CType Function (Visual Basic)

Visual Studio 2015

Returns the result of explicitly converting an expression to a specified data type, object, structure, class, or interface.

Syntax

```
CType(expression, typename)
```

Parts

expression
Any valid expression. If the value of *expression* is outside the range allowed by *typename*, Visual Basic throws an exception.

typename
Any expression that is legal within an **As** clause in a **Dim** statement, that is, the name of any data type, object, structure, class, or interface.

Remarks

Tip

You can also use the following functions to perform a type conversion:

- Type conversion functions such as **CByte**, **CDbl**, and **CInt** that perform a conversion to a specific data type. For more information, see Type Conversion Functions (Visual Basic).
- DirectCast Operator (Visual Basic) or TryCast Operator (Visual Basic). These operators require that one type inherit from or implement the other type. They can provide somewhat better performance than **CType** when converting to and from the **Object** data type.

CType is compiled inline, which means that the conversion code is part of the code that evaluates the expression. In some cases, the code runs faster because no procedures are called to perform the conversion.

If no conversion is defined from *expression* to *typename* (for example, from **Integer** to **Date**), Visual Basic displays a compile-time error message.

If a conversion fails at run time, the appropriate exception is thrown. If a narrowing conversion fails, an OverflowException is the most common result. If the conversion is undefined, an InvalidCastException in thrown. For example, this can happen if *expression* is of type **Object** and its run-time type has no conversion to *typename*.

If the data type of *expression* or *typename* is a class or structure you've defined, you can define **CType** on that class or structure as a conversion operator. This makes **CType** act as an *overloaded operator*. If you do this, you can control the behavior of conversions to and from your class or structure, including the exceptions that can be thrown.

Overloading

The **CType** operator can also be overloaded on a class or structure defined outside your code. If your code converts to or from such a class or structure, be sure you understand the behavior of its **CType** operator. For more information, see Operator Procedures (Visual Basic).

Converting Dynamic Objects

Type conversions of dynamic objects are performed by user-defined dynamic conversions that use the TryConvert or BindConvert methods. If you're working with dynamic objects, use the CTypeDynamic method to convert the dynamic object.

Example

The following example uses the **CType** function to convert an expression to the **Single** data type.

VB

```vb
Dim testNumber As Long = 1000
' The following line of code sets testNewType to 1000.0.
Dim testNewType As Single = CType(testNumber, Single)
```

For additional examples, see Implicit and Explicit Conversions (Visual Basic).

See Also

OverflowException
InvalidCastException
Type Conversion Functions (Visual Basic)
Conversion Functions (Visual Basic)
Operator Statement
How to: Define a Conversion Operator (Visual Basic)
Type Conversion in the .NET Framework

New Operator (Visual Basic)

Visual Studio 2015

Introduces a **New** clause to create a new object instance, specifies a constructor constraint on a type parameter, or identifies a **Sub** procedure as a class constructor.

Remarks

In a declaration or assignment statement, a **New** clause must specify a defined class from which the instance can be created. This means that the class must expose one or more constructors that the calling code can access.

You can use a **New** clause in a declaration statement or an assignment statement. When the statement runs, it calls the appropriate constructor of the specified class, passing any arguments you have supplied. The following example demonstrates this by creating instances of a `Customer` class that has two constructors, one that takes no parameters and one that takes a string parameter.

VB

```
' For customer1, call the constructor that takes no arguments.
Dim customer1 As New Customer()

' For customer2, call the constructor that takes the name of the
' customer as an argument.
Dim customer2 As New Customer("Blue Yonder Airlines")

' For customer3, declare an instance of Customer in the first line
' and instantiate it in the second.
Dim customer3 As Customer
customer3 = New Customer()

' With Option Infer set to On, the following declaration declares
' and instantiates a new instance of Customer.
Dim customer4 = New Customer("Coho Winery")
```

Since arrays are classes, **New** can create a new array instance, as shown in the following examples.

VB

```
Dim intArray1() As Integer
intArray1 = New Integer() {1, 2, 3, 4}

Dim intArray2() As Integer = {5, 6}

' The following example requires that Option Infer be set to On.
Dim intArray3() = New Integer() {6, 7, 8}
```

The common language runtime (CLR) throws an OutOfMemoryException error if there is insufficient memory to create the new instance.

