Declared Elements in Visual Basic

Visual Studio 2015

A *declared element* is a programming element that is defined in a declaration statement. Declared elements include variables, constants, enumerations, classes, structures, modules, interfaces, procedures, procedure parameters, function returns, external procedure references, operators, properties, events, and delegates.

Declaration statements include the following:

- Dim Statement (Visual Basic)
- Const Statement (Visual Basic)
- Enum Statement (Visual Basic)
- Class Statement (Visual Basic)
- Structure Statement
- Module Statement
- Interface Statement (Visual Basic)
- Function Statement (Visual Basic)
- Sub Statement (Visual Basic)
- Declare Statement
- Operator Statement
- Property Statement
- Event Statement
- Delegate Statement

In This Section

Declared Element Names (Visual Basic)
: Describes how to name elements and use alphabetic case.

Declared Element Characteristics (Visual Basic)
: Covers characteristics, such as scope, possessed by declared elements.

References to Declared Elements (Visual Basic)
: Describes how the compiler matches a reference to a declaration and how to qualify a name.

Related Sections

Program Structure and Code Conventions (Visual Basic)

Presents guidelines for making your code easier to read, understand, and maintain.

Statements (Visual Basic)

Describes statements that name and define procedures, variables, arrays, and constants.

Declaration Contexts and Default Access Levels (Visual Basic)

Lists the types of declared elements and shows for each one its declaration statement, in what context you can declare it, and its default access level.

Dim Statement (Visual Basic)

Visual Studio 2015

Declares and allocates storage space for one or more variables.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [[ Shared ] [ Shadows ] | [ Static ]]
[ ReadOnly ]
Dim [ WithEvents ] variablelist
```

Parts

- *attributelist*

  Optional. See Attribute List.

- *accessmodifier*

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - **Protected Friend**

  See Access Levels in Visual Basic.

- **Shared**

  Optional. See Shared.

- **Shadows**

  Optional. See Shadows.

- **Static**

Optional. See Static.

- **ReadOnly**

  Optional. See ReadOnly.

- **WithEvents**

  Optional. Specifies that these are object variables that refer to instances of a class that can raise events. See WithEvents.

- *variablelist*

  Required. List of variables being declared in this statement.

  ```
  variable [ , variable ... ]
  ```

  Each *variable* has the following syntax and parts:

  ```
  variablename [ ( [ boundslist ] ) ] [ As [ New ] datatype
  [ With { [ .propertyname = propinitializer [ , ... ] ] } ] ] [ = initializer ]
  ```

Part	Description
variablename	Required. Name of the variable. See Declared Element Names (Visual Basic).
boundslist	Optional. List of bounds of each dimension of an array variable.
New	Optional. Creates a new instance of the class when the **Dim** statement runs.
datatype	Optional. Data type of the variable.
With	Optional. Introduces the object initializer list.
propertyname	Optional. The name of a property in the class you are making an instance of.
propinitializer	Required after *propertyname* =. The expression that is evaluated and assigned to the property name.
initializer	Optional if **New** is not specified. Expression that is evaluated and assigned to the variable when it is created.

Remarks

The Visual Basic compiler uses the **Dim** statement to determine the variable's data type and other information, such as what code can access the variable. The following example declares a variable to hold an **Integer** value.

VB

```
Dim numberOfStudents As Integer
```

You can specify any data type or the name of an enumeration, structure, class, or interface.

VB

```
Dim finished As Boolean
Dim monitorBox As System.Windows.Forms.Form
```

For a reference type, you use the **New** keyword to create a new instance of the class or structure that is specified by the data type. If you use **New**, you do not use an initializer expression. Instead, you supply arguments, if they are required, to the constructor of the class from which you are creating the variable.

VB

```
Dim bottomLabel As New System.Windows.Forms.Label
```

You can declare a variable in a procedure, block, class, structure, or module. You cannot declare a variable in a source file, namespace, or interface. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

A variable that is declared at module level, outside any procedure, is a *member variable* or *field*. Member variables are in scope throughout their class, structure, or module. A variable that is declared at procedure level is a *local variable*. Local variables are in scope only within their procedure or block.

The following access modifiers are used to declare variables outside a procedure: **Public**, **Protected**, **Friend**, **Protected Friend**, and **Private**. For more information, see Access Levels in Visual Basic.

The **Dim** keyword is optional and usually omitted if you specify any of the following modifiers: **Public**, **Protected**, **Friend**, **Protected Friend**, **Private**, **Shared**, **Shadows**, **Static**, **ReadOnly**, or **WithEvents**.

VB

```
Public maximumAllowed As Double
Protected Friend currentUserName As String
Private salary As Decimal
Static runningTotal As Integer
```

If **Option Explicit** is on (the default), the compiler requires a declaration for every variable you use. For more information, see Option Explicit Statement (Visual Basic).

Specifying an Initial Value

You can assign a value to a variable when it is created. For a value type, you use an *initializer* to supply an expression to be assigned to the variable. The expression must evaluate to a constant that can be calculated at compile time.

VB

```
Dim quantity As Integer = 10
Dim message As String = "Just started"
```

If an initializer is specified and a data type is not specified in an **As** clause, *type inference* is used to infer the data type from the initializer. In the following example, both `num1` and `num2` are strongly typed as integers. In the second declaration, type inference infers the type from the value 3.

VB

```
' Use explicit typing.
Dim num1 As Integer = 3

' Use local type inference.
Dim num2 = 3
```

Type inference applies at the procedure level. It does not apply outside a procedure in a class, structure, module, or interface. For more information about type inference, see Option Infer Statement and Local Type Inference (Visual Basic).

For information about what happens when a data type or initializer is not specified, see Default Data Types and Values later in this topic.

You can use an *object initializer* to declare instances of named and anonymous types. The following code creates an instance of a `Student` class and uses an object initializer to initialize properties.

VB

```
Dim student1 As New Student With {.First = "Michael",
                                  .Last = "Tucker"}
```

For more information about object initializers, see How to: Declare an Object by Using an Object Initializer (Visual Basic), Object Initializers: Named and Anonymous Types (Visual Basic), and Anonymous Types (Visual Basic).

Declaring Multiple Variables

You can declare several variables in one declaration statement, specifying the variable name for each one, and following each array name with parentheses. Multiple variables are separated by commas.

VB

```
Dim lastTime, nextTime, allTimes() As Date
```

If you declare more than one variable with one **As** clause, you cannot supply an initializer for that group of variables.

You can specify different data types for different variables by using a separate **As** clause for each variable you declare. Each variable takes the data type specified in the first **As** clause encountered after its *variablename* part.

VB

```
Dim a, b, c As Single, x, y As Double, i As Integer
' a, b, and c are all Single; x and y are both Double
```

Arrays

You can declare a variable to hold an *array*, which can hold multiple values. To specify that a variable holds an array, follow its *variablename* immediately with parentheses. For more information about arrays, see Arrays in Visual Basic.

You can specify the lower and upper bound of each dimension of an array. To do this, include a *boundslist* inside the parentheses. For each dimension, the *boundslist* specifies the upper bound and optionally the lower bound. The lower bound is always zero, whether you specify it or not. Each index can vary from zero through its upper bound value.

The following two statements are equivalent. Each statement declares an array of 21 **Integer** elements. When you access the array, the index can vary from 0 through 20.

VB

```
Dim totals(20) As Integer
Dim totals(0 To 20) As Integer
```

The following statement declares a two-dimensional array of type **Double**. The array has 4 rows (3 + 1) of 6 columns (5 + 1) each. Note that an upper bound represents the highest possible value for the index, not the length of the dimension. The length of the dimension is the upper bound plus one.

VB

```
Dim matrix2(3, 5) As Double
```

An array can have from 1 to 32 dimensions.

You can leave all the bounds blank in an array declaration. If you do this, the array has the number of dimensions you specify, but it is uninitialized. It has a value of **Nothing** until you initialize at least some of its elements. The **Dim** statement must specify bounds either for all dimensions or for no dimensions.

VB

```
' Declare an array with blank array bounds.
Dim messages() As String
' Initialize the array.
ReDim messages(4)
```

If the array has more than one dimension, you must include commas between the parentheses to indicate the number of dimensions.

VB

```
Dim oneDimension(), twoDimensions(,), threeDimensions(,,) As Byte
```

You can declare a *zero-length array* by declaring one of the array's dimensions to be -1. A variable that holds a zero-length array does not have the value **Nothing**. Zero-length arrays are required by certain common language runtime functions. If you try to access such an array, a runtime exception occurs. For more information, see Arrays in Visual Basic.

You can initialize the values of an array by using an array literal. To do this, surround the initialization values with braces (**{}**).

VB

```
Dim longArray() As Long = {0, 1, 2, 3}
```

For multidimensional arrays, the initialization for each separate dimension is enclosed in braces in the outer dimension. The elements are specified in row-major order.

VB

