Declared Element Names (Visual Basic)

Visual Studio 2015

Every declared element has a name, also called an *identifier*, which is what the code uses to refer to it.

Rules

An element name in Visual Basic must observe the following rules:

- It must begin with an alphabetic character or an underscore (_).
- It must only contain alphabetic characters, decimal digits, and underscores.
- It must contain at least one alphabetic character or decimal digit if it begins with an underscore.
- It must not be more than 1023 characters long.

The length limit of 1023 characters also applies to the entire string of a fully qualified name, such as `outerNamespace.middleNamespace.innerNamespace.thisClass.thisElement`.

The following example shows some valid element names.

```
aB123__45

_567
```

The following example shows some invalid element names. The first contains only an underscore, the second begins with a decimal digit, and the third contains an invalid character ($).

```
' Three INVALID element names

_

12ABC

xyz$wv
```

Caution

Element names starting with an underscore (_) are not part of the Language Independence and Language-Independent Components (CLS), so CLS-compliant code cannot use a component that defines such names. However, an underscore in any other position in an element name is CLS-compliant.

Name Length Guidelines

As a practical matter, your name should be as short as possible while still clearly identifying the nature of the element. This improves the readability of your code and reduces line length and source-file size.

On the other hand, your name should not be so short that it does not adequately describe what the element represents and how your code uses it. This is important for the readability of your code. If somebody else is trying to understand it, or if you yourself are looking at it a long time after you wrote it, suitable element names can save a considerable amount of time.

Escaped Names

Generally, an element name must not match any of the keywords reserved by Visual Basic, such as **Case** or **Friend**. However, you can define an *escaped name*, which is enclosed by brackets (**[ ]**). An escaped name can match any Visual Basic keyword, since the brackets remove any ambiguity. You also use the brackets when you refer to the name later in your code.

In general, you should use escaped names only when:

- Your code has migrated from a previous version of Visual Basic that did not reserve the keyword being used as a name; or
- You are working with code written in another language in which the given keyword is not reserved.

Otherwise, you should consider renaming the element if its name conflicts with a keyword. The integrated development environment (IDE) provides an easy way to do this. For more information, see Refactoring and Rename Dialog Box (Visual Basic).

Case Sensitivity in Names

Element names in Visual Basic are case-insensitive. This means that when the compiler compares two names that differ in alphabetic case only, it interprets them as the same name. For example, it considers `ABC` and `abc` to refer to the same declared element.

However, the common language runtime (CLR) uses case-sensitive binding. Therefore, when you produce an assembly or a DLL and make it available to other assemblies, your names are no longer case-insensitive. For example, if you define a class with an element called `ABC`, and other assemblies make use of your class through the common language runtime, they must refer to the element as `ABC`. If you subsequently recompile your class and change the element's name to `abc`, the other assemblies using your class could no longer access that element. Therefore, when you release an updated version of an assembly, you should not change the alphabetic case of any public elements.

Names and Locales

Comparison of names is independent of locale. If two names match in one locale, they are guaranteed to match in all

locales.

See Also

Declared Elements in Visual Basic
Declared Element Characteristics (Visual Basic)
References to Declared Elements (Visual Basic)
Statements (Visual Basic)

References to Declared Elements (Visual Basic)

Visual Studio 2015

When your code refers to a declared element, the Visual Basic compiler matches the name in your reference to the appropriate declaration of that name. If more than one element is declared with the same name, you can control which of those elements is to be referenced by *qualifying* its name.

The compiler attempts to match a name reference to a name declaration with the *narrowest scope*. This means it starts with the code making the reference and works outward through successive levels of containing elements.

The following example shows references to two variables with the same name. The example declares two variables, each named `totalCount`, at different levels of scope in module `container`. When the procedure `showCount` displays `totalCount` without qualification, the Visual Basic compiler resolves the reference to the declaration with the narrowest scope, namely the local declaration inside `showCount`. When it qualifies `totalCount` with the containing module `container`, the compiler resolves the reference to the declaration with the broader scope.

VB

```
' Assume these two modules are both in the same assembly.
Module container
    Public totalCount As Integer = 1
    Public Sub showCount()
        Dim totalCount As Integer = 6000
        ' The following statement displays the local totalCount (6000).
        MsgBox("Unqualified totalCount is " & CStr(totalCount))
        ' The following statement displays the module's totalCount (1).
        MsgBox("container.totalCount is " & CStr(container.totalCount))
    End Sub
End Module
Module callingModule
    Public Sub displayCount()
        container.showCount()
        ' The following statement displays the containing module's totalCount (1).
        MsgBox("container.totalCount is " & CStr(container.totalCount))
    End Sub
End Module
```

Qualifying an Element Name

If you want to override this search process and specify a name declared in a broader scope, you must *qualify* the name with the containing element of the broader scope. In some cases, you might also have to qualify the containing element.

Qualifying a name means preceding it in your source statement with information that identifies where the target element is defined. This information is called a *qualification string*. It can include one or more namespaces and a module, class, or

structure.

The qualification string should unambiguously specify the module, class, or structure containing the target element. The container might in turn be located in another containing element, usually a namespace. You might need to include several containing elements in the qualification string.

To access a declared element by qualifying its name

1. Determine the location in which the element has been defined. This might include a namespace, or even a hierarchy of namespaces. Within the lowest-level namespace, the element must be contained in a module, class, or structure.

   VB

   ```
   ' Assume the following hierarchy exists outside your code.
   Namespace outerSpace
       Namespace innerSpace
           Module holdsTotals
               Public Structure totals
                   Public thisTotal As Integer
                   Public Shared grandTotal As Long
               End Structure
           End Module
       End Namespace
   End Namespace
   ```

2. Determine a qualification path based on the target element's location. Start with the highest-level namespace, proceed to the lowest-level namespace, and end with the module, class, or structure containing the target element. Each element in the path must contain the element that follows it.

   `outerSpace` → `innerSpace` → `holdsTotals` → `totals`

3. Prepare the qualification string for the target element. Place a period (**.**) after every element in the path. Your application must have access to every element in your qualification string.

   VB

   ```
   outerSpace.innerSpace.holdsTotals.totals.
   ```

4. Write the expression or assignment statement referring to the target element in the normal way.

   VB

   ```
   grandTotal = 9000
   ```

5. Precede the target element name with the qualification string. The name should immediately follow the period (**.**) that follows the module, class, or structure that contains the element.

   VB

   ```
   ' Assume the following module is part of your code.
   Module accessGrandTotal
   ```

```
        Public Sub setGrandTotal()
            outerSpace.innerSpace.holdsTotals.totals.grandTotal = 9000
        End Sub
    End Module
```

6. The compiler uses the qualification string to find a clear, unambiguous declaration to which it can match the target element reference.

You might also have to qualify a name reference if your application has access to more than one programming element that has the same name. For example, the System.Windows.Forms and System.Web.UI.WebControls namespaces both contain a `Label` class (System.Windows.Forms.Label and System.Web.UI.WebControls.Label). If your application uses both, or if it defines its own `Label` class, you must distinguish the different `Label` objects. Include the namespace or import alias in the variable declaration. The following example uses the import alias.

VB

```
' The following statement must precede all your declarations.
Imports win = System.Windows.Forms, web = System.Web.UI.WebControls
' The following statement references the Windows.Forms.Label class.
Dim winLabel As New win.Label()
```

Members of Other Containing Elements

When you use a nonshared member of another class or structure, you must first qualify the member name with a variable or expression that points to an instance of the class or structure. In the following example, `demoClass` is an instance of a class named `class1`.