Note

The **New** keyword is also used in type parameter lists to specify that the supplied type must expose an accessible parameterless constructor. For more information about type parameters and constraints, see Type List (Visual Basic).

To create a constructor procedure for a class, set the name of a **Sub** procedure to the **New** keyword. For more information, see Object Lifetime: How Objects Are Created and Destroyed (Visual Basic).

The **New** keyword can be used in these contexts:

Dim Statement (Visual Basic)

Of Clause (Visual Basic)

Sub Statement (Visual Basic)

See Also

OutOfMemoryException
Keywords (Visual Basic)
Type List (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)

Nothing (Visual Basic)

Visual Studio 2015

Represents the default value of any data type. For reference types, the default value is the **null** reference. For value types, the default value depends on whether the value type is nullable.

Note

For non-nullable value types, **Nothing** in Visual Basic differs from **null** in C#. In Visual Basic, if you set a variable of a non-nullable value type to **Nothing**, the variable is set to the default value for its declared type. In C#, if you assign a variable of a non-nullable value type to **null**, a compile-time error occurs.

Remarks

Nothing represents the default value of a data type. The default value depends on whether the variable is of a value type or of a reference type.

A variable of a *value type* directly contains its value. Value types include all numeric data types, **Boolean**, **Char**, **Date**, all structures, and all enumerations. A variable of a *reference type* stores a reference to an instance of the object in memory. Reference types include classes, arrays, delegates, and strings. For more information, see Value Types and Reference Types.

If a variable is of a value type, the behavior of **Nothing** depends on whether the variable is of a nullable data type. To represent a nullable value type, add a **?** modifier to the type name. Assigning **Nothing** to a nullable variable sets the value to **null**. For more information and examples, see Nullable Value Types (Visual Basic).

If a variable is of a value type that is not nullable, assigning **Nothing** to it sets it to the default value for its declared type. If that type contains variable members, they are all set to their default values. The following example illustrates this for scalar types.

VB

```
Module Module1

    Sub Main()
        Dim ts As TestStruct
        Dim i As Integer
        Dim b As Boolean

        ' The following statement sets ts.Name to Nothing and ts.Number to 0.
        ts = Nothing

        ' The following statements set i to 0 and b to False.
        i = Nothing
        b = Nothing
        Console.WriteLine("ts.Name: " & ts.Name)
        Console.WriteLine("ts.Number: " & ts.Number)
        Console.WriteLine("i: " & i)
        Console.WriteLine("b: " & b)

        Console.ReadKey()
    End Sub

    Public Structure TestStruct
        Public Name As String
        Public Number As Integer
    End Structure
End Module
```

If a variable is of a reference type, assigning **Nothing** to the variable sets it to a **null** reference of the variable's type. A variable that is set to a **null** reference is not associated with any object. The following example demonstrates this.

VB

```
Module Module1

    Sub Main()

        Dim testObject As Object
        ' The following statement sets testObject so that it does not refer to
        ' any instance.
        testObject = Nothing

        Dim tc As New TestClass
        tc = Nothing
        ' The fields of tc cannot be accessed. The following statement causes
        ' a NullReferenceException at run time. (Compare to the assignment of
        ' Nothing to structure ts in the previous example.)
        'Console.WriteLine(tc.Field1)

    End Sub

    Class TestClass
        Public Field1 As Integer
        ' . . .
    End Class
End Module
```

When checking whether a reference (or nullable value type) variable is **null**, do not use `= Nothing` or `<> Nothing`. Always use `Is Nothing` or `IsNot Nothing`.

For strings in Visual Basic, the empty string equals **Nothing**. Therefore, `"" = Nothing` is true.

The following example shows comparisons that use the **Is** and **IsNot** operators.

VB

```
Module Module1
```

```
    Sub Main()

        Dim testObject As Object
        testObject = Nothing
        Console.WriteLine(testObject Is Nothing)
        ' Output: True

        Dim tc As New TestClass
        tc = Nothing
        Console.WriteLine(tc IsNot Nothing)
        ' Output: False

        ' Declare a nullable value type.
        Dim n? As Integer
        Console.WriteLine(n Is Nothing)
        ' Output: True

        n = 4
        Console.WriteLine(n Is Nothing)
        ' Output: False

        n = Nothing
        Console.WriteLine(n IsNot Nothing)
        ' Output: False

        Console.ReadKey()
    End Sub

    Class TestClass
        Public Field1 As Integer
        Private field2 As Boolean
    End Class
End Module
```

If you declare a variable without using an **As** clause and set it to **Nothing**, the variable has a type of **Object**. An example of this is `Dim something = Nothing`. A compile-time error occurs in this case when **Option Strict** is on and **Option Infer** is off.