```
Dim twoDimensions(,) As Integer = {{0, 1, 2}, {10, 11, 12}}
```

For more information about array literals, see Arrays in Visual Basic.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a **Dim** statement.

Data type specified?	Initializer specified?	Example	Result
No	No	`Dim qty`	If Option Strict is off (the default), the variable is set to **Nothing**. If **Option Strict** is on, a compile-time error occurs.

No	Yes	`Dim qty = 5`	If Option Infer is on (the default), the variable takes the data type of the initializer. See Local Type Inference (Visual Basic). If **Option Infer** is off and **Option Strict** is off, the variable takes the data type of **Object**. If **Option Infer** is off and **Option Strict** is on, a compile-time error occurs.
Yes	No	`Dim qty As Integer`	The variable is initialized to the default value for the data type. See the table later in this section.
Yes	Yes	`Dim qty As Integer = 5`	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

If you specify a data type but do not specify an initializer, Visual Basic initializes the variable to the default value for its data type. The following table shows the default initialization values.

Data type	Default value
All numeric types (including **Byte** and **SByte**)	0
Char	Binary 0
All reference types (including **Object**, **String**, and all arrays)	**Nothing**
Boolean	**False**
Date	12:00 AM of January 1 of the year 1 (01/01/0001 12:00:00 AM)

Each element of a structure is initialized as if it were a separate variable. If you declare the length of an array but do not initialize its elements, each element is initialized as if it were a separate variable.

Static Local Variable Lifetime

A **Static** local variable has a longer lifetime than that of the procedure in which it is declared. The boundaries of the variable's lifetime depend on where the procedure is declared and whether it is **Shared**.

Procedure declaration	Variable initialized	Variable stops existing
In a module	The first time the procedure is called	When your program stops

		execution
In a class or structure, procedure is **Shared**	The first time the procedure is called either on a specific instance or on the class or structure itself	When your program stops execution
In a class or structure, procedure isn't **Shared**	The first time the procedure is called on a specific instance	When the instance is released for garbage collection (GC)

Attributes and Modifiers

You can apply attributes only to member variables, not to local variables. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local variables.

At module level, you cannot use the **Static** modifier to declare member variables. At procedure level, you cannot use **Shared**, **Shadows**, **ReadOnly**, **WithEvents**, or any access modifiers to declare local variables.

You can specify what code can access a variable by supplying an *accessmodifier*. Class and module member variables (outside any procedure) default to private access, and structure member variables default to public access. You can adjust their access levels with the access modifiers. You cannot use access modifiers on local variables (inside a procedure).

You can specify **WithEvents** only on member variables, not on local variables inside a procedure. If you specify **WithEvents**, the data type of the variable must be a specific class type, not **Object**. You cannot declare an array with **WithEvents**. For more information about events, see Events (Visual Basic).

Note
Code outside a class, structure, or module must qualify a member variable's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local variables within that procedure or block.

Releasing Managed Resources

The .NET Framework garbage collector disposes of managed resources without any extra coding on your part. However, you can force the disposal of a managed resource instead of waiting for the garbage collector.

If a class holds onto a particularly valuable and scarce resource (such as a database connection or file handle), you might not want to wait until the next garbage collection to clean up a class instance that's no longer in use. A class may implement the IDisposable interface to provide a way to release resources before a garbage collection. A class that implements that interface exposes a **Dispose** method that can be called to force valuable resources to be released immediately.

The **Using** statement automates the process of acquiring a resource, executing a set of statements, and then disposing of the resource. However, the resource must implement the IDisposable interface. For more information, see Using Statement (Visual Basic).

Example

The following example declares variables by using the **Dim** statement with various options.

VB

```
' Declare and initialize a Long variable.
Dim startingAmount As Long = 500

' Declare a variable that refers to a Button object,
' create a Button object, and assign the Button object
' to the variable.
Dim switchButton As New System.Windows.Forms.Button

' Declare a local variable that always retains its value,
' even after its procedure returns to the calling code.
Static totalSales As Double

' Declare a variable that refers to an array.
Dim highTemperature(31) As Integer

' Declare and initialize an array variable that
' holds four Boolean check values.
Dim checkValues() As Boolean = {False, False, True, False}
```

Example

The following example lists the prime numbers between 1 and 30. The scope of local variables is described in code comments.

VB

```
Public Sub ListPrimes()
    ' The sb variable can be accessed only
    ' within the ListPrimes procedure.
    Dim sb As New System.Text.StringBuilder()
```

```
        ' The number variable can be accessed only
        ' within the For...Next block.  A different
        ' variable with the same name could be declared
        ' outside of the For...Next block.
        For number As Integer = 1 To 30
            If CheckIfPrime(number) = True Then
                sb.Append(number.ToString & " ")
            End If
        Next

        Debug.WriteLine(sb.ToString)
        ' Output: 2 3 5 7 11 13 17 19 23 29
    End Sub


    Private Function CheckIfPrime(ByVal number As Integer) As Boolean
        If number < 2 Then
            Return False
        Else
            ' The root and highCheck variables can be accessed
            ' only within the Else block.  Different variables
            ' with the same names could be declared outside of
            ' the Else block.
            Dim root As Double = Math.Sqrt(number)
            Dim highCheck As Integer = Convert.ToInt32(Math.Truncate(root))

            ' The div variable can be accessed only within
            ' the For...Next block.
            For div As Integer = 2 To highCheck
                If number Mod div = 0 Then
                    Return False
                End If
            Next

            Return True
        End If
    End Function
```

Example

In the following example, the `speedValue` variable is declared at the class level. The **Private** keyword is used to declare the variable. The variable can be accessed by any procedure in the `Car` class.

VB

```
' Create a new instance of a Car.
Dim theCar As New Car()
theCar.Accelerate(30)
theCar.Accelerate(20)
theCar.Accelerate(-5)

Debug.WriteLine(theCar.Speed.ToString)
' Output: 45
```

VB

```
Public Class Car
    ' The speedValue variable can be accessed by
    ' any procedure in the Car class.
    Private speedValue As Integer = 0

    Public ReadOnly Property Speed() As Integer
        Get
            Return speedValue
        End Get
    End Property

    Public Sub Accelerate(ByVal speedIncrease As Integer)
        speedValue += speedIncrease
    End Sub
End Class
```

See Also

Const Statement (Visual Basic)
ReDim Statement (Visual Basic)
Option Explicit Statement (Visual Basic)
Option Infer Statement
Option Strict Statement
Compile Page, Project Designer (Visual Basic)
Variable Declaration in Visual Basic
Arrays in Visual Basic
Object Initializers: Named and Anonymous Types (Visual Basic)
Anonymous Types (Visual Basic)
Object Initializers: Named and Anonymous Types (Visual Basic)
How to: Declare an Object by Using an Object Initializer (Visual Basic)
Local Type Inference (Visual Basic)

Const Statement (Visual Basic)

Visual Studio 2015

Declares and defines one or more constants.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ]
Const constantlist
```

Parts

attributelist
Optional. List of attributes that apply to all the constants declared in this statement. See Attribute List (Visual Basic) in angle brackets ("<" and ">").

accessmodifier
Optional. Use this to specify what code can access these constants. Can be Public (Visual Basic), Protected (Visual Basic), Friend (Visual Basic), **Protected Friend**, or Private (Visual Basic).

Shadows
Optional. Use this to redeclare and hide a programming element in a base class. See Shadows.

constantlist
Required. List of constants being declared in this statement.

constant [, *constant* ...]

Each *constant* has the following syntax and parts:

constantname [As *datatype*] = *initializer*

Part	Description
constantname	Required. Name of the constant. See Declared Element Names (Visual Basic).
datatype	Required if **Option Strict** is **On**. Data type of the constant.
initializer	Required. Expression that is evaluated at compile time and assigned to the constant.

Remarks

If you have a value that never changes in your application, you can define a named constant and use it in place of a literal value. A name is easier to remember than a value. You can define the constant just once and use it in many places in your code. If in a later version you need to redefine the value, the **Const** statement is the only place you need to make a change.

You can use **Const** only at module or procedure level. This means the *declaration context* for a variable must be a class, structure, module, procedure, or block, and cannot be a source file, namespace, or interface. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Local constants (inside a procedure) default to public access, and you cannot use any access modifiers on them. Class and module member constants (outside any procedure) default to private access, and structure member constants default to public access. You can adjust their access levels with the access modifiers.

Rules

- **Declaration Context.** A constant declared at module level, outside any procedure, is a *member constant*; it is a member of the class, structure, or module that declares it.

A constant declared at procedure level is a *local constant*; it is local to the procedure or block that declares it.

- **Attributes.** You can apply attributes only to member constants, not to local constants. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local constants.

- **Modifiers.** By default, all constants are **Shared**, **Static**, and **ReadOnly**. You cannot use any of these keywords when declaring a constant.

  At procedure level, you cannot use **Shadows** or any access modifiers to declare local constants.

- **Multiple Constants.** You can declare several constants in the same declaration statement, specifying the *constantname* part for each one. Multiple constants are separated by commas.

Data Type Rules

- **Data Types.** The **Const** statement can declare the data type of a variable. You can specify any data type or the name of an enumeration.

- **Default Type.** If you do not specify *datatype*, the constant takes the data type of *initializer*. If you specify both *datatype* and *initializer*, the data type of *initializer* must be convertible to *datatype*. If neither *datatype* nor *initializer* is present, the data type defaults to **Object**.

- **Different Types.** You can specify different data types for different constants by using a separate **As** clause for each variable you declare. However, you cannot declare several constants to be of the same type by using a common **As** clause.

- **Initialization.** You must initialize the value of every constant in *constantlist*. You use *initializer* to supply an expression to be assigned to the constant. The expression can be any combination of literals, other constants that are already defined, and enumeration members that are already defined. You can use arithmetic and logical operators to combine such elements.

  You cannot use variables or functions in *initializer*. However, you can use conversion keywords such as **CByte** and **CShort**. You can also use **AscW** if you call it with a constant **String** or **Char** argument, since that can be evaluated at compile time.

Behavior

- **Scope.** Local constants are accessible only from within their procedure or block. Member constants are accessible from anywhere within their class, structure, or module.

- **Qualification.** Code outside a class, structure, or module must qualify a member constant's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local constants within that procedure or block.

Example

The following example uses the **Const** statement to declare constants for use in place of literal values.