VB

```
Dim demoClass As class1 = New class1()
demoClass.someSub[(argumentlist)]
```

You cannot use the class name itself to qualify a member that is not Shared (Visual Basic). You must first create an instance in an object variable (in this case `demoClass`) and then reference it by the variable name.

If a class or structure has a **Shared** member, you can qualify that member either with the class or structure name or with a variable or expression that points to an instance.

A module does not have any separate instances, and all its members are **Shared** by default. Therefore, you qualify a module member with the module name.

The following example shows qualified references to module member procedures. The example declares two **Sub** procedures, both named `perform`, in different modules in a project. Each one can be specified without qualification within its own module but must be qualified if referenced from anywhere else. Because the final reference in `module3` does not qualify `perform`, the compiler cannot resolve that reference.

VB

```
' Assume these three modules are all in the same assembly.
Module module1
    Public Sub perform()
        MsgBox("module1.perform() now returning")
    End Sub
End Module
Module module2
    Public Sub perform()
        MsgBox("module2.perform() now returning")
    End Sub
    Public Sub doSomething()
        ' The following statement calls perform in module2, the active module.
        perform()
        ' The following statement calls perform in module1.
        module1.perform()
    End Sub
End Module
Module module3
    Public Sub callPerform()
        ' The following statement calls perform in module1.
        module1.perform()
        ' The following statement makes an unresolvable name reference
        ' and therefore generates a COMPILER ERROR.
        perform() ' INVALID statement
    End Sub
End Module
```

References to Projects

To use Public (Visual Basic) elements defined in another project, you must first set a *reference* to that project's assembly or type library. To set a reference, click **Add Reference** on the **Project** menu, or use the -reference (Visual Basic) command-line compiler option.

For example, you can use the XML object model of the .NET Framework. If you set a reference to the System.Xml namespace, you can declare and use any of its classes, such as XmlDocument. The following example uses XmlDocument.

VB

```
' Assume this project has a reference to System.Xml
' The following statement creates xDoc as an XML document object.
Dim xDoc As System.Xml.XmlDocument
```

Importing Containing Elements

You can use the Imports Statement (.NET Namespace and Type) to *import* the namespaces that contain the modules or classes that you want to use. This enables you to refer to the elements defined in an imported namespace without fully qualifying their names. The following example rewrites the previous example to import the System.Xml namespace.

VB

```
' Assume this project has a reference to System.Xml
' The following statement must precede all your declarations.
Imports System.Xml
' The following statement creates xDoc as an XML document object.
Dim xDoc As XmlDocument
```

In addition, the **Imports** statement can define an *import alias* for each imported namespace. This can make the source code shorter and easier to read. The following example rewrites the previous example to use `xD` as an alias for the System.Xml namespace.

VB

```
' Assume this project has a reference to System.Xml
' The following statement must precede all your declarations.
Imports xD = System.Xml
' The following statement creates xDoc as an XML document object.
Dim xDoc As xD.XmlDocument
```

The **Imports** statement does not make elements from other projects available to your application. That is, it does not take the place of setting a reference. Importing a namespace just removes the requirement to qualify the names defined in that namespace.

You can also use the **Imports** statement to import modules, classes, structures, and enumerations. You can then use the members of such imported elements without qualification. However, you must always qualify nonshared members of classes and structures with a variable or expression that evaluates to an instance of the class or structure.

Naming Guidelines

When you define two or more programming elements that have the same name, a *name ambiguity* can result when the compiler attempts to resolve a reference to that name. If more than one definition is in scope, or if no definition is in scope, the reference is irresolvable. For an example, see "Qualified Reference Example" on this Help page.

You can avoid name ambiguity by giving all your elements unique names. Then you can make reference to any element without having to qualify its name with a namespace, module, or class. You also reduce the chances of accidentally referring to the wrong element.

Shadowing

When two programming elements share the same name, one of them can hide, or *shadow*, the other one. A shadowed element is not available for reference; instead, when your code uses the shadowed element name, the Visual Basic compiler resolves it to the shadowing element. For a more detailed explanation with examples, see Shadowing in Visual Basic.

See Also

- Declared Element Names (Visual Basic)
- Declared Element Characteristics (Visual Basic)
- NIB How to: Modify Project Properties and Configuration Settings
- Variables in Visual Basic
- Imports Statement (.NET Namespace and Type)
- New Operator (Visual Basic)
- Public (Visual Basic)

Shadowing in Visual Basic

Visual Studio 2015

When two programming elements share the same name, one of them can hide, or *shadow*, the other one. In such a situation, the shadowed element is not available for reference; instead, when your code uses the element name, the Visual Basic compiler resolves it to the shadowing element.

Purpose

The main purpose of shadowing is to protect the definition of your class members. The base class might undergo a change that creates an element with the same name as one you have already defined. If this happens, the **Shadows** modifier forces references through your class to be resolved to the member you defined, instead of to the new base class element.

Types of Shadowing

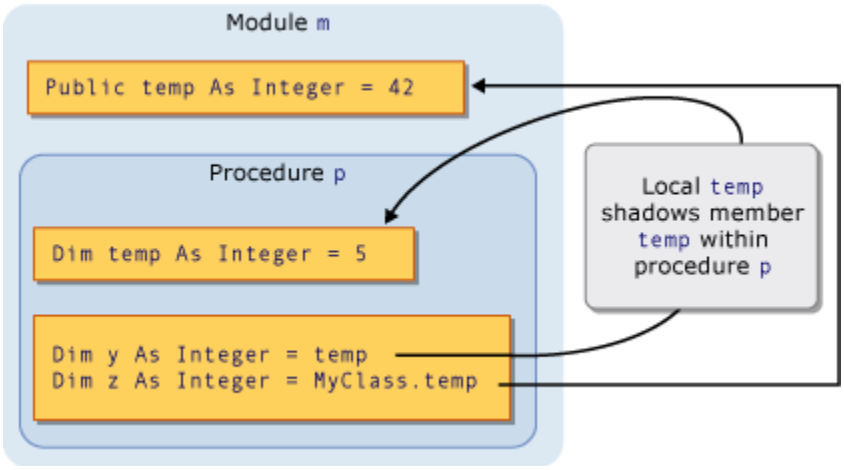
An element can shadow another element in two different ways. The shadowing element can be declared inside a subregion of the region containing the shadowed element, in which case the shadowing is accomplished *through scope*. Or a deriving class can redefine a member of a base class, in which case the shadowing is done *through inheritance*.

Shadowing Through Scope

It is possible for programming elements in the same module, class, or structure to have the same name but different scope. When two elements are declared in this manner and the code refers to the name they share, the element with the narrower scope shadows the other element (block scope is the narrowest).

For example, a module can define a **Public** variable named `temp`, and a procedure within the module can declare a local variable also named `temp`. References to `temp` from within the procedure access the local variable, while references to `temp` from outside the procedure access the **Public** variable. In this case, the procedure variable `temp` shadows the module variable `temp`.

The following illustration shows two variables, both named `temp`. The local variable `temp` shadows the member variable `temp` when accessed from within its own procedure `p`. However, the **MyClass** keyword bypasses the shadowing and accesses the member variable.