When you assign **Nothing** to an object variable, it no longer refers to any object instance. If the variable had previously referred to an instance, setting it to **Nothing** does not terminate the instance itself. The instance is terminated, and the memory and system resources associated with it are released, only after the garbage collector (GC) detects that there are no active references remaining.

Nothing differs from the DBNull object, which represents an uninitialized variant or a nonexistent database column.

See Also

Dim Statement (Visual Basic)
Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)
Lifetime in Visual Basic
Is Operator (Visual Basic)

IsNot Operator (Visual Basic)
Nullable Value Types (Visual Basic)

TryCast Operator (Visual Basic)

Visual Studio 2015

Introduces a type conversion operation that does not throw an exception.

Remarks

If an attempted conversion fails, **CType** and **DirectCast** both throw an InvalidCastException error. This can adversely affect the performance of your application. **TryCast** returns Nothing (Visual Basic), so that instead of having to handle a possible exception, you need only test the returned result against **Nothing**.

You use the **TryCast** keyword the same way you use the CType Function (Visual Basic) and the DirectCast Operator (Visual Basic) keyword. You supply an expression as the first argument and a type to convert it to as the second argument. **TryCast** operates only on reference types, such as classes and interfaces. It requires an inheritance or implementation relationship between the two types. This means that one type must inherit from or implement the other.

Errors and Failures

TryCast generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, **TryCast** returns Nothing (Visual Basic).

Conversion Keywords

A comparison of the type conversion keywords is as follows.

Keyword	Data types	Argument relationship	Run-time failure
CType Function (Visual Basic)	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws InvalidCastException
DirectCast Operator (Visual Basic)	Any data types	One type must inherit from or implement the other type	Throws InvalidCastException
TryCast	Reference types only	One type must inherit from or implement the other type	Returns Nothing (Visual Basic)

Example

The following example shows how to use **TryCast**.

VB

```
Function PrintTypeCode(ByVal obj As Object) As String
    Dim objAsConvertible As IConvertible = TryCast(obj, IConvertible)
    If objAsConvertible Is Nothing Then
        Return obj.ToString() & " does not implement IConvertible"
    Else
        Return "Type code is " & objAsConvertible.GetTypeCode()
    End If
End Function
```

See Also

Widening and Narrowing Conversions (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)

DirectCast Operator (Visual Basic)

Visual Studio 2015

Introduces a type conversion operation based on inheritance or implementation.

Remarks

DirectCast does not use the Visual Basic run-time helper routines for conversion, so it can provide somewhat better performance than **CType** when converting to and from data type **Object**.

You use the **DirectCast** keyword similar to the way you use the CType Function (Visual Basic) and the TryCast Operator (Visual Basic) keyword. You supply an expression as the first argument and a type to convert it to as the second argument. **DirectCast** requires an inheritance or implementation relationship between the data types of the two arguments. This means that one type must inherit from or implement the other.

Errors and Failures

DirectCast generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, the runtime throws an InvalidCastException error.

Conversion Keywords

A comparison of the type conversion keywords is as follows.

Keyword	Data types	Argument relationship	Run-time failure
CType Function (Visual Basic)	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws InvalidCastException
DirectCast	Any data types	One type must inherit from or implement the other type	Throws InvalidCastException
TryCast Operator (Visual Basic)	Reference types only	One type must inherit from or implement the other type	Returns Nothing (Visual Basic)

Example

The following example demonstrates two uses of **DirectCast**, one that fails at run time and one that succeeds.

VB

```
Dim q As Object = 2.37
Dim i As Integer = CType(q, Integer)
' The following conversion fails at run time
Dim j As Integer = DirectCast(q, Integer)
Dim f As New System.Windows.Forms.Form
Dim c As System.Windows.Forms.Control
' The following conversion succeeds.
c = DirectCast(f, System.Windows.Forms.Control)
```

In the preceding example, the run-time type of q is **Double**. **CType** succeeds because **Double** can be converted to **Integer**. However, the first **DirectCast** fails at run time because the run-time type of **Double** has no inheritance relationship with **Integer**, even though a conversion exists. The second **DirectCast** succeeds because it converts from type Form to type Control, from which Form inherits.