VB

```
' The following statements declare constants.
Const maximum As Long = 459
Public Const helpString As String = "HELP"
Private Const startValue As Integer = 5
```

Example

If you define a constant with data type **Object**, the Visual Basic compiler gives it the type of *initializer*, instead of **Object**. In the following example, the constant `naturalLogBase` has the run-time type **Decimal**.

VB

```
Const naturalLogBase As Object = CDec(2.7182818284)
MsgBox("Run-time type of constant naturalLogBase is " &
    naturalLogBase.GetType.ToString())
```

The preceding example uses the ToString method on the Type object returned by the GetType Operator (Visual Basic), because Type cannot be converted to **String** using **CStr**.

See Also

Asc
AscW
Enum Statement (Visual Basic)
#Const Directive
Dim Statement (Visual Basic)
ReDim Statement (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)
Constants and Enumerations in Visual Basic
Constants and Enumerations (Visual Basic)
Type Conversion Functions (Visual Basic)

Enum Statement (Visual Basic)

Visual Studio 2015

Declares an enumeration and defines the values of its members.

Syntax

```
[ <attributelist> ] [ accessmodifier ]  [ Shadows ]
Enum enumerationname [ As datatype ]
   memberlist
End Enum
```

Parts

- *attributelist*

  Optional. List of attributes that apply to this enumeration. You must enclose the attribute list in angle brackets ("**<**" and "**>**").

  The FlagsAttribute attribute indicates that the value of an instance of the enumeration can include multiple enumeration members, and that each member represents a bit field in the enumeration value.

- *accessmodifier*

  Optional. Specifies what code can access this enumeration. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private

  You can specify **Protected Friend** to allow access from code within the enumeration's class, a derived class, or the same assembly.

- **Shadows**

  Optional. Specifies that this enumeration redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can specify Shadows only on the enumeration itself, not on any of its members.

- *enumerationname*

  Required. Name of the enumeration. For information on valid names, see Declared Element Names (Visual Basic).

- *datatype*

  Optional. Data type of the enumeration and all its members.

- *memberlist*

  Required. List of member constants being declared in this statement. Multiple members appear on individual source code lines.

  Each *member* has the following syntax and parts: `[<attribute list>] member name [ = initializer ]`

Part	Description
membername	Required. Name of this member.
initializer	Optional. Expression that is evaluated at compile time and assigned to this member.

- **End Enum**

  Terminates the **Enum** block.

Remarks

If you have a set of unchanging values that are logically related to each other, you can define them together in an enumeration. This provides meaningful names for the enumeration and its members, which are easier to remember than their values. You can then use the enumeration members in many places in your code.

The benefits of using enumerations include the following:

- Reduces errors caused by transposing or mistyping numbers.
- Makes it easy to change values in the future.
- Makes code easier to read, which means it is less likely that errors will be introduced.
- Ensures forward compatibility. If you use enumerations, your code is less likely to fail if in the future someone changes the values corresponding to the member names.

An enumeration has a name, an underlying data type, and a set of members. Each member represents a constant.

An enumeration declared at class, structure, module, or interface level, outside any procedure, is a *member enumeration*. It is a member of the class, structure, module, or interface that declares it.

Member enumerations can be accessed from anywhere within their class, structure, module, or interface. Code outside a

class, structure, or module must qualify a member enumeration's name with the name of that class, structure, or module. You can avoid the need to use fully qualified names by adding an Imports statement to the source file.

An enumeration declared at namespace level, outside any class, structure, module, or interface, is a member of the namespace in which it appears.

The *declaration context* for an enumeration must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

You can apply attributes to an enumeration as a whole, but not to its members individually. An attribute contributes information to the assembly's metadata.

Data Type

The **Enum** statement can declare the data type of an enumeration. Each member takes the enumeration's data type. You can specify **Byte**, **Integer**, **Long**, **SByte**, **Short**, **UInteger**, **ULong**, or **UShort**.

If you do not specify *datatype* for the enumeration, each member takes the data type of its *initializer*. If you specify both *datatype* and *initializer*, the data type of *initializer* must be convertible to *datatype*. If neither *datatype* nor *initializer* is present, the data type defaults to **Integer**.

Initializing Members

The **Enum** statement can initialize the contents of selected members in *memberlist*. You use *initializer* to supply an expression to be assigned to the member.

If you do not specify *initializer* for a member, Visual Basic initializes it either to zero (if it is the first *member* in *memberlist*), or to a value greater by one than that of the immediately preceding *member*.

The expression supplied in each *initializer* can be any combination of literals, other constants that are already defined, and enumeration members that are already defined, including a previous member of this enumeration. You can use arithmetic and logical operators to combine such elements.

You cannot use variables or functions in *initializer*. However, you can use conversion keywords such as **CByte** and **CShort**. You can also use **AscW** if you call it with a constant **String** or **Char** argument, since that can be evaluated at compile time.

Enumerations cannot have floating-point values. If a member is assigned a floating-point value and **Option Strict** is set to on, a compiler error occurs. If **Option Strict** is off, the value is automatically converted to the **Enum** type.

If the value of a member exceeds the allowable range for the underlying data type, or if you initialize any member to the maximum value allowed by the underlying data type, the compiler reports an error.

Modifiers

Class, structure, module, and interface member enumerations default to public access. You can adjust their access levels with the access modifiers. Namespace member enumerations default to friend access. You can adjust their access levels to public, but not to private or protected. For more information, see Access Levels in Visual Basic.

All enumeration members have public access, and you cannot use any access modifiers on them. However, if the

enumeration itself has a more restricted access level, the specified enumeration access level takes precedence.

By default, all enumerations are types and their fields are constants. Therefore the **Shared**, **Static**, and **ReadOnly** keywords cannot be used when declaring an enumeration or its members.

Assigning Multiple Values

Enumerations typically represent mutually exclusive values. By including the FlagsAttribute attribute in the **Enum** declaration, you can instead assign multiple values to an instance of the enumeration. The FlagsAttribute attribute specifies that the enumeration be treated as a bit field, that is, a set of flags. These are called *bitwise* enumerations.

When you declare an enumeration by using the FlagsAttribute attribute, we recommend that you use powers of 2, that is, 1, 2, 4, 8, 16, and so on, for the values. We also recommend that "None" be the name of a member whose value is 0. For additional guidelines, see FlagsAttribute and Enum.

Example

The following example shows how to use the **Enum** statement. Note that the member is referred to as `EggSizeEnum.Medium`, and not as `Medium`.

VB

```
Public Class Egg
    Enum EggSizeEnum
        Jumbo
        ExtraLarge
        Large
        Medium
        Small
    End Enum

    Public Sub Poach()
        Dim size As EggSizeEnum

        size = EggSizeEnum.Medium
        ' Continue processing...
    End Sub
End Class
```

Example

The method in the following example is outside the `Egg` class. Therefore, `EggSizeEnum` is fully qualified as `Egg.EggSizeEnum`.

VB

```
Public Sub Scramble(ByVal size As Egg.EggSizeEnum)
    ' Process for the three largest sizes.
```

```
    ' Throw an exception for any other size.
    Select Case size
        Case Egg.EggSizeEnum.Jumbo
            ' Process.
        Case Egg.EggSizeEnum.ExtraLarge
            ' Process.
        Case Egg.EggSizeEnum.Large
            ' Process.
        Case Else
            Throw New ApplicationException("size is invalid: " & size.ToString)
    End Select
End Sub
```

Example

The following example uses the **Enum** statement to define a related set of named constant values. In this case, the values are colors you might choose to design data entry forms for a database.

VB

```
Public Enum InterfaceColors
    MistyRose = &HE1E4FF&
    SlateGray = &H908070&
    DodgerBlue = &HFF901E&
    DeepSkyBlue = &HFFBF00&
    SpringGreen = &H7FFF00&
    ForestGreen = &H228B22&
    Goldenrod = &H20A5DA&
    Firebrick = &H2222B2&
End Enum
```

Example

The following example shows values that include both positive and negative numbers.

VB

```
Enum SecurityLevel
    IllegalEntry = -1
    MinimumSecurity = 0
    MaximumSecurity = 1
End Enum
```

Example

In the following example, an **As** clause is used to specify the *datatype* of an enumeration.

VB

```
Public Enum MyEnum As Byte
  Zero
```

```
    One
    Two
End Enum
```

Example

The following example shows how to use a bitwise enumeration. Multiple values can be assigned to an instance of a bitwise enumeration. The **Enum** declaration includes the FlagsAttribute attribute, which indicates that the enumeration can be treated as a set of flags.

VB

```
' Apply the Flags attribute, which allows an instance
' of the enumeration to have multiple values.
<Flags()> Public Enum FilePermissions As Integer
    None = 0
    Create = 1
    Read = 2
    Update = 4
    Delete = 8
End Enum

Public Sub ShowBitwiseEnum()

    ' Declare the non-exclusive enumeration object and
    ' set it to multiple values.
    Dim perm As FilePermissions
    perm = FilePermissions.Read Or FilePermissions.Update

    ' Show the values in the enumeration object.
    Console.WriteLine(perm.ToString)
    ' Output: Read, Update

    ' Show the total integer value of all values
    ' in the enumeration object.
    Console.WriteLine(CInt(perm))
    ' Output: 6

    ' Show whether the enumeration object contains
    ' the specified flag.
    Console.WriteLine(perm.HasFlag(FilePermissions.Update))
    ' Output: True
End Sub
```

Example

The following example iterates through an enumeration. It uses the GetNames method to retrieve an array of member names from the enumeration, and GetValues to retrieve an array of member values.

