ' Simple hierarhcy of classes.
    Public Class Person
        Public Property FirstName As String
        Public Property LastName As String
    End Class

    Public Class Employee
        Inherits Person
    End Class
    ' The custom comparer for the Person type
    ' with standard implementations of Equals()
    ' and GetHashCode() methods.
    Class PersonComparer
        Implements IEqualityComparer(Of Person)

        Public Function Equals1(
            ByVal x As Person,
            ByVal y As Person) As Boolean _
            Implements IEqualityComparer(Of Person).Equals

            If x Is y Then Return True
            If x Is Nothing OrElse y Is Nothing Then Return False
            Return (x.FirstName = y.FirstName) AndAlso
                (x.LastName = y.LastName)
        End Function
        Public Function GetHashCode1(
            ByVal person As Person) As Integer _
            Implements IEqualityComparer(Of Person).GetHashCode

            If person Is Nothing Then Return 0
            Dim hashFirstName =
                If(person.FirstName Is Nothing,
                0, person.FirstName.GetHashCode())
            Dim hashLastName = person.LastName.GetHashCode()
            Return hashFirstName Xor hashLastName
        End Function
    End Class

    Sub Main()
        Dim employees = New List(Of Employee) From {
            New Employee With {.FirstName = "Michael", .LastName = "Alexander"},
            New Employee With {.FirstName = "Jeff", .LastName = "Price"}
        }

        ' You can pass PersonComparer,
        ' which implements IEqualityComparer(Of Person),
        ' although the method expects IEqualityComparer(Of Employee)

        Dim noduplicates As IEnumerable(Of Employee) = employees.Distinct(New
    PersonComparer())

        For Each employee In noduplicates
            Console.WriteLine(employee.FirstName & " " & employee.LastName)
        Next
    End Sub
```

See Also

Variance in Generic Interfaces (Visual Basic)

Variance in Delegates (Visual Basic)

Visual Studio 2015

.NET Framework 3.5 introduced variance support for matching method signatures with delegate types in all delegates in C# and Visual Basic. This means that you can assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. This includes both generic and non-generic delegates.

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

VB

```vb
Public Class First
End Class

Public Class Second
    Inherits First
End Class

Public Delegate Function SampleDelegate(ByVal a As Second) As First
Public Delegate Function SampleGenericDelegate(Of A, R)(ByVal a As A) As R
```

When you create delegates of the `SampleDelegate` or `SampleDelegate(Of A, R)` types, you can assign any one of the following methods to those delegates.

VB

```vb
' Matching signature.
Public Shared Function ASecondRFirst(
    ByVal second As Second) As First
    Return New First()
End Function

' The return type is more derived.
Public Shared Function ASecondRSecond(
    ByVal second As Second) As Second
    Return New Second()
End Function

' The argument type is less derived.
Public Shared Function AFirstRFirst(
    ByVal first As First) As First
    Return New First()
End Function

' The return type is more derived
' and the argument type is less derived.
Public Shared Function AFirstRSecond(
    ByVal first As First) As Second
    Return New Second()
End Function
```

The following code example illustrates the implicit conversion between the method signature and the delegate type.

VB

```
' Assigning a method with a matching signature
' to a non-generic delegate. No conversion is necessary.
Dim dNonGeneric As SampleDelegate = AddressOf ASecondRFirst
' Assigning a method with a more derived return type
' and less derived argument type to a non-generic delegate.
' The implicit conversion is used.
Dim dNonGenericConversion As SampleDelegate = AddressOf AFirstRSecond

' Assigning a method with a matching signature to a generic delegate.
' No conversion is necessary.
Dim dGeneric As SampleGenericDelegate(Of Second, First) = AddressOf ASecondRFirst
' Assigning a method with a more derived return type
' and less derived argument type to a generic delegate.
' The implicit conversion is used.
Dim dGenericConversion As SampleGenericDelegate(Of Second, First) = AddressOf
AFirstRSecond
```

For more examples, see Using Variance in Delegates (Visual Basic) and Using Variance for Func and Action Generic Delegates (Visual Basic).

Variance in Generic Type Parameters

In .NET Framework 4 and later you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the **in** or **out** keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

VB

