# Interfaces (Visual Basic)

**Visual Studio 2015**

*Interfaces* define the properties, methods, and events that classes can implement. Interfaces allow you to define features as small groups of closely related properties, methods, and events; this reduces compatibility problems because you can develop enhanced implementations for your interfaces without jeopardizing existing code. You can add new features at any time by developing additional interfaces and implementations.

There are several other reasons why you might want to use interfaces instead of class inheritance:

- Interfaces are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces are more flexible than base classes because you can define a single implementation that can implement multiple interfaces.
- Interfaces are better in situations in which you do not have to inherit implementation from a base class.
- Interfaces are useful when you cannot use class inheritance. For example, structures cannot inherit from classes, but they can implement interfaces.

## Declaring Interfaces

Interface definitions are enclosed within the **Interface** and **End Interface** statements. Following the **Interface** statement, you can add an optional **Inherits** statement that lists one or more inherited interfaces. The **Inherits** statements must precede all other statements in the declaration except comments. The remaining statements in the interface definition should be **Event**, **Sub**, **Function**, **Property**, **Interface**, **Class**, **Structure**, and **Enum** statements. Interfaces cannot contain any implementation code or statements associated with implementation code, such as **End Sub** or **End Property**.

In a namespace, interface statements are **Friend** by default, but they can also be explicitly declared as **Public** or **Friend**. Interfaces defined within classes, modules, interfaces, and structures are **Public** by default, but they can also be explicitly declared as **Public**, **Friend**, **Protected**, or **Private**.

> **Note**
>
> The **Shadows** keyword can be applied to all interface members. The **Overloads** keyword can be applied to **Sub**, **Function**, and **Property** statements declared in an interface definition. In addition, **Property** statements can have the **Default**, **ReadOnly**, or **WriteOnly** modifiers. None of the other modifiers—**Public**, **Private**, **Friend**, **Protected**, **Shared**, **Overrides**, **MustOverride**, or **Overridable**—are allowed. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

For example, the following code defines an interface with one function, one property, and one event.

**VB**

```
Interface IAsset
    Event ComittedChange(ByVal Success As Boolean)
    Property Division() As String
    Function GetID() As Integer
End Interface
```

# Implementing Interfaces

The Visual Basic reserved word **Implements** is used in two ways. The **Implements** statement signifies that a class or structure implements an interface. The **Implements** keyword signifies that a class member or structure member implements a specific interface member.

## Implements Statement

If a class or structure implements one or more interfaces, it must include the **Implements** statement immediately after the **Class** or **Structure** statement. The **Implements** statement requires a comma-separated list of interfaces to be implemented by a class. The class or structure must implement all interface members using the **Implements** keyword.

## Implements Keyword

The **Implements** keyword requires a comma-separated list of interface members to be implemented. Generally, only a single interface member is specified, but you can specify multiple members. The specification of an interface member consists of the interface name, which must be specified in an implements statement within the class; a period; and the name of the member function, property, or event to be implemented. The name of a member that implements an interface member can use any legal identifier, and it is not limited to the **InterfaceName_MethodName** convention used in earlier versions of Visual Basic.

For example, the following code shows how to declare a subroutine named `Sub1` that implements a method of an interface:

**VB**

```
Class Class1
    Implements interfaceclass.interface2

    Sub Sub1(ByVal i As Integer) Implements interfaceclass.interface2.Sub1
    End Sub
End Class
```

The parameter types and return types of the implementing member must match the interface property or member declaration in the interface. The most common way to implement an element of an interface is with a member that has the same name as the interface, as shown in the previous example.

To declare the implementation of an interface method, you can use any attributes that are legal on instance method declarations, including **Overloads**, **Overrides**, **Overridable**, **Public**, **Private**, **Protected**, **Friend**, **Protected Friend**, **MustOverride**, **Default**, and **Static**. The **Shared** attribute is not legal since it defines a class rather than an instance

method.

Using **Implements**, you can also write a single method that implements multiple methods defined in an interface, as in the following example:

**VB**

```
Class Class2
    Implements I1, I2

    Protected Sub M1() Implements I1.M1, I1.M2, I2.M3, I2.M4
    End Sub
End Class
```

You can use a private member to implement an interface member. When a private member implements a member of an interface, that member becomes available by way of the interface even though it is not available directly on object variables for the class.

## Interface Implementation Examples

Classes that implement an interface must implement all its properties, methods, and events.

The following example defines two interfaces. The second interface, `Interface2`, inherits `Interface1` and defines an additional property and method.

**VB**

```
Interface Interface1
    Sub sub1(ByVal i As Integer)
End Interface

' Demonstrates interface inheritance.
Interface Interface2
    Inherits Interface1
    Sub M1(ByVal y As Integer)
    ReadOnly Property Num() As Integer
End Interface
```

The next example implements `Interface1`, the interface defined in the previous example:

**VB**

```
Public Class ImplementationClass1
    Implements Interface1
    Sub Sub1(ByVal i As Integer) Implements Interface1.sub1
        ' Insert code here to implement this method.
    End Sub
End Class
```

The final example implements `Interface2`, including a method inherited from `Interface1`:

**VB**

```
Public Class ImplementationClass2
    Implements Interface2
    Dim INum As Integer = 0
    Sub sub1(ByVal i As Integer) Implements Interface2.sub1
        ' Insert code here that implements this method.
    End Sub
    Sub M1(ByVal x As Integer) Implements Interface2.M1
        ' Insert code here to implement this method.
    End Sub

    ReadOnly Property Num() As Integer Implements Interface2.Num
        Get
            Num = INum
        End Get
    End Property
End Class
```

You can implement a readonly property with a readwrite property (that is, you do not have to declare it readonly in the implementing class). Implementing an interface promises to implement at least the members that the interface declares, but you can offer more functionality, such as allowing your property to be writable.