See Also

Convert.ChangeType
Widening and Narrowing Conversions (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)

TypeOf Operator (Visual Basic)

Visual Studio 2015

Compares an object reference variable to a data type.

Syntax

```
result = TypeOf objectexpression Is typename
```

```
result = TypeOf objectexpression IsNot typename
```

Parts

result
Returned. A **Boolean** value.

objectexpression
Required. Any expression that evaluates to a reference type.

typename
Required. Any data type name.

Remarks

The **TypeOf** operator determines whether the run-time type of *objectexpression* is compatible with *typename*. The compatibility depends on the type category of *typename*. The following table shows how compatibility is determined.

Type category of *typename*	**Compatibility criterion**
Class	*objectexpression* is of type *typename* or inherits from *typename*

Structure	*objectexpression* is of type *typename*
Interface	*objectexpression* implements *typename* or inherits from a class that implements *typename*

If the run-time type of *objectexpression* satisfies the compatibility criterion, *result* is **True**. Otherwise, *result* is **False**. If *objectexpression* is null, then **TypeOf**...**Is** returns **False**, and ...**IsNot** returns **True**.

TypeOf is always used with the **Is** keyword to construct a **TypeOf**...**Is** expression, or with the **IsNot** keyword to construct a **TypeOf**...**IsNot** expression.

Example

The following example uses **TypeOf**...**Is** expressions to test the type compatibility of two object reference variables with various data types.

VB

```
Dim refInteger As Object = 2
MsgBox("TypeOf Object[Integer] Is Integer? " & TypeOf refInteger Is Integer)
MsgBox("TypeOf Object[Integer] Is Double? " & TypeOf refInteger Is Double)
Dim refForm As Object = New System.Windows.Forms.Form
MsgBox("TypeOf Object[Form] Is Form? " & TypeOf refForm Is System.Windows.Forms.Form)
MsgBox("TypeOf Object[Form] Is Label? " & TypeOf refForm Is System.Windows.Forms.Label)
MsgBox("TypeOf Object[Form] Is Control? " & TypeOf refForm Is
System.Windows.Forms.Control)
MsgBox("TypeOf Object[Form] Is IComponent? " & TypeOf refForm Is
System.ComponentModel.IComponent)
```

The variable `refInteger` has a run-time type of **Integer**. It is compatible with **Integer** but not with **Double**. The variable `refForm` has a run-time type of Form. It is compatible with Form because that is its type, with Control because Form inherits from Control, and with IComponent because Form inherits from Component, which implements IComponent. However, `refForm` is not compatible with Label.

See Also

Is Operator (Visual Basic)
IsNot Operator (Visual Basic)
Comparison Operators in Visual Basic
Operator Precedence in Visual Basic
Operators Listed by Functionality (Visual Basic)
Operators and Expressions in Visual Basic

Type List (Visual Basic)

Visual Studio 2015

Specifies the *type parameters* for a *generic* programming element. Multiple parameters are separated by commas. Following is the syntax for one type parameter.

Syntax

```
[genericmodifier] typename [ As constraintlist ]
```

Parts

Term	Definition
genericmodifier	Optional. Can be used only in generic interfaces and delegates. You can declare a type covariant by using the Out keyword or contravariant by using the In keyword. See Covariance and Contravariance (C# and Visual Basic).
typename	Required. Name of the type parameter. This is a placeholder, to be replaced by a defined type supplied by the corresponding type argument.
constraintlist	Optional. List of requirements that constrain the data type that can be supplied for *typename*. If you have multiple constraints, enclose them in curly braces (**{ }**) and separate them with commas. You must introduce the constraint list with the As keyword. You use **As** only once, at the beginning of the list.

Remarks

Every generic programming element must take at least one type parameter. A type parameter is a placeholder for a specific type (a *constructed element*) that client code specifies when it creates an instance of the generic type. You can define a generic class, structure, interface, procedure, or delegate.

For more information on when to define a generic type, see Generic Types in Visual Basic (Visual Basic). For more information on type parameter names, see Declared Element Names (Visual Basic).