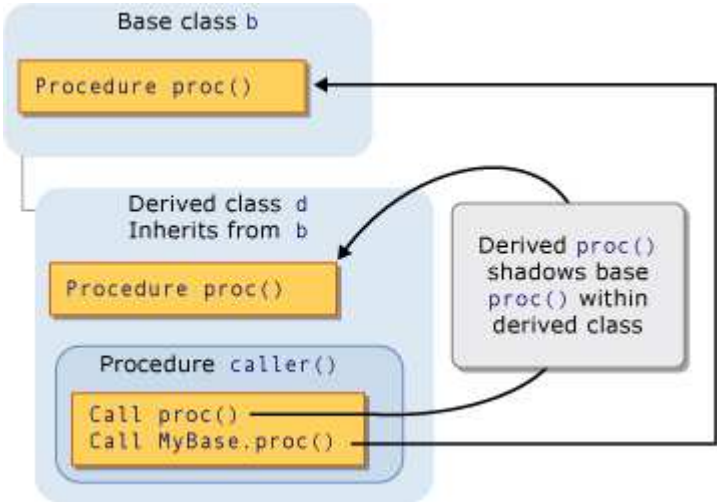
Shadowing through scope

For an example of shadowing through scope, see How to: Hide a Variable with the Same Name as Your Variable (Visual Basic).

Shadowing Through Inheritance

If a derived class redefines a programming element inherited from a base class, the redefining element shadows the original element. You can shadow any type of declared element, or set of overloaded elements, with any other type. For example, an **Integer** variable can shadow a **Function** procedure. If you shadow a procedure with another procedure, you can use a different parameter list and a different return type.

The following illustration shows a base class `b` and a derived class `d` that inherits from `b`. The base class defines a procedure named `proc`, and the derived class shadows it with another procedure of the same name. The first **Call** statement accesses the shadowing `proc` in the derived class. However, the **MyBase** keyword bypasses the shadowing and accesses the shadowed procedure in the base class.



Shadowing through inheritance

For an example of shadowing through inheritance, see How to: Hide a Variable with the Same Name as Your Variable (Visual Basic) and How to: Hide an Inherited Variable (Visual Basic).

Shadowing and Access Level

The shadowing element is not always accessible from the code using the derived class. For example, it might be

declared **Private**. In such a case, shadowing is defeated and the compiler resolves any reference to the same element it would have if there had been no shadowing. This element is the accessible element the fewest derivational steps backward from the shadowing class. If the shadowed element is a procedure, the resolution is to the closest accessible version with the same name, parameter list, and return type.

The following example shows an inheritance hierarchy of three classes. Each class defines a **Sub** procedure `display`, and each derived class shadows the `display` procedure in its base class.

```
Public Class firstClass
    Public Sub display()
        MsgBox("This is firstClass")
    End Sub
End Class
Public Class secondClass
    Inherits firstClass
    Private Shadows Sub display()
        MsgBox("This is secondClass")
    End Sub
End Class
Public Class thirdClass
    Inherits secondClass
    Public Shadows Sub display()
        MsgBox("This is thirdClass")
    End Sub
End Class
Module callDisplay
    Dim first As New firstClass
    Dim second As New secondClass
    Dim third As New thirdClass
    Public Sub callDisplayProcedures()
        ' The following statement displays "This is firstClass".
        first.display()
        ' The following statement displays "This is firstClass".
        second.display()
        ' The following statement displays "This is thirdClass".
        third.display()
    End Sub
End Module
```

In the preceding example, the derived class `secondClass` shadows `display` with a **Private** procedure. When module `callDisplay` calls `display` in `secondClass`, the calling code is outside `secondClass` and therefore cannot access the private `display` procedure. Shadowing is defeated, and the compiler resolves the reference to the base class `display` procedure.

However, the further derived class `thirdClass` declares `display` as **Public**, so the code in `callDisplay` can access it.

Shadowing and Overriding

Do not confuse shadowing with overriding. Both are used when a derived class inherits from a base class, and both redefine one declared element with another. But there are significant differences between the two. For a comparison, see Differences Between Shadowing and Overriding (Visual Basic).

Shadowing and Overloading

If you shadow the same base class element with more than one element in your derived class, the shadowing elements become overloaded versions of that element. For more information, see Procedure Overloading (Visual Basic).

Accessing a Shadowed Element

When you access an element from a derived class, you normally do so through the current instance of that derived class, by qualifying the element name with the **Me** keyword. If your derived class shadows the element in the base class, you can access the base class element by qualifying it with the **MyBase** keyword.

For an example of accessing a shadowed element, see How to: Access a Variable Hidden by a Derived Class (Visual Basic).

Declaration of the Object Variable

How you create the object variable can also affect whether the derived class accesses a shadowing element or the shadowed element. The following example creates two objects from a derived class, but one object is declared as the base class and the other as the derived class.

```
Public Class baseCls
    ' The following statement declares the element that is to be shadowed.
    Public z As Integer = 100
End Class
Public Class dervCls
    Inherits baseCls
    ' The following statement declares the shadowing element.
    Public Shadows z As String = "*"
End Class
Public Class useClasses
    ' The following statement creates the object declared as the base class.
    Dim basObj As baseCls = New dervCls()
    ' Note that dervCls widens to its base class baseCls.
    ' The following statement creates the object declared as the derived class.
    Dim derObj As dervCls = New dervCls()
    Public Sub showZ()
    ' The following statement outputs 100 (the shadowed element).
        MsgBox("Accessed through base class: " & basObj.z)
    ' The following statement outputs "*" (the shadowing element).
        MsgBox("Accessed through derived class: " & derObj.z)
    End Sub
```

```
End Class
```

In the preceding example, the variable `basObj` is declared as the base class. Assigning a `dervCls` object to it constitutes a widening conversion and is therefore valid. However, the base class cannot access the shadowing version of the variable `z` in the derived class, so the compiler resolves `basObj.z` to the original base class value.

See Also

References to Declared Elements (Visual Basic)
Scope in Visual Basic
Widening and Narrowing Conversions (Visual Basic)
Shadows (Visual Basic)
Overrides (Visual Basic)
Me, My, MyBase, and MyClass in Visual Basic
Inheritance Basics (Visual Basic)

Differences Between Shadowing and Overriding (Visual Basic)

Visual Studio 2015

When you define a class that inherits from a base class, you sometimes want to redefine one or more of the base class elements in the derived class. Shadowing and overriding are both available for this purpose.

Comparison

Shadowing and overriding are both used when a derived class inherits from a base class, and both redefine one declared element with another. But there are significant differences between the two.

The following table compares shadowing with overriding.

Point of comparison	Shadowing	Overriding
Purpose	Protects against a subsequent base-class modification that introduces a member you have already defined in your derived class	Achieves polymorphism by defining a different implementation of a procedure or property with the same calling sequence[1]
Redefined element	Any declared element type	Only a procedure (**Function**, **Sub**, or **Operator**) or property
Redefining element	Any declared element type	Only a procedure or property with the identical calling sequence[1]
Access level of redefining element	Any access level	Cannot change access level of overridden element
Readability and writability of redefining element	Any combination	Cannot change readability or writability of overridden property
Control over redefining	Base class element cannot enforce or prohibit shadowing	Base class element can specify **MustOverride**, **NotOverridable**, or **Overridable**
Keyword usage	**Shadows** recommended in derived class; **Shadows** assumed if neither **Shadows** nor **Overrides** specified[2]	**Overridable** or **MustOverride** required in base class; **Overrides** required in derived class

Inheritance of redefining element by classes deriving from your derived class	Shadowing element inherited by further derived classes; shadowed element still hidden[3]	Overriding element inherited by further derived classes; overridden element still overridden

[1] The *calling sequence* consists of the element type (**Function**, **Sub**, **Operator**, or **Property**), name, parameter list, and return type. You cannot override a procedure with a property, or the other way around. You cannot override one kind of procedure (**Function**, **Sub**, or **Operator**) with another kind.