VB

```
Enum EggSizeEnum
```

```
        Jumbo
        ExtraLarge
        Large
        Medium
        Small
    End Enum

    Public Sub Iterate()
        Dim names = [Enum].GetNames(GetType(EggSizeEnum))
        For Each name In names
            Console.Write(name & " ")
        Next
        Console.WriteLine()
        ' Output: Jumbo ExtraLarge Large Medium Small

        Dim values = [Enum].GetValues(GetType(EggSizeEnum))
        For Each value In values
            Console.Write(value & " ")
        Next
        Console.WriteLine()
        ' Output: 0 1 2 3 4
    End Sub
```

See Also

- Enum
- AscW
- Const Statement (Visual Basic)
- Dim Statement (Visual Basic)
- Implicit and Explicit Conversions (Visual Basic)
- Type Conversion Functions (Visual Basic)
- Constants and Enumerations (Visual Basic)

Class Statement (Visual Basic)

Visual Studio 2015

Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ]
[ Partial ] _
Class name [ ( Of typelist ) ]
    [ Inherits classname ]
    [ Implements interfacenames ]
    [ statements ]
End Class
```

Parts

Term	Definition
attributelist	Optional. See Attribute List.
accessmodifier	Optional. Can be one of the following: • Public • Protected • Friend • Private • **Protected Friend** See Access Levels in Visual Basic.
Shadows	Optional. See Shadows.
MustInherit	Optional. See MustInherit (Visual Basic).
NotInheritable	Optional. See NotInheritable (Visual Basic).
Partial	Optional. Indicates a partial definition of the class. See Partial (Visual Basic).

name	Required. Name of this class. See Declared Element Names (Visual Basic).
Of	Optional. Specifies that this is a generic class.
typelist	Required if you use the Of keyword. List of type parameters for this class. See Type List.
Inherits	Optional. Indicates that this class inherits the members of another class. See Inherits Statement.
classname	Required if you use the **Inherits** statement. The name of the class from which this class derives.
Implements	Optional. Indicates that this class implements the members of one or more interfaces. See Implements Statement.
interfacenames	Required if you use the **Implements** statement. The names of the interfaces this class implements.
statements	Optional. Statements which define the members of this class.
End Class	Required. Terminates the **Class** definition.

Remarks

A **Class** statement defines a new data type. A *class* is a fundamental building block of object-oriented programming (OOP). For more information, see Objects and Classes in Visual Basic.

You can use **Class** only at namespace or module level. This means the *declaration context* for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Each instance of a class has a lifetime independent of all other instances. This lifetime begins when it is created by a New Operator (Visual Basic) clause or by a function such as CreateObject. It ends when all variables pointing to the instance have been set to Nothing (Visual Basic) or to instances of other classes.

Classes default to Friend (Visual Basic) access. You can adjust their access levels with the access modifiers. For more information, see Access Levels in Visual Basic.

Rules

- **Nesting.** You can define one class within another. The outer class is called the *containing class*, and the inner class is called a *nested class*.
- **Inheritance.** If the class uses the Inherits Statement, you can specify only one base class or interface. A class cannot inherit from more than one element.

  A class cannot inherit from another class with a more restrictive access level. For example, a **Public** class cannot inherit from a **Friend** class.

  A class cannot inherit from a class nested within it.

- **Implementation.** If the class uses the Implements Statement, you must implement every member defined by every interface you specify in *interfacenames*. An exception to this is reimplementation of a base class member. For more information, see "Reimplementation" in Implements Clause (Visual Basic).

- **Default Property.** A class can specify at most one property as its *default property*. For more information, see Default (Visual Basic).

Behavior

- **Access Level.** Within a class, you can declare each member with its own access level. Class members default to Public (Visual Basic) access, except variables and constants, which default to Private (Visual Basic) access. When a class has more restricted access than one of its members, the class access level takes precedence.

- **Scope.** A class is in scope throughout its containing namespace, class, structure, or module.

  The scope of every class member is the entire class.

  Lifetime. Visual Basic does not support static classes. The functional equivalent of a static class is provided by a module. For more information, see Module Statement.

  Class members have lifetimes depending on how and where they are declared. For more information, see Lifetime in Visual Basic.

- **Qualification.** Code outside a class must qualify a member's name with the name of that class.

  If code inside a nested class makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested class, then in its containing class, and so on out to the outermost containing element.

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.

- **Shared Members.** You can control whether a member of a class is a shared or instance member.

- **Object Orientation.** Classes are object-oriented, but modules are not. You can create one or more instances of a class. For more information, see Objects and Classes in Visual Basic.

Example

The following example uses a **Class** statement to define a class and several members.

VB

```
Class bankAccount
    Shared interestRate As Decimal
    Private accountNumber As String
    Private accountBalance As Decimal
    Public holdOnAccount As Boolean = False

    Public ReadOnly Property balance() As Decimal
        Get
            Return accountBalance
        End Get
    End Property

    Public Sub postInterest()
        accountBalance = accountBalance * (1 + interestRate)
    End Sub

    Public Sub postDeposit(ByVal amountIn As Decimal)
        accountBalance = accountBalance + amountIn
    End Sub

    Public Sub postWithdrawal(ByVal amountOut As Decimal)
        accountBalance = accountBalance - amountOut
    End Sub
End Class
```

See Also

Objects and Classes in Visual Basic
Structures and Classes (Visual Basic)
Interface Statement (Visual Basic)
Module Statement
Property Statement
Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
How to: Use a Generic Class (Visual Basic)

Structure Statement

Visual Studio 2015

Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Partial ] _
Structure name [ ( Of typelist ) ]
    [ Implements interfacenames ]
    [ datamemberdeclarations ]
    [ methodmemberdeclarations ]
End Structure
```

Parts

Term	Definition
attributelist	Optional. See Attribute List.
accessmodifier	Optional. Can be one of the following: • Public • Protected • Friend • Private • **Protected Friend** See Access Levels in Visual Basic.
Shadows	Optional. See Shadows.
Partial	Optional. Indicates a partial definition of the structure. See Partial (Visual Basic).
name	Required. Name of this structure. See Declared Element Names (Visual Basic).
Of	Optional. Specifies that this is a generic structure.

typelist	Required if you use the Of keyword. List of type parameters for this structure. See Type List.
Implements	Optional. Indicates that this structure implements the members of one or more interfaces. See Implements Statement.
interfacenames	Required if you use the **Implements** statement. The names of the interfaces this structure implements.
datamemberdeclarations	Required. Zero or more **Const**, **Dim**, **Enum**, or **Event** statements declaring *data members* of the structure.
methodmemberdeclarations	Optional. Zero or more declarations of **Function**, **Operator**, **Property**, or **Sub** procedures, which serve as *method members* of the structure.
End Structure	Required. Terminates the **Structure** definition.

Remarks

The **Structure** statement defines a composite value type that you can customize. A *structure* is a generalization of the user-defined type (UDT) of previous versions of Visual Basic. For more information, see Structures (Visual Basic).

Structures support many of the same features as classes. For example, structures can have properties and procedures, they can implement interfaces, and they can have parameterized constructors. However, there are significant differences between structures and classes in areas such as inheritance, declarations, and usage. Also, classes are reference types and structures are value types. For more information, see Structures and Classes (Visual Basic).

You can use **Structure** only at namespace or module level. This means the *declaration context* for a structure must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Structures default to Friend (Visual Basic) access. You can adjust their access levels with the access modifiers. For more information, see Access Levels in Visual Basic.

Rules

- **Nesting.** You can define one structure within another. The outer structure is called the *containing structure*, and the inner structure is called a *nested structure*. However, you cannot access a nested structure's members through the containing structure. Instead, you must declare a variable of the nested structure's data type.

- **Member Declaration.** You must declare every member of a structure. A structure member cannot be Protected or **Protected Friend** because nothing can inherit from a structure. The structure itself, however, can be **Protected** or **Protected Friend**.

  You can declare zero or more nonshared variables or nonshared, noncustom events in a structure. You cannot have only constants, properties, and procedures, even if some of them are nonshared.

- **Initialization.** You cannot initialize the value of any nonshared data member of a structure as part of its

declaration. You must either initialize such a data member by means of a parameterized constructor on the structure, or assign a value to the member after you have created an instance of the structure.

- **Inheritance.** A structure cannot inherit from any type other than ValueType, from which all structures inherit. In particular, one structure cannot inherit from another.

  You cannot use the Inherits Statement in a structure definition, even to specify ValueType.

- **Implementation.** If the structure uses the Implements Statement, you must implement every member defined by every interface you specify in *interfacenames*.

- **Default Property.** A structure can specify at most one property as its *default property*, using the Default (Visual Basic) modifier. For more information, see Default (Visual Basic).

Behavior

- **Access Level.** Within a structure, you can declare each member with its own access level. All structure members default to Public (Visual Basic) access. Note that if the structure itself has a more restricted access level, this automatically restricts access to its members, even if you adjust their access levels with the access modifiers.

- **Scope.** A structure is in scope throughout its containing namespace, class, structure, or module.

  The scope of every structure member is the entire structure.

- **Lifetime.** A structure does not itself have a lifetime. Rather, each instance of that structure has a lifetime independent of all other instances.

  The lifetime of an instance begins when it is created by a New Operator (Visual Basic) clause. It ends when the lifetime of the variable that holds it ends.

  You cannot extend the lifetime of a structure instance. An approximation to static structure functionality is provided by a module. For more information, see Module Statement.

  Structure members have lifetimes depending on how and where they are declared. For more information, see "Lifetime" in Class Statement (Visual Basic).

- **Qualification.** Code outside a structure must qualify a member's name with the name of that structure.

  If code inside a nested structure makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested structure, then in its containing structure, and so on out to the outermost containing element. For more information, see References to Declared Elements (Visual Basic).

- **Memory Consumption.** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration. If you need to control the storage layout of a structure, you can apply the StructLayoutAttribute attribute to the **Structure** statement.