Rules

- **Parentheses.** If you supply a type parameter list, you must enclose it in parentheses, and you must introduce the list with the Of keyword. You use **Of** only once, at the beginning of the list.
- **Constraints.** A list of *constraints* on a type parameter can include the following items in any combination:
    - Any number of interfaces. The supplied type must implement every interface in this list.
    - At most one class. The supplied type must inherit from that class.
    - The **New** keyword. The supplied type must expose a parameterless constructor that your generic type can access. This is useful if you constrain a type parameter by one or more interfaces. A type that implements interfaces does not necessarily expose a constructor, and depending on the access level of a constructor, the code within the generic type might not be able to access it.
    - Either the **Class** keyword or the **Structure** keyword. The **Class** keyword constrains a generic type parameter to require that any type argument passed to it be a reference type, for example a string, array, or delegate, or an object created from a class. The **Structure** keyword constrains a generic type parameter to require that any type argument passed to it be a value type, for example a structure, enumeration, or elementary data type. You cannot include both **Class** and **Structure** in the same *constraintlist*.

  The supplied type must satisfy every requirement you include in *constraintlist*.

  Constraints on each type parameter are independent of constraints on other type parameters.

Behavior

- **Compile-Time Substitution.** When you create a constructed type from a generic programming element, you supply a defined type for each type parameter. The Visual Basic compiler substitutes that supplied type for every occurrence of *typename* within the generic element.
- **Absence of Constraints.** If you do not specify any constraints on a type parameter, your code is limited to the operations and members supported by the Object Data Type for that type parameter.

Example

The following example shows a skeleton definition of a generic dictionary class, including a skeleton function to add a new entry to the dictionary.

VB

```
Public Class dictionary(Of entryType, keyType As {IComparable, IFormattable, New})
    Public Sub add(ByVal et As entryType, ByVal kt As keyType)
        Dim dk As keyType
        If kt.CompareTo(dk) = 0 Then
        End If
    End Sub
End Class
```

Example

Because `dictionary` is generic, the code that uses it can create a variety of objects from it, each having the same functionality but acting on a different data type. The following example shows a line of code that creates a `dictionary` object with **String** entries and **Integer** keys.

VB

```
Dim dictInt As New dictionary(Of String, Integer)
```

Example

The following example shows the equivalent skeleton definition generated by the preceding example.

VB

```
Public Class dictionary
    Public Sub add(ByVal et As String, ByVal kt As Integer)
        Dim dk As Integer
        If kt.CompareTo(dk) = 0 Then
        End If
    End Sub
End Class
```

See Also

Of Clause (Visual Basic)
New Operator (Visual Basic)
Access Levels in Visual Basic
Object Data Type
Function Statement (Visual Basic)
Structure Statement
Sub Statement (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Covariance and Contravariance (C# and Visual Basic)
In (Generic Modifier) (Visual Basic)
Out (Generic Modifier) (Visual Basic)

Is Operator (Visual Basic)

Visual Studio 2015

Compares two object reference variables.

Syntax

```
result = object1 Is object2
```

Parts

result
Required. Any **Boolean** value.

object1
Required. Any **Object** name.

object2
Required. Any **Object** name.

Remarks

The **Is** operator determines if two object references refer to the same object. However, it does not perform value comparisons. If *object1* and *object2* both refer to the exact same object instance, *result* is **True**; if they do not, *result* is **False**.

Is can also be used with the **TypeOf** keyword to make a **TypeOf**...**Is** expression, which tests whether an object variable is compatible with a data type.

Note

The **Is** keyword is also used in the Select...Case Statement (Visual Basic).

Example

The following example uses the **Is** operator to compare pairs of object references. The results are assigned to a **Boolean** value representing whether the two objects are identical.

VB

```
Dim myObject As New Object
Dim otherObject As New Object
Dim yourObject, thisObject, thatObject As Object
Dim myCheck As Boolean
yourObject = myObject
thisObject = myObject
thatObject = otherObject
' The following statement sets myCheck to True.
myCheck = yourObject Is thisObject
' The following statement sets myCheck to False.
myCheck = thatObject Is thisObject
' The following statement sets myCheck to False.
myCheck = myObject Is thatObject
thatObject = myObject
' The following statement sets myCheck to True.
myCheck = thisObject Is thatObject
```

As the preceding example demonstrates, you can use the **Is** operator to test both early bound and late bound objects.