[2] If you do not specify either **Shadows** or **Overrides**, the compiler issues a warning message to help you be sure which kind of redefinition you want to use. If you ignore the warning, the shadowing mechanism is used.

[3] If the shadowing element is inaccessible in a further derived class, shadowing is not inherited. For example, if you declare the shadowing element as **Private**, a class deriving from your derived class inherits the original element instead of the shadowing element.

Guidelines

You normally use overriding in the following cases:

- You are defining polymorphic derived classes.
- You want the safety of having the compiler enforce the identical element type and calling sequence.

You normally use shadowing in the following cases:

- You anticipate that your base class might be modified and define an element using the same name as yours.
- You want the freedom of changing the element type or calling sequence.

See Also

References to Declared Elements (Visual Basic)
Shadowing in Visual Basic
How to: Hide a Variable with the Same Name as Your Variable (Visual Basic)
How to: Hide an Inherited Variable (Visual Basic)
How to: Access a Variable Hidden by a Derived Class (Visual Basic)
Shadows (Visual Basic)
Overrides (Visual Basic)

How to: Hide a Variable with the Same Name as Your Variable (Visual Basic)

Visual Studio 2015

You can hide a variable by *shadowing* it, that is, by redefining it with a variable of the same name. You can shadow the variable you want to hide in two ways:

- **Shadowing Through Scope.** You can shadow it through scope by redeclaring it inside a subregion of the region containing the variable you want to hide.
- **Shadowing Through Inheritance.** If the variable you want to hide is defined at class level, you can shadow it through inheritance by redeclaring it with the Shadows (Visual Basic) keyword in a derived class.

Two Ways to Hide a Variable

To hide a variable by shadowing it through scope

1. Determine the region defining the variable you want to hide, and determine a subregion in which to redefine it with your variable.

Variable's region	Allowable subregion for redefining it
Module	A class within the module
Class	A subclass within the class A procedure within the class

   You cannot redefine a procedure variable in a block within that procedure, for example in an **If**...**End If** construction or a **For** loop.

2. Create the subregion if it does not already exist.

3. Within the subregion, write a Dim Statement (Visual Basic) declaring the shadowing variable.

   When code inside the subregion refers to the variable name, the compiler resolves the reference to the shadowing variable.

   The following example illustrates shadowing through scope, as well as a reference that bypasses the shadowing.

```
Module shadowByScope
    ' The following statement declares num as a module-level variable.
    Public num As Integer
    Sub show()
        ' The following statement declares num as a local variable.
        Dim num As Integer
        ' The following statement sets the value of the local variable.
        num = 2
        ' The following statement displays the module-level variable.
        MsgBox(CStr(shadowByScope.num))
    End Sub
    Sub useModuleLevelNum()
        ' The following statement sets the value of the module-level variable.
        num = 1
        show()
    End Sub
End Module
```

The preceding example declares the variable `num` both at module level and at procedure level (in the procedure `show`). The local variable `num` shadows the module-level variable `num` within `show`, so the local variable is set to 2. However, there is no local variable to shadow `num` in the `useModuleLevelNum` procedure. Therefore, `useModuleLevelNum` sets the value of the module-level variable to 1.

The **MsgBox** call inside `show` bypasses the shadowing mechanism by qualifying `num` with the module name. Therefore, it displays the module-level variable instead of the local variable.

To hide a variable by shadowing it through inheritance

1. Be sure the variable you want to hide is declared in a class, and at class level (outside any procedure). Otherwise you cannot shadow it through inheritance.

2. Define a class derived from the variable's class if one does not already exist.

3. Inside the derived class, write a **Dim** statement declaring your variable. Include the Shadows (Visual Basic) keyword in the declaration.

   When code in the derived class refers to the variable name, the compiler resolves the reference to your variable.

   The following example illustrates shadowing through inheritance. It makes two references, one that accesses the shadowing variable and one that bypasses the shadowing.

```
Public Class shadowBaseClass
    Public shadowString As String = "This is the base class string."
End Class
Public Class shadowDerivedClass
    Inherits shadowBaseClass
    Public Shadows shadowString As String = "This is the derived class string."
    Public Sub showStrings()
        Dim s As String = "Unqualified shadowString: " & shadowString &
```

```
                vbCrLf & "MyBase.shadowString: " & MyBase.shadowString
            MsgBox(s)
        End Sub
    End Class
```

The preceding example declares the variable `shadowString` in the base class and shadows it in the derived class. The procedure `showStrings` in the derived class displays the shadowing version of the string when the name `shadowString` is not qualified. It then displays the shadowed version when `shadowString` is qualified with the **MyBase** keyword.

Robust Programming

Shadowing introduces more than one version of a variable with the same name. When a code statement refers to the variable name, the version to which the compiler resolves the reference depends on factors such as the location of the code statement and the presence of a qualifying string. This can increase the risk of referring to an unintended version of a shadowed variable. You can lower that risk by fully qualifying all references to a shadowed variable.

See Also

References to Declared Elements (Visual Basic)
Shadowing in Visual Basic
Differences Between Shadowing and Overriding (Visual Basic)
How to: Hide an Inherited Variable (Visual Basic)
How to: Access a Variable Hidden by a Derived Class (Visual Basic)
Overrides (Visual Basic)
Me, My, MyBase, and MyClass in Visual Basic
Inheritance Basics (Visual Basic)

How to: Hide an Inherited Variable (Visual Basic)

Visual Studio 2015

A derived class inherits all the definitions of its base class. If you want to define a variable using the same name as an element of the base class, you can hide, or *shadow*, that base class element when you define your variable in the derived class. If you do this, code in the derived class accesses your variable unless it explicitly bypasses the shadowing mechanism.

Another reason you might want to hide an inherited variable is to protect against base class revision. The base class might undergo a change that alters the element you are inheriting. If this happens, the **Shadows** modifier forces references from the derived class to be resolved to your variable, instead of to the base class element.

To hide an inherited variable

1. Be sure the variable you want to hide is declared at class level (outside any procedure). Otherwise you do not need to hide it.

2. Inside your derived class, write a Dim Statement (Visual Basic) declaring your variable. Use the same name as that of the inherited variable.

3. Include the Shadows (Visual Basic) keyword in the declaration.

   When code in the derived class refers to the variable name, the compiler resolves the reference to your variable.

   The following example illustrates shadowing of an inherited variable.

```
Public Class shadowBaseClass
    Public shadowString As String = "This is the base class string."
End Class
Public Class shadowDerivedClass
    Inherits shadowBaseClass
    Public Shadows shadowString As String = "This is the derived class string."
    Public Sub showStrings()
        Dim s As String = "Unqualified shadowString: " & shadowString &
            vbCrLf & "MyBase.shadowString: " & MyBase.shadowString
        MsgBox(s)
    End Sub
End Class
```

   The preceding example declares the variable `shadowString` in the base class and shadows it in the derived class. The procedure `showStrings` in the derived class displays the shadowing version of the string when the name `shadowString` is not qualified. It then displays the shadowed version when `shadowString` is qualified with the **MyBase** keyword.