Example

The following example uses the **Structure** statement to define a set of related data for an employee. It shows the use of **Public**, **Friend**, and **Private** members to reflect the sensitivity of the data items. It also shows procedure, property, and event members.

VB

```
Public Structure employee
    ' Public members, accessible from throughout declaration region.
    Public firstName As String
    Public middleName As String
    Public lastName As String
    ' Friend members, accessible from anywhere within the same assembly.
    Friend employeeNumber As Integer
    Friend workPhone As Long
    ' Private members, accessible only from within the structure itself.
    Private homePhone As Long
    Private level As Integer
    Private salary As Double
    Private bonus As Double
    ' Procedure member, which can access structure's private members.
    Friend Sub calculateBonus(ByVal rate As Single)
        bonus = salary * CDbl(rate)
    End Sub
    ' Property member to return employee's eligibility.
    Friend ReadOnly Property eligible() As Boolean
        Get
            Return level >= 25
        End Get
    End Property
    ' Event member, raised when business phone number has changed.
    Public Event changedWorkPhone(ByVal newPhone As Long)
End Structure
```

See Also

Class Statement (Visual Basic)
Interface Statement (Visual Basic)
Module Statement
Dim Statement (Visual Basic)
Const Statement (Visual Basic)
Enum Statement (Visual Basic)
Event Statement
Operator Statement
Property Statement
Structures and Classes (Visual Basic)

Module Statement

Visual Studio 2015

Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ]  Module name
    [ statements ]
End Module
```

Parts

attributelist
Optional. See Attribute List (Visual Basic).

accessmodifier
Optional. Can be one of the following:

- Public
- Friend

See Access Levels in Visual Basic.

name
Required. Name of this module. See Declared Element Names (Visual Basic).

statements
Optional. Statements which define the variables, properties, events, procedures, and nested types of this module.

End Module
Terminates the **Module** definition.

Remarks

A **Module** statement defines a reference type available throughout its namespace. A *module* (sometimes called a *standard module*) is similar to a class but with some important distinctions. Every module has exactly one instance and does not

need to be created or assigned to a variable. Modules do not support inheritance or implement interfaces. Notice that a module is not a *type* in the sense that a class or structure is — you cannot declare a programming element to have the data type of a module.

You can use **Module** only at namespace level. This means the *declaration context* for a module must be a source file or namespace, and cannot be a class, structure, module, interface, procedure, or block. You cannot nest a module within another module, or within any type. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

A module has the same lifetime as your program. Because its members are all **Shared**, they also have lifetimes equal to that of the program.

Modules default to Friend (Visual Basic) access. You can adjust their access levels with the access modifiers. For more information, see Access Levels in Visual Basic.

All members of a module are implicitly **Shared**.

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.
- **Shared Members.** You can control whether a member of a class is a shared or instance member.
- **Object Orientation.** Classes are object-oriented, but modules are not. So only classes can be instantiated as objects. For more information, see Objects and Classes in Visual Basic.

Rules

- **Modifiers.** All module members are implicitly Shared (Visual Basic). You cannot use the **Shared** keyword when declaring a member, and you cannot alter the shared status of any member.
- **Inheritance.** A module cannot inherit from any type other than Object, from which all modules inherit. In particular, one module cannot inherit from another.

  You cannot use the Inherits Statement in a module definition, even to specify Object.
- **Default Property.** You cannot define any default properties in a module. For more information, see Default (Visual Basic).

Behavior

- **Access Level.** Within a module, you can declare each member with its own access level. Module members default to Public (Visual Basic) access, except variables and constants, which default to Private (Visual Basic) access. When a module has more restricted access than one of its members, the specified module access level

takes precedence.

- **Scope.** A module is in scope throughout its namespace.

  The scope of every module member is the entire module. Notice that all members undergo *type promotion*, which causes their scope to be promoted to the namespace containing the module. For more information, see Type Promotion (Visual Basic).

- **Qualification.** You can have multiple modules in a project, and you can declare members with the same name in two or more modules. However, you must qualify any reference to such a member with the appropriate module name if the reference is from outside that module. For more information, see References to Declared Elements (Visual Basic).

Example

VB

```
Public Module thisModule
    Sub Main()
        Dim userName As String = InputBox("What is your name?")
        MsgBox("User name is" & userName)
    End Sub
    ' Insert variable, property, procedure, and event declarations.
End Module
```

See Also

Class Statement (Visual Basic)
Namespace Statement
Structure Statement
Interface Statement (Visual Basic)
Property Statement
Type Promotion (Visual Basic)

Interface Statement (Visual Basic)

Visual Studio 2015

Declares the name of an interface and introduces the definitions of the members that the interface comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] _
Interface name [ ( Of typelist ) ]
    [ Inherits interfacenames ]
    [ [ modifiers ] Property membername ]
    [ [ modifiers ] Function membername ]
    [ [ modifiers ] Sub membername ]
    [ [ modifiers ] Event membername ]
    [ [ modifiers ] Interface membername ]
    [ [ modifiers ] Class membername ]
    [ [ modifiers ] Structure membername ]
End Interface
```

Parts

Term	Definition
attributelist	Optional. See Attribute List.
accessmodifier	Optional. Can be one of the following: • Public • Protected • Friend • Private • **Protected Friend** See Access Levels in Visual Basic.
Shadows	Optional. See Shadows.
name	Required. Name of this interface. See Declared Element Names (Visual Basic).

Of	Optional. Specifies that this is a generic interface.
typelist	Required if you use the Of Clause (Visual Basic) keyword. List of type parameters for this interface. Optionally, each type parameter can be declared variant by using **In** and **Out** generic modifiers. See Type List.
Inherits	Optional. Indicates that this interface inherits the attributes and members of another interface or interfaces. See Inherits Statement.
interfacenames	Required if you use the **Inherits** statement. The names of the interfaces from which this interface derives.
modifiers	Optional. Appropriate modifiers for the interface member being defined.
Property	Optional. Defines a property that is a member of the interface.
Function	Optional. Defines a **Function** procedure that is a member of the interface.
Sub	Optional. Defines a **Sub** procedure that is a member of the interface.
Event	Optional. Defines an event that is a member of the interface.
Interface	Optional. Defines an interface that is a nested within this interface. The nested interface definition must terminate with an **End Interface** statement.
Class	Optional. Defines a class that is a member of the interface. The member class definition must terminate with an **End Class** statement.
Structure	Optional. Defines a structure that is a member of the interface. The member structure definition must terminate with an **End Structure** statement.
membername	Required for each property, procedure, event, interface, class, or structure defined as a member of the interface. The name of the member.
End Interface	Terminates the **Interface** definition.

Remarks

An *interface* defines a set of members, such as properties and procedures, that classes and structures can implement. The interface defines only the signatures of the members and not their internal workings.

A class or structure implements the interface by supplying code for every member defined by the interface. Finally, when the application creates an instance from that class or structure, an object exists and runs in memory. For more information, see Objects and Classes in Visual Basic and Interfaces (Visual Basic).

You can use **Interface** only at namespace or module level. This means the *declaration context* for an interface must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information,

see Declaration Contexts and Default Access Levels (Visual Basic).

Interfaces default to Friend (Visual Basic) access. You can adjust their access levels with the access modifiers. For more information, see Access Levels in Visual Basic.

Rules

- **Nesting Interfaces.** You can define one interface within another. The outer interface is called the *containing interface*, and the inner interface is called a *nested interface*.

- **Member Declaration.** When you declare a property or procedure as a member of an interface, you are defining only the *signature* of that property or procedure. This includes the element type (property or procedure), its parameters and parameter types, and its return type. Because of this, the member definition uses only one line of code, and terminating statements such as **End Function** or **End Property** are not valid in an interface.

  In contrast, when you define an enumeration or structure, or a nested class or interface, it is necessary to include their data members.

- **Member Modifiers.** You cannot use any access modifiers when defining module members, nor can you specify Shared (Visual Basic) or any procedure modifier except Overloads (Visual Basic). You can declare any member with Shadows (Visual Basic), and you can use Default (Visual Basic) when defining a property, as well as ReadOnly (Visual Basic) or WriteOnly (Visual Basic).

- **Inheritance.** If the interface uses the Inherits Statement, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

  An interface cannot inherit from another interface with a more restrictive access level. For example, a **Public** interface cannot inherit from a **Friend** interface.

  An interface cannot inherit from an interface nested within it.

- **Implementation.** When a class uses the Implements Clause (Visual Basic) statement to implement this interface, it must implement every member defined within the interface. Furthermore, each signature in the implementing code must exactly match the corresponding signature defined in this interface. However, the name of the member in the implementing code does not have to match the member name as defined in the interface.

  When a class is implementing a procedure, it cannot designate the procedure as **Shared**.

- **Default Property.** An interface can specify at most one property as its *default property*, which can be referenced without using the property name. You specify such a property by declaring it with the Default (Visual Basic) modifier.

  Notice that this means that an interface can define a default property only if it inherits none.

Behavior

- **Access Level.** All interface members implicitly have Public (Visual Basic) access. You cannot use any access modifier when defining a member. However, a class implementing the interface can declare an access level for each implemented member.

If you assign a class instance to a variable, the access level of its members can depend on whether the data type of the variable is the underlying interface or the implementing class. The following example illustrates this.

VB

```
Public Interface IDemo
    Sub doSomething()
End Interface
Public Class implementIDemo
    Implements IDemo
    Private Sub doSomething() Implements IDemo.doSomething
    End Sub
End Class
Dim varAsInterface As IDemo = New implementIDemo()
Dim varAsClass As implementIDemo = New implementIDemo()
```

If you access class members through `varAsInterface`, they all have public access. However, if you access members through `varAsClass`, the **Sub** procedure `doSomething` has private access.