See Also

TypeOf Operator (Visual Basic)
IsNot Operator (Visual Basic)
Comparison Operators in Visual Basic
Operator Precedence in Visual Basic
Operators Listed by Functionality (Visual Basic)
Operators and Expressions in Visual Basic

Of Clause (Visual Basic)

Visual Studio 2015

Introduces an **Of** clause, which identifies a *type parameter* on a *generic* class, structure, interface, delegate, or procedure. For information on generic types, see Generic Types in Visual Basic (Visual Basic).

Using the Of Keyword

The following code example uses the **Of** keyword to define the outline of a class that takes two type parameters. It *constrains* the `keyType` parameter by the IComparable interface, which means the consuming code must supply a type argument that implements IComparable. This is necessary so that the `add` procedure can call the IComparable.CompareTo method. For more information on constraints, see Type List (Visual Basic).

```
Public Class Dictionary(Of entryType, keyType As IComparable)
    Public Sub add(ByVal e As entryType, ByVal k As keyType)
        Dim dk As keyType
        If k.CompareTo(dk) = 0 Then
        End If
    End Sub
    Public Function find(ByVal k As keyType) As entryType
    End Function
End Class
```

If you complete the preceding class definition, you can construct a variety of `dictionary` classes from it. The types you supply to `entryType` and `keyType` determine what type of entry the class holds and what type of key it associates with each entry. Because of the constraint, you must supply to `keyType` a type that implements IComparable.

The following code example creates an object that holds **String** entries and associates an **Integer** key with each one. **Integer** implements IComparable and therefore satisfies the constraint on `keyType`.

```
Dim d As New dictionary(Of String, Integer)
```

The **Of** keyword can be used in these contexts:

Class Statement

Delegate Statement

Function Statement

Interface Statement

Structure Statement

Sub Statement

See Also

IComparable
Type List (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
In (Generic Modifier) (Visual Basic)
Out (Generic Modifier) (Visual Basic)

In (Generic Modifier) (Visual Basic)

Visual Studio 2015

For generic type parameters, the **In** keyword specifies that the type parameter is contravariant.

Remarks

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types.

For more information, see Covariance and Contravariance (C# and Visual Basic).

Rules

You can use the **In** keyword in generic interfaces and delegates.

A type parameter can be declared contravariant in a generic interface or delegate if it is used only as a type of method arguments and not used as a method return type. **ByRef** parameters cannot be covariant or contravariant.

Covariance and contravariance are supported for reference types and not supported for value types.

In Visual Basic, you cannot declare events in contravariant interfaces without specifying the delegate type. Also, contravariant interfaces cannot have nested classes, enums, or structures, but they can have nested interfaces.

Behavior

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, because in .NET Framework 4, in the IComparer(Of T) interface, type T is contravariant, you can assign an object of the `IComparer(Of Person)` type to an object of the `IComparer(Of Employee)` type without using any special conversion methods if `Person` inherits `Employee`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

Example

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

VB

```
' Contravariant interface.
Interface IContravariant(Of In A)
```

```
End Interface

' Extending contravariant interface.
Interface IExtContravariant(Of In A)
    Inherits IContravariant(Of A)
End Interface

' Implementing contravariant interface.
Class Sample(Of A)
    Implements IContravariant(Of A)
End Class

Sub Main()
    Dim iobj As IContravariant(Of Object) = New Sample(Of Object)()
    Dim istr As IContravariant(Of String) = New Sample(Of String)()

    ' You can assign iobj to istr, because
    ' the IContravariant interface is contravariant.
    istr = iobj
End Sub
```

Example

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also shows how you can implicitly convert a delegate type.

VB

```
' Contravariant delegate.
Public Delegate Sub DContravariant(Of In A)(ByVal argument As A)

' Methods that match the delegate signature.
Public Shared Sub SampleControl(ByVal control As Control)
End Sub

Public Shared Sub SampleButton(ByVal control As Button)
End Sub

Private Sub Test()

    ' Instantiating the delegates with the methods.
    Dim dControl As DContravariant(Of Control) =
        AddressOf SampleControl
    Dim dButton As DContravariant(Of Button) =
        AddressOf SampleButton

    ' You can assign dControl to dButton
    ' because the DContravariant delegate is contravariant.
    dButton = dControl

    ' Invoke the delegate.
    dButton(New Button())
End Sub
```

See Also

Variance in Generic Interfaces (C# and Visual Basic)
Out (Generic Modifier) (Visual Basic)