Robust Programming

Shadowing introduces more than one version of a variable with the same name. When a code statement refers to the variable name, the version to which the compiler resolves the reference depends on factors such as the location of the code statement and the presence of a qualifying string. This can increase the risk of referring to an unintended version of a shadowed variable. You can lower that risk by fully qualifying all references to a shadowed variable.

See Also

References to Declared Elements (Visual Basic)
Shadowing in Visual Basic
Differences Between Shadowing and Overriding (Visual Basic)
How to: Hide a Variable with the Same Name as Your Variable (Visual Basic)
How to: Access a Variable Hidden by a Derived Class (Visual Basic)
Overrides (Visual Basic)
Me, My, MyBase, and MyClass in Visual Basic
Inheritance Basics (Visual Basic)

How to: Access a Variable Hidden by a Derived Class (Visual Basic)

Visual Studio 2015

When code in a derived class accesses a variable, the compiler normally resolves the reference to the closest accessible version, that is, the accessible version the fewest derivational steps backward from the accessing class. If the variable is defined in the derived class, the code normally accesses that definition.

If the derived class variable shadows a variable in the base class, it hides the base class version. However, you can access the base class variable by qualifying it with the **MyBase** keyword.

To access a base class variable hidden by a derived class

- In an expression or assignment statement, precede the variable name with the **MyBase** keyword and a period (.).

  The compiler resolves the reference to the base class version of the variable.

  The following example illustrates shadowing through inheritance. It makes two references, one that accesses the shadowing variable and one that bypasses the shadowing.

```
Public Class shadowBaseClass
    Public shadowString As String = "This is the base class string."
End Class
Public Class shadowDerivedClass
    Inherits shadowBaseClass
    Public Shadows shadowString As String = "This is the derived class string."
    Public Sub showStrings()
        Dim s As String = "Unqualified shadowString: " & shadowString &
            vbCrLf & "MyBase.shadowString: " & MyBase.shadowString
        MsgBox(s)
    End Sub
End Class
```

  The preceding example declares the variable `shadowString` in the base class and shadows it in the derived class. The procedure `showStrings` in the derived class displays the shadowing version of the string when the name `shadowString` is not qualified. It then displays the shadowed version when `shadowString` is qualified with the **MyBase** keyword.

Robust Programming

To lower the risk of referring to an unintended version of a shadowed variable, you can fully qualify all references to a shadowed variable. Shadowing introduces more than one version of a variable with the same name. When a code statement

refers to the variable name, the version to which the compiler resolves the reference depends on factors such as the location of the code statement and the presence of a qualifying string. This can increase the risk of referring to the wrong version of the variable.

See Also

- References to Declared Elements (Visual Basic)
- Shadowing in Visual Basic
- Differences Between Shadowing and Overriding (Visual Basic)
- How to: Hide a Variable with the Same Name as Your Variable (Visual Basic)
- How to: Hide an Inherited Variable (Visual Basic)
- Shadows (Visual Basic)
- Overrides (Visual Basic)
- Me, My, MyBase, and MyClass in Visual Basic
- Inheritance Basics (Visual Basic)

Inheritance Basics (Visual Basic)

Visual Studio 2015

The **Inherits** statement is used to declare a new class, called a *derived class*, based on an existing class, known as a *base class*. Derived classes inherit, and can extend, the properties, methods, events, fields, and constants defined in the base class. The following section describes some of the rules for inheritance, and the modifiers you can use to change the way classes inherit or are inherited:

- By default, all classes are inheritable unless marked with the **NotInheritable** keyword. Classes can inherit from other classes in your project or from classes in other assemblies that your project references.
- Unlike languages that allow multiple inheritance, Visual Basic allows only single inheritance in classes; that is, derived classes can have only one base class. Although multiple inheritance is not allowed in classes, classes can implement multiple interfaces, which can effectively accomplish the same ends.
- To prevent exposing restricted items in a base class, the access type of a derived class must be equal to or more restrictive than its base class. For example, a **Public** class cannot inherit a **Friend** or a **Private** class, and a **Friend** class cannot inherit a **Private** class.

Inheritance Modifiers

Visual Basic introduces the following class-level statements and modifiers to support inheritance:

- **Inherits** statement — Specifies the base class.
- **NotInheritable** modifier — Prevents programmers from using the class as a base class.
- **MustInherit** modifier — Specifies that the class is intended for use as a base class only. Instances of **MustInherit** classes cannot be created directly; they can only be created as base class instances of a derived class. (Other programming languages, such as C++ and C#, use the term *abstract class* to describe such a class.)

Overriding Properties and Methods in Derived Classes

By default, a derived class inherits properties and methods from its base class. If an inherited property or method has to behave differently in the derived class it can be *overridden*. That is, you can define a new implementation of the method in the derived class. The following modifiers are used to control how properties and methods are overridden:

- **Overridable** — Allows a property or method in a class to be overridden in a derived class.
- **Overrides** — Overrides an **Overridable** property or method defined in the base class.
- **NotOverridable** — Prevents a property or method from being overridden in an inheriting class. By default, **Public**

methods are **NotOverridable**.

- **MustOverride** — Requires that a derived class override the property or method. When the **MustOverride** keyword is used, the method definition consists of just the **Sub**, **Function**, or **Property** statement. No other statements are allowed, and specifically there is no **End Sub** or **End Function** statement. **MustOverride** methods must be declared in **MustInherit** classes.

Suppose you want to define classes to handle payroll. You could define a generic `Payroll` class that contains a `RunPayroll` method that calculates payroll for a typical week. You could then use `Payroll` as a base class for a more specialized `BonusPayroll` class, which could be used when distributing employee bonuses.

The `BonusPayroll` class can inherit, and override, the `PayEmployee` method defined in the base `Payroll` class.

The following example defines a base class, `Payroll`, and a derived class, `BonusPayroll`, which overrides an inherited method, `PayEmployee`. A procedure, `RunPayroll`, creates and then passes a `Payroll` object and a `BonusPayroll` object to a function, `Pay`, that executes the `PayEmployee` method of both objects.

VB

```
Const BonusRate As Decimal = 1.45D
Const PayRate As Decimal = 14.75D

Class Payroll
    Overridable Function PayEmployee(
        ByVal HoursWorked As Decimal,
        ByVal PayRate As Decimal) As Decimal

        PayEmployee = HoursWorked * PayRate
    End Function
End Class

Class BonusPayroll
    Inherits Payroll
    Overrides Function PayEmployee(
        ByVal HoursWorked As Decimal,
        ByVal PayRate As Decimal) As Decimal

        ' The following code calls the original method in the base
        ' class, and then modifies the returned value.
        PayEmployee = MyBase.PayEmployee(HoursWorked, PayRate) * BonusRate
    End Function
End Class

Sub RunPayroll()
    Dim PayrollItem As Payroll = New Payroll
    Dim BonusPayrollItem As New BonusPayroll
    Dim HoursWorked As Decimal = 40

    MsgBox("Normal pay is: " &
        PayrollItem.PayEmployee(HoursWorked, PayRate))
    MsgBox("Pay with bonus is: " &
        BonusPayrollItem.PayEmployee(HoursWorked, PayRate))
End Sub
```

The MyBase Keyword

The **MyBase** keyword behaves like an object variable that refers to the base class of the current instance of a class. **MyBase** is frequently used to access base class members that are overridden or shadowed in a derived class. In particular, `MyBase.New` is used to explicitly call a base class constructor from a derived class constructor.