- **Scope.** An interface is in scope throughout its namespace, class, structure, or module.

  The scope of every interface member is the entire interface.

- **Lifetime.** An interface does not itself have a lifetime, nor do its members. When a class implements an interface and an object is created as an instance of that class, the object has a lifetime within the application in which it is running. For more information, see "Lifetime" in Class Statement (Visual Basic).

Example

The following example uses the **Interface** statement to define an interface named `thisInterface`, which must be implemented with a **Property** statement and a **Function** statement.

VB

```
Public Interface thisInterface
    Property thisProp(ByVal thisStr As String) As Char
    Function thisFunc(ByVal thisInt As Integer) As Integer
End Interface
```

Note that the **Property** and **Function** statements do not introduce blocks ending with **End Property** and **End Function** within the interface. The interface defines only the signatures of its members. The full **Property** and **Function** blocks appear in a class that implements `thisInterface`.

See Also

Interfaces (Visual Basic)
Class Statement (Visual Basic)

Module Statement
Structure Statement
Property Statement
Function Statement (Visual Basic)
Sub Statement (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Variance in Generic Interfaces (C# and Visual Basic)
In (Generic Modifier) (Visual Basic)
Out (Generic Modifier) (Visual Basic)

Function Statement (Visual Basic)

Visual Studio 2015

Declares the name, parameters, and code that define a **Function** procedure.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ proceduremodifiers ] [ Shared ] [ Shadows ] [
Async | Iterator ]
Function name [ (Of typeparamlist) ] [ (parameterlist) ] [ As returntype ]
[ Implements implementslist | Handles eventlist ]
    [ statements ]
    [ Exit Function ]
    [ statements ]
End Function
```

Parts

- *attributelist*

  Optional. See Attribute List.

- *accessmodifier*

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - **Protected Friend**

  See Access Levels in Visual Basic.

- *proceduremodifiers*

  Optional. Can be one of the following:

  - Overloads

  - Overrides
  - Overridable
  - NotOverridable
  - MustOverride
  - **MustOverride Overrides**
  - **NotOverridable Overrides**

- **Shared**

  Optional. See Shared.

- **Shadows**

  Optional. See Shadows.

- **Async**

  Optional. See Async.

- **Iterator**

  Optional. See Iterator.

- *name*

  Required. Name of the procedure. See Declared Element Names (Visual Basic).

- *typeparamlist*

  Optional. List of type parameters for a generic procedure. See Type List.

- *parameterlist*

  Optional. List of local variable names representing the parameters of this procedure. See Parameter List (Visual Basic).

- *returntype*

  Required if **Option Strict** is **On**. Data type of the value returned by this procedure.

- **Implements**

  Optional. Indicates that this procedure implements one or more **Function** procedures, each one defined in an interface implemented by this procedure's containing class or structure. See Implements Statement.

- *implementslist*

  Required if **Implements** is supplied. List of **Function** procedures being implemented.

  ```
  implementedprocedure [ , implementedprocedure ... ]
  ```

Each *implementedprocedure* has the following syntax and parts:

```
interface.definedname
```

Part	Description
interface	Required. Name of an interface implemented by this procedure's containing class or structure.
definedname	Required. Name by which the procedure is defined in *interface*.

- **Handles**

  Optional. Indicates that this procedure can handle one or more specific events. See Handles Clause (Visual Basic).

- *eventlist*

  Required if **Handles** is supplied. List of events this procedure handles.

  ```
  eventspecifier [ , eventspecifier ... ]
  ```

  Each *eventspecifier* has the following syntax and parts:

  ```
  eventvariable.event
  ```

Part	Description
eventvariable	Required. Object variable declared with the data type of the class or structure that raises the event.
event	Required. Name of the event this procedure handles.

- *statements*

  Optional. Block of statements to be executed within this procedure.

- **End Function**

  Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Each procedure, in turn, is declared within a class, a structure, or a module that is referred to as the containing class, structure, or module.

To return a value to the calling code, use a **Function** procedure; otherwise, use a **Sub** procedure.

Defining a Function

You can define a **Function** procedure only at the module level. Therefore, the declaration context for a function must be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Function procedures default to public access. You can adjust their access levels with the access modifiers.

A **Function** procedure can declare the data type of the value that the procedure returns. You can specify any data type or the name of an enumeration, a structure, a class, or an interface. If you don't specify the *returntype* parameter, the procedure returns **Object**.

If this procedure uses the **Implements** keyword, the containing class or structure must also have an **Implements** statement that immediately follows its **Class** or **Structure** statement. The **Implements** statement must include each interface that's specified in *implementslist*. However, the name by which an interface defines the **Function** (in *definedname*) doesn't need to match the name of this procedure (in *name*).

Note

You can use lambda expressions to define function expressions inline. For more information, see Function Expression (Visual Basic) and Lambda Expressions (Visual Basic).

Returning from a Function

When the **Function** procedure returns to the calling code, execution continues with the statement that follows the statement that called the procedure.

To return a value from a function, you can either assign the value to the function name or include it in a **Return** statement.

The **Return** statement simultaneously assigns the return value and exits the function, as the following example shows.

VB

```
Function myFunction(ByVal j As Integer) As Double
    Return 3.87 * j
End Function
```

The following example assigns the return value to the function name `myFunction` and then uses the **Exit Function** statement to return.

VB

```
Function myFunction(ByVal j As Integer) As Double
    myFunction = 3.87 * j
    Exit Function
End Function
```

The **Exit Function** and **Return** statements cause an immediate exit from a **Function** procedure. Any number of **Exit Function** and **Return** statements can appear anywhere in the procedure, and you can mix **Exit Function** and **Return** statements.

If you use **Exit Function** without assigning a value to *name*, the procedure returns the default value for the data type that's specified in *returntype*. If *returntype* isn't specified, the procedure returns **Nothing**, which is the default value for **Object**.

Calling a Function

You call a **Function** procedure by using the procedure name, followed by the argument list in parentheses, in an expression. You can omit the parentheses only if you aren't supplying any arguments. However, your code is more readable if you always include the parentheses.

You call a **Function** procedure the same way that you call any library function such as **Sqrt**, **Cos**, or **ChrW**.

You can also call a function by using the **Call** keyword. In that case, the return value is ignored. Use of the **Call** keyword isn't recommended in most cases. For more information, see Call Statement (Visual Basic).

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, you shouldn't use a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Async Functions

The *Async* feature allows you to invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a function with the Async modifier, you can use the Await operator in the function. When control reaches an **Await** expression in the **Async** function, control returns to the caller, and progress in the function is suspended until the awaited task completes. When the task is complete, execution can resume in the function.

> **Note**
>
> An **Async** procedure returns to the caller when either it encounters the first awaited object that's not yet complete, or it gets to the end of the **Async** procedure, whichever occurs first.

An **Async** function can have a return type of Task(Of TResult) or Task. An example of an **Async** function that has a return type of Task(Of TResult) is provided below.

An **Async** function cannot declare any ByRef parameters.

A Sub Statement (Visual Basic) can also be marked with the **Async** modifier. This is primarily used for event handlers, where a value cannot be returned. An **Async Sub** procedure can't be awaited, and the caller of an **Async Sub** procedure can't catch exceptions that are thrown by the **Sub** procedure.

For more information about **Async** functions, see Asynchronous Programming with Async and Await (C# and Visual Basic), Control Flow in Async Programs (C# and Visual Basic), and Async Return Types (C# and Visual Basic).

Iterator Functions

An *iterator* function performs a custom iteration over a collection, such as a list or array. An iterator function uses the Yield statement to return each element one at a time. When a Yield statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You call an iterator from client code by using a For Each...Next statement.

The return type of an iterator function can be IEnumerable, IEnumerable(Of T), IEnumerator, or IEnumerator(Of T).

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

Example

The following example uses the **Function** statement to declare the name, parameters, and code that form the body of a **Function** procedure. The **ParamArray** modifier enables the function to accept a variable number of arguments.

VB

```
Public Function calcSum(ByVal ParamArray args() As Double) As Double
    calcSum = 0
    If args.Length <= 0 Then Exit Function
    For i As Integer = 0 To UBound(args, 1)
        calcSum += args(i)
```

```
    Next i
End Function
```

Example

The following example invokes the function declared in the preceding example.

VB

```
Module Module1

    Sub Main()
        ' In the following function call, calcSum's local variables
        ' are assigned the following values: args(0) = 4, args(1) = 3,
        ' and so on. The displayed sum is 10.
        Dim returnedValue As Double = calcSum(4, 3, 2, 1)
        Console.WriteLine("Sum: " & returnedValue)
        ' Parameter args accepts zero or more arguments. The sum
        ' displayed by the following statements is 0.
        returnedValue = calcSum()
        Console.WriteLine("Sum: " & returnedValue)
    End Sub

    Public Function calcSum(ByVal ParamArray args() As Double) As Double
        calcSum = 0
        If args.Length <= 0 Then Exit Function
        For i As Integer = 0 To UBound(args, 1)
            calcSum += args(i)
        Next i
    End Function

End Module
```

Example

In the following example, `DelayAsync` is an **Async Function** that has a return type of Task(Of TResult). `DelayAsync` has a **Return** statement that returns an integer. Therefore the function declaration of `DelayAsync` needs to have a return type of **Task(Of Integer)**. Because the return type is **Task(Of Integer)**, the evaluation of the **Await** expression in `DoSomethingAsync` produces an integer. This is demonstrated in this statement: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an **Async Sub** procedure. Because `DoSomethingAsync` is an **Async** function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement demonstrates: `Await DoSomethingAsync()`. The `startButton_Click` **Sub** procedure must be defined with the **Async** modifier because it has an **Await** expression.

VB