```
' Type T is declared covariant by using the out keyword.
Public Delegate Function SampleGenericDelegate(Of Out T)() As T
Sub Test()
    Dim dString As SampleGenericDelegate(Of String) = Function() " "
    ' You can assign delegates to each other,
    ' because the type T is declared covariant.
    Dim dObject As SampleGenericDelegate(Of Object) = dString
End Sub
```

If you use only variance support to match method signatures with delegate types and do not use the **in** and **out** keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, `SampleGenericDelegate(Of String)` can't be explicitly converted to `SampleGenericDelegate(Of Object)`, although `String` inherits `Object`. You can fix this problem by marking the generic parameter `T` with the **out** keyword.

VB

```
Public Delegate Function SampleGenericDelegate(Of T)() As T
Sub Test()
    Dim dString As SampleGenericDelegate(Of String) = Function() " "

    ' You can assign the dObject delegate
    ' to the same lambda expression as dString delegate
    ' because of the variance support for
    ' matching method signatures with delegate types.
    Dim dObject As SampleGenericDelegate(Of Object) = Function() " "

    ' The following statement generates a compiler error
    ' because the generic type T is not marked as covariant.
    ' Dim dObject As SampleGenericDelegate(Of Object) = dString


End Sub
```

Generic Delegates That Have Variant Type Parameters in the .NET Framework

.NET Framework 4 introduced variance support for generic type parameters in several existing generic delegates:

- **Action** delegates from the System namespace, for example, Action(Of T) and Action(Of T1, T2)
- **Func** delegates from the System namespace, for example, Func(Of TResult) and Func(Of T, TResult)
- The Predicate(Of T) delegate
- The Comparison(Of T) delegate
- The Converter(Of TInput, TOutput) delegate

For more information and examples, see Using Variance for Func and Action Generic Delegates (Visual Basic).

Declaring Variant Type Parameters in Generic Delegates

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the **out** keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

VB

```
Public Delegate Function DCovariant(Of Out R)() As R
```

You can declare a generic type parameter contravariant in a generic delegate by using the **in** keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

VB

```
Public Delegate Sub DContravariant(Of In A)(ByVal a As A)
```

Important

ByRef parameters in Visual Basic can't be marked as variant.

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

VB

```
Public Delegate Function DVariant(Of In A, Out R)(ByVal a As A) As R
```

Instantiating and Invoking Variant Generic Delegates

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda expression.

VB

```
Dim dvariant As DVariant(Of String, String) = Function(str) str + " "
dvariant("test")
```

Combining Variant Generic Delegates

You should not combine variant delegates. The Combine method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the Combine method (in C# and Visual Basic) or by using the **+** operator (in C#), as shown in the following code example.

VB

```
im actObj As Action(Of Object) = Sub(x) Console.WriteLine("object: {0}", x)
```

```
Dim actStr As Action(Of String) = Sub(x) Console.WriteLine("string: {0}", x)

' The following statement throws an exception at run time.
' Dim actCombine = [Delegate].Combine(actStr, actObj)
```

Variance in Generic Type Parameters for Value and Reference Types

Variance for generic type parameters is supported for reference types only. For example, `DVariant(Of Int)`can't be implicitly converted to `DVariant(Of Object)` or `DVariant(Of Long)`, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

VB

```
' The type T is covariant.
Public Delegate Function DVariant(Of Out T)() As T
' The type T is invariant.
Public Delegate Function DInvariant(Of T)() As T
Sub Test()
    Dim i As Integer = 0
    Dim dInt As DInvariant(Of Integer) = Function() i
    Dim dVaraintInt As DVariant(Of Integer) = Function() i

    ' All of the following statements generate a compiler error
    ' because type variance in generic parameters is not supported
    ' for value types, even if generic type parameters are declared variant.
    ' Dim dObject As DInvariant(Of Object) = dInt
    ' Dim dLong As DInvariant(Of Long) = dInt
    ' Dim dVaraintObject As DInvariant(Of Object) = dInt
    ' Dim dVaraintLong As DInvariant(Of Long) = dInt
End Sub
```

Relaxed Delegate Conversion in Visual Basic

Relaxed delegate conversion enables more flexibility in matching method signatures with delegate types. For example, it lets you omit parameter specifications and omit function return values when you assign a method to a delegate. For more information, see Relaxed Delegate Conversion (Visual Basic).

See Also

Generics in the .NET Framework
Using Variance for Func and Action Generic Delegates (Visual Basic)

Using Variance in Delegates (Visual Basic)

Visual Studio 2015

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

VB