For example, suppose you are designing a derived class that overrides a method inherited from the base class. The overridden method can call the method in the base class and modify the return value as shown in the following code fragment:

VB

```
Class DerivedClass
    Inherits BaseClass
    Public Overrides Function CalculateShipping(
```

```
        ByVal Dist As Double,
        ByVal Rate As Double) As Double

        ' Call the method in the base class and modify the return value.
        Return MyBase.CalculateShipping(Dist, Rate) * 2
    End Function
End Class
```

The following list describes restrictions on using **MyBase**:

- **MyBase** refers to the immediate base class and its inherited members. It cannot be used to access **Private** members in the class.
- **MyBase** is a keyword, not a real object. **MyBase** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- The method that **MyBase** qualifies does not have to be defined in the immediate base class; it may instead be defined in an indirectly inherited base class. In order for a reference qualified by **MyBase** to compile correctly, some base class must contain a method matching the name and types of parameters that appear in the call.
- You cannot use **MyBase** to call **MustOverride** base class methods.
- **MyBase** cannot be used to qualify itself. Therefore, the following code is not valid:

  ```
  MyBase.MyBase.BtnOK_Click()
  ```

- **MyBase** cannot be used in modules.
- **MyBase** cannot be used to access base class members that are marked as **Friend** if the base class is in a different assembly.

For more information and another example, see How to: Access a Variable Hidden by a Derived Class (Visual Basic).

The MyClass Keyword

The **MyClass** keyword behaves like an object variable that refers to the current instance of a class as originally implemented. **MyClass** resembles **Me**, but every method and property call on **MyClass** is treated as if the method or property were NotOverridable (Visual Basic). Therefore, the method or property is not affected by overriding in a derived class.

- **MyClass** is a keyword, not a real object. **MyClass** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- **MyClass** refers to the containing class and its inherited members.
- **MyClass** can be used as a qualifier for **Shared** members.
- **MyClass** cannot be used inside a **Shared** method, but can be used inside an instance method to access a shared member of a class.

- **MyClass** cannot be used in standard modules.
- **MyClass** can be used to qualify a method that is defined in a base class and that has no implementation of the method provided in that class. Such a reference has the same meaning as **MyBase.***Method*.

The following example compares **Me** and **MyClass**.

```
Class baseClass
    Public Overridable Sub testMethod()
        MsgBox("Base class string")
    End Sub
    Public Sub useMe()
        ' The following call uses the calling class's method, even if
        ' that method is an override.
        Me.testMethod()
    End Sub
    Public Sub useMyClass()
        ' The following call uses this instance's method and not any
        ' override.
        MyClass.testMethod()
    End Sub
End Class
Class derivedClass : Inherits baseClass
    Public Overrides Sub testMethod()
        MsgBox("Derived class string")
    End Sub
End Class
Class testClasses
    Sub startHere()
        Dim testObj As derivedClass = New derivedClass()
        ' The following call displays "Derived class string".
        testObj.useMe()
        ' The following call displays "Base class string".
        testObj.useMyClass()
    End Sub
End Class
```

Even though `derivedClass` overrides `testMethod`, the **MyClass** keyword in `useMyClass` nullifies the effects of overriding, and the compiler resolves the call to the base class version of `testMethod`.

See Also

Inherits Statement
Me, My, MyBase, and MyClass in Visual Basic

Declared Element Characteristics (Visual Basic)

Visual Studio 2015

A *characteristic* of a declared element is an aspect of that element that affects how code can interact with it. Every declared element has one or more of the following characteristics associated with it:

- *Data type* — the values the element can hold, and how it stores those values. For more information, see Data Type Summary (Visual Basic).
- *Lifetime* — the period of execution time during which the element is available for use. For more information, see Lifetime in Visual Basic.
- *Scope* — the set of all code that can refer to the element without qualifying its name. For more information, see How to: Control the Scope of a Variable (Visual Basic).
- *Access level* — the permission for code to make use of the element. For more information, see How to: Control the Availability of a Variable (Visual Basic).

Characteristics of the Elements

The following table shows the declared elements and the characteristics that apply to each one.

Element	Data Type	Lifetime	Scope [1]	Access Level
Variable	Yes	Yes	Yes	Yes
Constant	Yes	No	Yes	Yes
Enumeration	Yes	No	Yes	Yes
Structure	No	No	Yes	Yes
Property	Yes	Yes	Yes	Yes
Method	No	Yes	Yes	Yes
Procedure (**Sub** or **Function**)	No	Yes	Yes	Yes
Procedure parameter	Yes	Yes	Yes	No

Function return	Yes	Yes	Yes	No
Operator	Yes	No	Yes	Yes
Interface	No	No	Yes	Yes
Class	No	No	Yes	Yes
Event	No	No	Yes	Yes
Delegate	No	No	Yes	Yes

[1] Scope is sometimes referred to as *visibility*.

See Also

Declared Elements in Visual Basic
Declared Element Names (Visual Basic)
References to Declared Elements (Visual Basic)
Lifetime in Visual Basic
Scope in Visual Basic
Access Levels in Visual Basic
Data Types in Visual Basic
Variable Declaration in Visual Basic

Lifetime in Visual Basic

Visual Studio 2015

The *lifetime* of a declared element is the period of time during which it is available for use. Variables are the only elements that have lifetime. For this purpose, the compiler treats procedure parameters and function returns as special cases of variables. The lifetime of a variable represents the period of time during which it can hold a value. Its value can change over its lifetime, but it always holds some value.

Different Lifetimes

A *member variable* (declared at module level, outside any procedure) typically has the same lifetime as the element in which it is declared. A nonshared variable declared in a class or structure exists as a separate copy for each instance of the class or structure in which it is declared. Each such variable has the same lifetime as its instance. However, a **Shared** variable has only a single lifetime, which lasts for the entire time your application is running.

A *local variable* (declared inside a procedure) exists only while the procedure in which it is declared is running. This applies also to that procedure's parameters and to any function return. However, if that procedure calls other procedures, the local variables retain their values while the called procedures are running.

Beginning of Lifetime

A local variable's lifetime begins when control enters the procedure in which it is declared. Every local variable is initialized to the default value for its data type as soon as the procedure begins running. When the procedure encounters a **Dim** statement that specifies initial values, it sets those variables to those values, even if your code had already assigned other values to them.

Each member of a structure variable is initialized as if it were a separate variable. Similarly, each element of an array variable is initialized individually.

Variables declared within a block inside a procedure (such as a **For** loop) are initialized on entry to the procedure. These initializations take effect whether or not your code ever executes the block.

End of Lifetime

When a procedure terminates, the values of its local variables are not preserved, and Visual Basic reclaims their memory. The next time you call the procedure, all its local variables are created afresh and reinitialized.

When an instance of a class or structure terminates, its nonshared variables lose their memory and their values. Each new instance of the class or structure creates and reinitializes its nonshared variables. However, **Shared** variables are preserved until your application stops running.

Extension of Lifetime

If you declare a local variable with the **Static** keyword, its lifetime is longer than the execution time of its procedure. The following table shows how the procedure declaration determines how long a **Static** variable exists.