```
' Imports System.Diagnostics
' Imports System.Threading.Tasks
```

```
' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

'  Output:
'   Result: 5
```

See Also

Sub Statement (Visual Basic)
Function Procedures (Visual Basic)
Parameter List (Visual Basic)
Dim Statement (Visual Basic)
Call Statement (Visual Basic)
Of Clause (Visual Basic)
Parameter Arrays (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Troubleshooting Procedures (Visual Basic)
Lambda Expressions (Visual Basic)
Function Expression (Visual Basic)

Sub Statement (Visual Basic)

Visual Studio 2015

Declares the name, parameters, and code that define a **Sub** procedure.

Syntax

```
[ <attributelist> ] [ Partial ] [ accessmodifier ] [ proceduremodifiers ]
[ Shared ] [ Shadows ] [ Async ]
Sub name [ (Of typeparamlist) ] [ (parameterlist) ]
[ Implements implementslist | Handles eventlist ]
    [ statements ]
    [ Exit Sub ]
    [ statements ]
End Sub
```

Parts

- *attributelist*

  Optional. See Attribute List.

- **Partial**

  Optional. Indicates definition of a partial method. See Partial Methods (Visual Basic).

- *accessmodifier*

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - **Protected Friend**

  See Access Levels in Visual Basic.

- *proceduremodifiers*

Optional. Can be one of the following:

- Overloads
- Overrides
- Overridable
- NotOverridable
- MustOverride
- **MustOverride Overrides**
- **NotOverridable Overrides**

- **Shared**

  Optional. See Shared.

- **Shadows**

  Optional. See Shadows.

- **Async**

  Optional. See Async.

- *name*

  Required. Name of the procedure. See Declared Element Names (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).

- *typeparamlist*

  Optional. List of type parameters for a generic procedure. See Type List.

- *parameterlist*

  Optional. List of local variable names representing the parameters of this procedure. See Parameter List (Visual Basic).

- **Implements**

  Optional. Indicates that this procedure implements one or more **Sub** procedures, each one defined in an interface implemented by this procedure's containing class or structure. See Implements Statement.

- *implementslist*

  Required if **Implements** is supplied. List of **Sub** procedures being implemented.

  ```
  implementedprocedure [ , implementedprocedure ... ]
  ```

  Each *implementedprocedure* has the following syntax and parts:

```
interface.definedname
```

Part	Description
interface	Required. Name of an interface implemented by this procedure's containing class or structure.
definedname	Required. Name by which the procedure is defined in *interface*.

- **Handles**

  Optional. Indicates that this procedure can handle one or more specific events. See Handles Clause (Visual Basic).

- *eventlist*

  Required if **Handles** is supplied. List of events this procedure handles.

  ```
  eventspecifier [ , eventspecifier ... ]
  ```

  Each *eventspecifier* has the following syntax and parts:

  ```
  eventvariable.event
  ```

Part	Description
eventvariable	Required. Object variable declared with the data type of the class or structure that raises the event.
event	Required. Name of the event this procedure handles.

- *statements*

  Optional. Block of statements to run within this procedure.

- **End Sub**

  Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Use a **Sub** procedure when you don't want to return a value to the calling code. Use a **Function** procedure when you want to return a value.

Defining a Sub Procedure

You can define a **Sub** procedure only at the module level. The declaration context for a sub procedure must, therefore,

be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Sub procedures default to public access. You can adjust their access levels by using the access modifiers.

If the procedure uses the **Implements** keyword, the containing class or structure must have an **Implements** statement that immediately follows its **Class** or **Structure** statement. The **Implements** statement must include each interface that's specified in *implementslist*. However, the name by which an interface defines the **Sub** (in *definedname*) doesn't have to match the name of this procedure (in *name*).

Returning from a Sub Procedure

When a **Sub** procedure returns to the calling code, execution continues with the statement after the statement that called it.

The following example shows a return from a **Sub** procedure.

VB

```
Sub mySub(ByVal q As String)
    Return
End Sub
```

The **Exit Sub** and **Return** statements cause an immediate exit from a **Sub** procedure. Any number of **Exit Sub** and **Return** statements can appear anywhere in the procedure, and you can mix **Exit Sub** and **Return** statements.

Calling a Sub Procedure

You call a **Sub** procedure by using the procedure name in a statement and then following that name with its argument list in parentheses. You can omit the parentheses only if you don't supply any arguments. However, your code is more readable if you always include the parentheses.

A **Sub** procedure and a **Function** procedure can have parameters and perform a series of statements. However, a **Function** procedure returns a value, and a **Sub** procedure doesn't. Therefore, you can't use a **Sub** procedure in an expression.

You can use the **Call** keyword when you call a **Sub** procedure, but that keyword isn't recommended for most uses. For more information, see Call Statement (Visual Basic).

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, if your argument list includes expressions that call other procedures, you shouldn't assume that those expressions will be called in a particular order.

Async Sub Procedures

By using the Async feature, you can invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a procedure with the Async modifier, you can use the Await operator in the procedure. When control

reaches an **Await** expression in the **Async** procedure, control returns to the caller, and progress in the procedure is suspended until the awaited task completes. When the task is complete, execution can resume in the procedure.

> **Note**
>
> An **Async** procedure returns to the caller when either the first awaited object that's not yet complete is encountered or the end of the **Async** procedure is reached, whichever occurs first.

You can also mark a Function Statement (Visual Basic) with the **Async** modifier. An **Async** function can have a return type of Task(Of TResult) or Task. An example later in this topic shows an **Async** function that has a return type of Task(Of TResult).

Async Sub procedures are primarily used for event handlers, where a value can't be returned. An **Async Sub** procedure can't be awaited, and the caller of an **Async Sub** procedure can't catch exceptions that the **Sub** procedure throws.

An **Async** procedure can't declare any ByRef parameters.

For more information about **Async** procedures, see Asynchronous Programming with Async and Await (C# and Visual Basic), Control Flow in Async Programs (C# and Visual Basic), and Async Return Types (C# and Visual Basic).

Example

The following example uses the **Sub** statement to define the name, parameters, and code that form the body of a **Sub** procedure.

VB

```
Sub computeArea(ByVal length As Double, ByVal width As Double)
    ' Declare local variable.
    Dim area As Double
    If length = 0 Or width = 0 Then
        ' If either argument = 0 then exit Sub immediately.
        Exit Sub
    End If
    ' Calculate area of rectangle.
    area = length * width
    ' Print area to Immediate window.
    Debug.WriteLine(area)
End Sub
```

Example

In the following example, `DelayAsync` is an an **Async Function** that has a return type of Task(Of TResult). `DelayAsync` has a **Return** statement that returns an integer. Therefore, the function declaration of `DelayAsync` must have a return type of **Task(Of Integer)**. Because the return type is **Task(Of Integer)**, the evaluation of the **Await** expression in `DoSomethingAsync` produces an integer, as the following statement shows: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an **Async Sub** procedure. Because `DoSomethingAsync` is an **Async** function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `Await DoSomethingAsync()`. The `startButton_Click` **Sub** procedure must be defined with the **Async** modifier because it has an **Await** expression.

VB

```
' Imports System.Diagnostics
' Imports System.Threading.Tasks

' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

'  Output:
'   Result: 5
```

See Also

Implements Statement
Function Statement (Visual Basic)
Parameter List (Visual Basic)
Dim Statement (Visual Basic)
Call Statement (Visual Basic)
Of Clause (Visual Basic)
Parameter Arrays (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Troubleshooting Procedures (Visual Basic)
Partial Methods (Visual Basic)

Declare Statement

Visual Studio 2015

Declares a reference to a procedure implemented in an external file.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Sub ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ]
' -or-
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Function ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ] [ As returntype ]
```

Parts

Term	Definition
attributelist	Optional. See Attribute List.
accessmodifier	Optional. Can be one of the following: • Public • Protected • Friend • Private • **Protected Friend** See Access Levels in Visual Basic.
Shadows	Optional. See Shadows.
charsetmodifier	Optional. Specifies character set and file search information. Can be one of the following: • Ansi (Visual Basic) (default) • Unicode (Visual Basic) • Auto (Visual Basic)

Sub	Optional, but either **Sub** or **Function** must appear. Indicates that the external procedure does not return a value.
Function	Optional, but either **Sub** or **Function** must appear. Indicates that the external procedure returns a value.
name	Required. Name of this external reference. For more information, see Declared Element Names (Visual Basic).
Lib	Required. Introduces a **Lib** clause, which identifies the external file (DLL or code resource) that contains an external procedure.
libname	Required. Name of the file that contains the declared procedure.
Alias	Optional. Indicates that the procedure being declared cannot be identified within its file by the name specified in *name*. You specify its identification in *aliasname*.
aliasname	Required if you use the **Alias** keyword. String that identifies the procedure in one of two ways: The entry point name of the procedure within its file, within quotes (**""**) -or- A number sign (**#**) followed by an integer specifying the ordinal number of the procedure's entry point within its file
parameterlist	Required if the procedure takes parameters. See Parameter List (Visual Basic).
returntype	Required if **Function** is specified and **Option Strict** is **On**. Data type of the value returned by the procedure.

Remarks

Sometimes you need to call a procedure defined in a file (such as a DLL or code resource) outside your project. When you do this, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly, such as where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The **Declare** statement creates a reference to an external procedure and supplies this necessary information.

You can use **Declare** only at module level. This means the *declaration context* for an external reference must be a class, structure, or module, and cannot be a source file, namespace, interface, procedure, or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

External references default to Public (Visual Basic) access. You can adjust their access levels with the access modifiers.