```
Class Mammals
End Class

Class Dogs
    Inherits Mammals
End Class
Class Test
    Public Delegate Function HandlerMethod() As Mammals
    Public Shared Function MammalsHandler() As Mammals
        Return Nothing
    End Function
    Public Shared Function DogsHandler() As Dogs
        Return Nothing
    End Function
    Sub Test()
        Dim handlerMammals As HandlerMethod = AddressOf MammalsHandler
        ' Covariance enables this assignment.
        Dim handlerDogs As HandlerMethod = AddressOf DogsHandler
    End Sub
End Class
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters of a type that are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. For example, you can create an event handler that accepts an `EventArgs` input parameter and use it with a `Button.MouseClick` event that sends a `MouseEventArgs` type as a parameter, and also with a `TextBox.KeyDown` event that sends a `KeyEventArgs` parameter.

Code

VB

```
' Event hander that accepts a parameter of the EventArgs type.
Private Sub MultiHandler(ByVal sender As Object,
                         ByVal e As System.EventArgs)
    Label1.Text = DateTime.Now
End Sub

Private Sub Form1_Load(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles MyBase.Load

    ' You can use a method that has an EventArgs parameter,
    ' although the event expects the KeyEventArgs parameter.
    AddHandler Button1.KeyDown, AddressOf MultiHandler

    ' You can use the same method
    ' for the event that expects the MouseEventArgs parameter.
    AddHandler Button1.MouseClick, AddressOf MultiHandler
End Sub
```

See Also

Variance in Delegates (Visual Basic)
Using Variance for Func and Action Generic Delegates (Visual Basic)

Using Variance for Func and Action Generic Delegates (Visual Basic)

Visual Studio 2015

These examples demonstrate how to use covariance and contravariance in the **Func** and **Action** generic delegates to enable reuse of methods and provide more flexibility in your code.

For more information about covariance and contravariance, see Variance in Delegates (Visual Basic).

Using Delegates with Covariant Type Parameters

The following example illustrates the benefits of covariance support in the generic **Func** delegates. The `FindByTitle` method takes a parameter of the `String` type and returns an object of the `Employee` type. However, you can assign this method to the `Func(Of String, Person)` delegate because `Employee` inherits `Person`.

VB

```
' Simple hierarchy of classes.
Public Class Person
End Class

Public Class Employee
    Inherits Person
End Class

Class Finder
    Public Shared Function FindByTitle(
        ByVal title As String) As Employee
        ' This is a stub for a method that returns
        ' an employee that has the specified title.
        Return New Employee
    End Function

    Sub Test()
        ' Create an instance of the delegate without using variance.
        Dim findEmployee As Func(Of String, Employee) =
            AddressOf FindByTitle

        ' The delegate expects a method to return Person,
        ' but you can assign it a method that returns Employee.
        Dim findPerson As Func(Of String, Person) =
            AddressOf FindByTitle

        ' You can also assign a delegate
        ' that returns a more derived type to a delegate
        ' that returns a less derived type.
        findPerson = findEmployee
    End Sub
End Class
```

Using Delegates with Contravariant Type Parameters

The following example illustrates the benefits of contravariance support in the generic `Action` delegates. The `AddToContacts` method takes a parameter of the `Person` type. However, you can assign this method to the `Action(Of Employee)` delegate because `Employee` inherits `Person`.

VB

```
Public Class Person
End Class

Public Class Employee
    Inherits Person
End Class

Class AddressBook
    Shared Sub AddToContacts(ByVal person As Person)
        ' This method adds a Person object
        ' to a contact list.
    End Sub

    Sub Test()
        ' Create an instance of the delegate without using variance.
        Dim addPersonToContacts As Action(Of Person) =
            AddressOf AddToContacts

        ' The Action delegate expects
        ' a method that has an Employee parameter,
        ' but you can assign it a method that has a Person parameter
        ' because Employee derives from Person.
        Dim addEmployeeToContacts As Action(Of Employee) =
            AddressOf AddToContacts

        ' You can also assign a delegate
        ' that accepts a less derived parameter
        ' to a delegate that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts
    End Sub
End Class
```

See Also

Covariance and Contravariance (Visual Basic)

Generics in the .NET Framework