Procedure location and sharing	Static variable lifetime begins	Static variable lifetime ends
In a module (shared by default)	The first time the procedure is called	When your application stops running
In a class, **Shared** (procedure is not an instance member)	The first time the procedure is called either on a specific instance or on the class or structure name itself	When your application stops running
In an instance of a class, not **Shared** (procedure is an instance member)	The first time the procedure is called on the specific instance	When the instance is released for garbage collection (GC)

Static Variables of the Same Name

You can declare static variables with the same name in more than one procedure. If you do this, the Visual Basic compiler considers each such variable to be a separate element. The initialization of one of these variables does not affect the values of the others. The same applies if you define a procedure with a set of overloads and declare a static variable with the same name in each overload.

Containing Elements for Static Variables

You can declare a static local variable within a class, that is, inside a procedure in that class. However, you cannot declare a static local variable within a structure, either as a structure member or as a local variable of a procedure within that structure.

Example

Description

The following example declares a variable with the Static (Visual Basic) keyword. (Note that you do not need the **Dim** keyword when the Dim Statement (Visual Basic) uses a modifier such as **Static**.)

Code

VB

```
Function runningTotal(ByVal num As Integer) As Integer
    Static applesSold As Integer
    applesSold = applesSold + num
    Return applesSold
End Function
```

Comments

In the preceding example, the variable `applesSold` continues to exist after the procedure `runningTotal` returns to the calling code. The next time `runningTotal` is called, `applesSold` retains its previously calculated value.

If `applesSold` had been declared without using **Static**, the previous accumulated values would not be preserved across calls to `runningTotal`. The next time `runningTotal` was called, `applesSold` would have been recreated and initialized to 0, and `runningTotal` would have simply returned the same value with which it was called.

Compiling the Code

You can initialize the value of a static local variable as part of its declaration. If you declare an array to be **Static**, you can initialize its rank (number of dimensions), the length of each dimension, and the values of the individual elements.

Security

In the preceding example, you can produce the same lifetime by declaring `applesSold` at module level. If you changed the scope of a variable this way, however, the procedure would no longer have exclusive access to it. Because other procedures could access `applesSold` and change its value, the running total could be unreliable and the code could be more difficult to maintain.

See Also

Shared (Visual Basic)
Nothing (Visual Basic)
Declared Element Names (Visual Basic)
References to Declared Elements (Visual Basic)
Scope in Visual Basic
Access Levels in Visual Basic
Variables in Visual Basic
Variable Declaration in Visual Basic
Troubleshooting Data Types (Visual Basic)
Static (Visual Basic)

Scope in Visual Basic

Visual Studio 2015

The *scope* of a declared element is the set of all code that can refer to it without qualifying its name or making it available through an Imports Statement (.NET Namespace and Type). An element can have scope at one of the following levels:

Level	Description
Block scope	Available only within the code block in which it is declared
Procedure scope	Available to all code within the procedure in which it is declared
Module scope	Available to all code within the module, class, or structure in which it is declared
Namespace scope	Available to all code in the namespace in which it is declared

These levels of scope progress from the narrowest (block) to the widest (namespace), where *narrowest scope* means the smallest set of code that can refer to the element without qualification. For more information, see "Levels of Scope" on this page.

Specifying Scope and Defining Variables

You specify the scope of an element when you declare it. The scope can depend on the following factors:

- The region (block, procedure, module, class, or structure) in which you declare the element
- The namespace containing the element's declaration
- The access level you declare for the element

Use care when you define variables with the same name but different scope, because doing so can lead to unexpected results. For more information, see References to Declared Elements (Visual Basic).

Levels of Scope

A programming element is available throughout the region in which you declare it. All code in the same region can refer to the element without qualifying its name.

Block Scope

A block is a set of statements enclosed within initiating and terminating declaration statements, such as the following:

- **Do** and **Loop**
- **For** [**Each**] and **Next**
- **If** and **End If**
- **Select** and **End Select**
- **SyncLock** and **End SyncLock**
- **Try** and **End Try**
- **While** and **End While**
- **With** and **End With**

If you declare a variable within a block, you can use it only within that block. In the following example, the scope of the integer variable `cube` is the block between **If** and **End If**, and you can no longer refer to `cube` when execution passes out of the block.

```
If n < 1291 Then
    Dim cube As Integer
    cube = n ^ 3
End If
```

Note

Even if the scope of a variable is limited to a block, its lifetime is still that of the entire procedure. If you enter the block more than once during the procedure, each block variable retains its previous value. To avoid unexpected results in such a case, it is wise to initialize block variables at the beginning of the block.

Procedure Scope

An element declared within a procedure is not available outside that procedure. Only the procedure that contains the declaration can use it. Variables at this level are also known as *local variables*. You declare them with the Dim Statement (Visual Basic), with or without the Static (Visual Basic) keyword.

Procedure and block scope are closely related. If you declare a variable inside a procedure but outside any block within that procedure, you can think of the variable as having block scope, where the block is the entire procedure.

Note

All local elements, even if they are **Static** variables, are private to the procedure in which they appear. You cannot declare any element using the Public (Visual Basic) keyword within a procedure.

Module Scope

For convenience, the single term *module level* applies equally to modules, classes, and structures. You can declare elements at this level by placing the declaration statement outside of any procedure or block but within the module, class, or structure.

When you make a declaration at the module level, the access level you choose determines the scope. The namespace that contains the module, class, or structure also affects the scope.

Elements for which you declare Private (Visual Basic) access level are available to every procedure in that module, but not to any code in a different module. The **Dim** statement at module level defaults to **Private** if you do not use any access level keywords. However, you can make the scope and access level more obvious by using the **Private** keyword in the **Dim** statement.

In the following example, all procedures defined in the module can refer to the string variable `strMsg`. When the second procedure is called, it displays the contents of the string variable `strMsg` in a dialog box.

```
' Put the following declaration at module level (not in any procedure).
Private strMsg As String
' Put the following Sub procedure in the same module.
Sub initializePrivateVariable()
    strMsg = "This variable cannot be used outside this module."
End Sub
' Put the following Sub procedure in the same module.
Sub usePrivateVariable()
    MsgBox(strMsg)
End Sub
```

Namespace Scope

If you declare an element at module level using the Friend (Visual Basic) or Public (Visual Basic) keyword, it becomes available to all procedures throughout the namespace in which the element is declared. With the following alteration to the preceding example, the string variable `strMsg` can be referred to by code anywhere in the namespace of its declaration.

```
' Include this declaration at module level (not inside any procedure).
Public strMsg As String
```

Namespace scope includes nested namespaces. An element available from within a namespace is also available from within any namespace nested inside that namespace.

If your project does not contain any Namespace Statements, everything in the project is in the same namespace. In this case, namespace scope can be thought of as project scope. **Public** elements in a module, class, or structure are also available to any project that references their project.

Choice of Scope

When you declare a variable, you should keep in mind the following points when choosing its scope.

Advantages of Local Variables

Local variables are a good choice for any kind of temporary calculation, for the following reasons:

- **Name Conflict Avoidance.** Local variable names are not susceptible to conflict. For example, you can create several different procedures containing a variable called `intTemp`. As long as each `intTemp` is declared as a local variable, each procedure recognizes only its own version of `intTemp`. Any one procedure can alter the value in its local `intTemp` without affecting `intTemp` variables in other procedures.
- **Memory Consumption.** Local variables consume memory only while their procedure is running. Their memory is released when the procedure returns to the calling code. By contrast, Shared (Visual Basic) and Static (Visual Basic) variables consume memory resources until your application stops running, so use them only when necessary. *Instance variables* consume memory while their instance continues to exist, which makes them less efficient than local variables, but potentially more efficient than **Shared** or **Static** variables.