# Related Topics

| Title | Description |
| --- | --- |
| Walkthrough: Creating and Implementing Interfaces (Visual Basic) | Provides a detailed procedure that takes you through the process of defining and implementing your own interface. |
| Variance in Generic Interfaces (C# and Visual Basic) | Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework. |

# 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)

# Walkthrough: Creating and Implementing Interfaces (Visual Basic)

**Visual Studio 2015**

Interfaces describe the characteristics of properties, methods, and events, but leave the implementation details up to structures or classes.

This walkthrough demonstrates how to declare and implement an interface.

> **Note**
>
> This walkthrough doesn't provide information about how to create a user interface.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the Visual Studio IDE.

## To define an interface

1. Open a new Visual Basic Windows Application project.
2. Add a new module to the project by clicking **Add Module** on the **Project** menu.
3. Name the new module `Module1.vb` and click **Add**. The code for the new module is displayed.
4. Define an interface named `TestInterface` within `Module1` by typing `Interface TestInterface` between the **Module** and **End Module** statements, and then pressing ENTER. The **Code Editor** indents the **Interface** keyword and adds an **End Interface** statement to form a code block.
5. Define a property, method, and event for the interface by placing the following code between the **Interface** and **End Interface** statements:

**VB**

```
Property Prop1() As Integer
Sub Method1(ByVal X As Integer)
Event Event1()
```

# Implementation

You may notice that the syntax used to declare interface members is different from the syntax used to declare class members. This difference reflects the fact that interfaces cannot contain implementation code.

### To implement the interface

1. Add a class named `ImplementationClass` by adding the following statement to `Module1`, after the `End Interface` statement but before the `End Module` statement, and then pressing ENTER:

   **VB**

   ```
   Class ImplementationClass
   ```

   If you are working within the integrated development environment, the **Code Editor** supplies a matching `End Class` statement when you press ENTER.

2. Add the following **Implements** statement to `ImplementationClass`, which names the interface the class implements:

   **VB**

   ```
   Implements TestInterface
   ```

   When listed separately from other items at the top of a class or structure, the **Implements** statement indicates that the class or structure implements an interface.

   If you are working within the integrated development environment, the **Code Editor** implements the class members required by `TestInterface` when you press ENTER, and you can skip the next step.

3. If you are not working within the integrated development environment, you must implement all the members of the interface `MyInterface`. Add the following code to `ImplementationClass` to implement `Event1`, `Method1`, and `Prop1`:

   **VB**

   ```
   Event Event1() Implements TestInterface.Event1

   Public Sub Method1(ByVal X As Integer) Implements TestInterface.Method1
   End Sub

   Public Property Prop1() As Integer Implements TestInterface.Prop1
       Get
       End Get
       Set(ByVal value As Integer)
       End Set
   End Property
   ```

   The **Implements** statement names the interface and interface member being implemented.

4. Complete the definition of `Prop1` by adding a private field to the class that stored the property value:

```
VB
' Holds the value of the property.
Private pval As Integer
```

Return the value of the `pval` from the property get accessor.

```
VB
Return pval
```

Set the value of `pval` in the property set accessor.

```
VB
pval = value
```

5. Complete the definition of `Method1` by adding the following code.

```
VB
MsgBox("The X parameter for Method1 is " & X)
RaiseEvent Event1()
```

## To test the implementation of the interface

1. Right-click the startup form for your project in the **Solution Explorer**, and click **View Code**. The editor displays the class for your startup form. By default, the startup form is called `Form1`.

2. Add the following `testInstance` field to the `Form1` class:

```
VB
Dim WithEvents testInstance As TestInterface
```

By declaring `testInstance` as **WithEvents**, the `Form1` class can handle its events.

3. Add the following event handler to the `Form1` class to handle events raised by `testInstance`:

```
VB
Sub EventHandler() Handles testInstance.Event1
    MsgBox("The event handler caught the event.")
End Sub
```

4. Add a subroutine named `Test` to the `Form1` class to test the implementation class:

**VB**

```
Sub Test()
    '  Create an instance of the class.
    Dim T As New ImplementationClass
    ' Assign the class instance to the interface.
    ' Calls to the interface members are
    ' executed through the class instance.
    testInstance = T
    ' Set a property.
    testInstance.Prop1 = 9
    ' Read the property.
    MsgBox("Prop1 was set to " & testInstance.Prop1)
    '  Test the method and raise an event.
    testInstance.Method1(5)
End Sub
```

The `Test` procedure creates an instance of the class that implements `MyInterface`, assigns that instance to the `testInstance` field, sets a property, and runs a method through the interface.

5. Add code to call the `Test` procedure from the `Form1 Load` procedure of your startup form:

**VB**

```
Private Sub Form1_Load(ByVal sender As System.Object,
                       ByVal e As System.EventArgs) Handles MyBase.Load
    Test() ' Test the class.
End Sub
```

6. Run the `Test` procedure by pressing F5. The message "Prop1 was set to 9" is displayed. After you click OK, the message "The X parameter for Method1 is 5" is displayed. Click OK, and the message "The event handler caught the event" is displayed.

# See Also

Implements Statement
Interfaces (Visual Basic)
Interface Statement (Visual Basic)
Event Statement