Minimizing Scope

In general, when declaring any variable or constant, it is good programming practice to make the scope as narrow as possible (block scope is the narrowest). This helps conserve memory and minimizes the chances of your code erroneously referring to the wrong variable. Similarly, you should declare a variable to be Static (Visual Basic) only when it is necessary to preserve its value between procedure calls.

See Also

Declared Element Characteristics (Visual Basic)
How to: Control the Scope of a Variable (Visual Basic)
Lifetime in Visual Basic
Access Levels in Visual Basic
References to Declared Elements (Visual Basic)
Variable Declaration in Visual Basic

How to: Control the Scope of a Variable (Visual Basic)

Visual Studio 2015

Normally, a variable is in *scope*, or visible for reference, throughout the region in which you declare it. In some cases, the variable's *access level* can influence its scope.

For more information, see Scope in Visual Basic.

Scope at Block or Procedure Level

To make a variable visible only within a block

- Place the Dim Statement (Visual Basic) for the variable between the initiating and terminating declaration statements of that block, for example between the **For** and **Next** statements of a **For** loop.

  You can refer to the variable only from within the block.

To make a variable visible only within a procedure

- Place the **Dim** statement for the variable inside the procedure but outside any block (such as a **With**...**End With** block).

  You can refer to the variable only from within the procedure, including inside any block contained in the procedure.

Scope at Module or Namespace Level

For convenience, the single term *module level* applies equally to modules, classes, and structures. The access level of a module level variable determines its scope. The namespace that contains the module, class, or structure also influences the scope.

To make a variable visible throughout a module, class, or structure

1. Place the **Dim** statement for the variable inside the module, class, or structure, but outside any procedure.
2. Include the Private (Visual Basic) keyword in the **Dim** statement.
3. You can refer to the variable from anywhere within the module, class, or structure, but not from outside it.

To make a variable visible throughout a namespace

1. Place the **Dim** statement for the variable inside the module, class, or structure, but outside any procedure.
2. Include the Friend (Visual Basic) or Public (Visual Basic) keyword in the **Dim** statement.
3. You can refer to the variable from anywhere within the namespace containing the module, class, or structure.

Example

The following example declares a variable at module level and limits its visibility to code within the module.

```
Module demonstrateScope
    Private strMsg As String
    Sub initializePrivateVariable()
        strMsg = "This variable cannot be used outside this module."
    End Sub
    Sub usePrivateVariable()
        MsgBox(strMsg)
    End Sub
End Module
```

In the preceding example, all the procedures defined in module `demonstrateScope` can refer to the **String** variable `strMsg`. When the `usePrivateVariable` procedure is called, it displays the contents of the string variable `strMsg` in a dialog box.

With the following alteration to the preceding example, the string variable `strMsg` can be referred to by code anywhere in the namespace of its declaration.

```
Public strMsg As String
```

Robust Programming

The narrower the scope of a variable, the fewer opportunities you have for accidentally referring to it in place of another variable with the same name. You can also minimize problems of reference matching.

.NET Framework Security

The narrower the scope of a variable, the smaller the chances that malicious code can make improper use of it.

See Also

Scope in Visual Basic
Lifetime in Visual Basic
Access Levels in Visual Basic
Variables in Visual Basic
Variable Declaration in Visual Basic

Dim Statement (Visual Basic)

Type Promotion (Visual Basic)

Visual Studio 2015

When you declare a programming element in a module, Visual Basic promotes its scope to the namespace containing the module. This is known as *type promotion*.

The following example shows a skeleton definition of a module and two members of that module.

VB

```
Namespace projNamespace
    Module projModule
        Public Enum basicEnum As Integer
            one = 1
            two = 2
        End Enum
        Public Class innerClass
            Shared Sub numberSub(ByVal firstArg As Integer)
            End Sub
        End Class
    End Module
End Namespace
```

Within `projModule`, programming elements declared at module level are promoted to `projNamespace`. In the preceding example, `basicEnum` and `innerClass` are promoted, but `numberSub` is not, because it is not declared at module level.

Effect of Type Promotion

The effect of type promotion is that a qualification string does not need to include the module name. The following example makes two calls to the procedure in the preceding example.

VB

```
Sub usePromotion()

projNamespace.projModule.innerClass.numberSub(projNamespace.projModule.basicEnum.one)
    projNamespace.innerClass.numberSub(projNamespace.basicEnum.two)
End Sub
```

In the preceding example, the first call uses complete qualification strings. However, this is not necessary because of type promotion. The second call also accesses the module's members without including `projModule` in the qualification strings.

Defeat of Type Promotion

If the namespace already has a member with the same name as a module member, type promotion is defeated for that module member. The following example shows a skeleton definition of an enumeration and a module within the same namespace.

VB

```
Namespace thisNamespace
    Public Enum abc
        first = 1
        second
    End Enum
    Module thisModule
        Public Class abc
            Public Sub abcSub()
            End Sub
        End Class
        Public Class xyz
            Public Sub xyzSub()
            End Sub
        End Class
    End Module
End Namespace
```

In the preceding example, Visual Basic cannot promote class `abc` to `thisNameSpace` because there is already an enumeration with the same name at namespace level. To access `abcSub`, you must use the full qualification string `thisNamespace.thisModule.abc.abcSub`. However, class `xyz` is still promoted, and you can access `xyzSub` with the shorter qualification string `thisNamespace.xyz.xyzSub`.

Defeat of Type Promotion for Partial Types

If a class or structure inside a module uses the Partial (Visual Basic) keyword, type promotion is automatically defeated for that class or structure, whether or not the namespace has a member with the same name. Other elements in the module are still eligible for type promotion.

Consequences. Defeat of type promotion of a partial definition can cause unexpected results and even compiler errors. The following example shows skeleton partial definitions of a class, one of which is inside a module.

VB

```
Namespace sampleNamespace
    Partial Public Class sampleClass
        Public Sub sub1()
        End Sub
    End Class
    Module sampleModule
        Partial Public Class sampleClass
            Public Sub sub2()
            End Sub
        End Class
```

```
        End Module
    End Namespace
```

In the preceding example, the developer might expect the compiler to merge the two partial definitions of `sampleClass`. However, the compiler does not consider promotion for the partial definition inside `sampleModule`. As a result, it attempts to compile two separate and distinct classes, both named `sampleClass` but with different qualification paths.

The compiler merges partial definitions only when their fully qualified paths are identical.

Recommendations

The following recommendations represent good programming practice.

- **Unique Names.** When you have full control over the naming of programming elements, it is always a good idea to use unique names everywhere. Identical names require extra qualification and can make your code harder to read. They can also lead to subtle errors and unexpected results.
- **Full Qualification.** When you are working with modules and other elements in the same namespace, the safest approach is to always use full qualification for all programming elements. If type promotion is defeated for a module member and you do not fully qualify that member, you could inadvertently access a different programming element.

See Also

Module Statement
Namespace Statement
Partial (Visual Basic)
Scope in Visual Basic
How to: Control the Scope of a Variable (Visual Basic)
References to Declared Elements (Visual Basic)