Rules

- **Attributes.** You can apply attributes to an external reference. Any attribute you apply has effect only in your

project, not in the external file.

- **Modifiers.** External procedures are implicitly Shared (Visual Basic). You cannot use the **Shared** keyword when declaring an external reference, and you cannot alter its shared status.

  An external procedure cannot participate in overriding, implement interface members, or handle events. Accordingly, you cannot use the **Overrides**, **Overridable**, **NotOverridable**, **MustOverride**, **Implements**, or **Handles** keyword in a **Declare** statement.

- **External Procedure Name.** You do not have to give this external reference the same name (in *name*) as the procedure's entry-point name within its external file (*aliasname*). You can use an **Alias** clause to specify the entry-point name. This can be useful if the external procedure has the same name as a Visual Basic reserved modifier or a variable, procedure, or any other programming element in the same scope.

Note
Entry-point names in most DLLs are case-sensitive.

- **External Procedure Number.** Alternatively, you can use an **Alias** clause to specify the ordinal number of the entry point within the export table of the external file. To do this, you begin *aliasname* with a number sign (**#**). This can be useful if any character in the external procedure name is not allowed in Visual Basic, or if the external file exports the procedure without a name.

Data Type Rules

- **Parameter Data Types.** If **Option Strict** is **On**, you must specify the data type of each parameter in *parameterlist*. This can be any data type or the name of an enumeration, structure, class, or interface. Within *parameterlist*, you use an **As** clause to specify the data type of the argument to be passed to each parameter.

Note
If the external procedure was not written for the .NET Framework, you must take care that the data types correspond. For example, if you declare an external reference to a Visual Basic 6.0 procedure with an **Integer** parameter (16 bits in Visual Basic 6.0), you must identify the corresponding argument as **Short** in the **Declare** statement, because that is the 16-bit integer type in Visual Basic. Similarly, **Long** has a different data width in Visual Basic 6.0, and **Date** is implemented differently.

- **Return Data Type.** If the external procedure is a **Function** and **Option Strict** is **On**, you must specify the data type of the value returned to the calling code. This can be any data type or the name of an enumeration, structure, class, or interface.

Note

The Visual Basic compiler does not verify that your data types are compatible with those of the external procedure. If there is a mismatch, the common language runtime generates a MarshalDirectiveException exception at run time.

- **Default Data Types.** If **Option Strict** is **Off** and you do not specify the data type of a parameter in *parameterlist*, the Visual Basic compiler converts the corresponding argument to the Object Data Type. Similarly, if you do not specify *returntype*, the compiler takes the return data type to be **Object**.

Note

Because you are dealing with an external procedure that might have been written on a different platform, it is dangerous to make any assumptions about data types or to allow them to default. It is much safer to specify the data type of every parameter and of the return value, if any. This also improves the readability of your code.

Behavior

- **Scope.** An external reference is in scope throughout its class, structure, or module.
- **Lifetime.** An external reference has the same lifetime as the class, structure, or module in which it is declared.
- **Calling an External Procedure.** You call an external procedure the same way you call a **Function** or **Sub** procedure—by using it in an expression if it returns a value, or by specifying it in a Call Statement (Visual Basic) if it does not return a value.

  You pass arguments to the external procedure exactly as specified by *parameterlist* in the **Declare** statement. Do not take into account how the parameters were originally declared in the external file. Similarly, if there is a return value, use it exactly as specified by *returntype* in the **Declare** statement.

- **Character Sets.** You can specify in *charsetmodifier* how Visual Basic should marshal strings when it calls the external procedure. The **Ansi** modifier directs Visual Basic to marshal all strings to ANSI values, and the **Unicode** modifier directs it to marshal all strings to Unicode values. The **Auto** modifier directs Visual Basic to marshal strings according to .NET Framework rules based on the external reference *name*, or *aliasname* if specified. The default value is **Ansi**.

  charsetmodifier also specifies how Visual Basic should look up the external procedure within its external file. **Ansi** and **Unicode** both direct Visual Basic to look it up without modifying its name during the search. **Auto** directs Visual Basic to determine the base character set of the run-time platform and possibly modify the external procedure name, as follows:

  - On an ANSI platform, such as Windows 95, Windows 98, or Windows Millennium Edition, first look up the external procedure with no name modification. If that fails, append "A" to the end of the external procedure name and look it up again.
  - On a Unicode platform, such as Windows NT, Windows 2000, or Windows XP, first look up the external procedure with no name modification. If that fails, append "W" to the end of the external procedure name and look it up again.

- **Mechanism.** Visual Basic uses the .NET Framework *platform invoke* (PInvoke) mechanism to resolve and access external procedures. The **Declare** statement and the DllImportAttribute class both use this mechanism automatically, and you do not need any knowledge of PInvoke. For more information, see Walkthrough: Calling Windows APIs (Visual Basic).

Security Note

If the external procedure runs outside the common language runtime (CLR), it is *unmanaged code*. When you call such a procedure, for example a Win32 API function or a COM method, you might expose your application to security risks. For more information, see Secure Coding Guidelines for Unmanaged Code.

Example

The following example declares an external reference to a **Function** procedure that returns the current user name. It then calls the external procedure `GetUserNameA` as part of the `getUser` procedure.

VB

```
Declare Function getUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Sub getUser()
    Dim buffer As String = New String(CChar(" "), 25)
    Dim retVal As Integer = getUserName(buffer, 25)
    Dim userName As String = Strings.Left(buffer, InStr(buffer, Chr(0)) - 1)
    MsgBox(userName)
End Sub
```

Example

The DllImportAttribute provides an alternative way of using functions in unmanaged code. The following example declares an imported function without using a **Declare** statement.

VB

```
' Add an Imports statement at the top of the class, structure, or
' module that uses the DllImport attribute.
Imports System.Runtime.InteropServices
```

VB

```
<DllImportAttribute("kernel32.dll", EntryPoint:="MoveFileW",
    SetLastError:=True, CharSet:=CharSet.Unicode,
    ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function moveFile(ByVal src As String,
  ByVal dst As String) As Boolean
```

```
    ' This function copies a file from the path src to the path dst.
    ' Leave this function empty. The DLLImport attribute forces calls
    ' to moveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

See Also

LastDllError
Imports Statement (.NET Namespace and Type)
AddressOf Operator (Visual Basic)
Function Statement (Visual Basic)
Sub Statement (Visual Basic)
Parameter List (Visual Basic)
Call Statement (Visual Basic)
Walkthrough: Calling Windows APIs (Visual Basic)

Operator Statement

Visual Studio 2015

Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.

Syntax

```
[ <attrlist> ] Public [ Overloads ] Shared [ Shadows ] [ Widening | Narrowing ]
Operator operatorsymbol ( operand1 [, operand2 ]) [ As [ <attrlist> ] type ]
    [ statements ]
    [ statements ]
    Return returnvalue
    [ statements ]
End Operator
```

Parts

attrlist
: Optional. See Attribute List.

Public
: Required. Indicates that this operator procedure has Public (Visual Basic) access.

Overloads
: Optional. See Overloads (Visual Basic).

Shared
: Required. Indicates that this operator procedure is a Shared (Visual Basic) procedure.

Shadows
: Optional. See Shadows (Visual Basic).

Widening
: Required for a conversion operator unless you specify **Narrowing**. Indicates that this operator procedure defines a Widening (Visual Basic) conversion. See "Widening and Narrowing Conversions" on this Help page.

Narrowing
: Required for a conversion operator unless you specify **Widening**. Indicates that this operator procedure defines a Narrowing (Visual Basic) conversion. See "Widening and Narrowing Conversions" on this Help page.

operatorsymbol
: Required. The symbol or identifier of the operator that this operator procedure defines.

operand1
Required. The name and type of the single operand of a unary operator (including a conversion operator) or the left operand of a binary operator.

operand2
Required for binary operators. The name and type of the right operand of a binary operator.

operand1 and *operand2* have the following syntax and parts:

```
[ ByVal ] operandname [ As operandtype ]
```

Part	Description
ByVal	Optional, but the passing mechanism must be ByVal (Visual Basic).
operandname	Required. Name of the variable representing this operand. See Declared Element Names (Visual Basic).
operandtype	Optional unless **Option Strict** is **On**. Data type of this operand.

type
Optional unless **Option Strict** is **On**. Data type of the value the operator procedure returns.

statements
Optional. Block of statements that the operator procedure runs.

returnvalue
Required. The value that the operator procedure returns to the calling code.

End Operator
Required. Terminates the definition of this operator procedure.

Remarks

You can use **Operator** only in a class or structure. This means the *declaration context* for an operator cannot be a source file, namespace, module, interface, procedure, or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

All operators must be **Public Shared**. You cannot specify **ByRef**, **Optional**, or **ParamArray** for either operand.

You cannot use the operator symbol or identifier to hold a return value. You must use the **Return** statement, and it must specify a value. Any number of **Return** statements can appear anywhere in the procedure.

Defining an operator in this way is called *operator overloading*, whether or not you use the **Overloads** keyword. The following table lists the operators you can define.

Type	Operators

Unary	**+**, **-**, **IsFalse**, **IsTrue**, **Not**
Binary	**+**, **-**, *****, **/**, ****, **&**, **^**, **>>**, **<<**, **=**, **<>**, **>**, **>=**, **<**, **<=**, **And**, **Like**, **Mod**, **Or**, **Xor**
Conversion (unary)	**CType**

Note that the **=** operator in the binary list is the comparison operator, not the assignment operator.

When you define **CType**, you must specify either **Widening** or **Narrowing**.

Matched Pairs

You must define certain operators as matched pairs. If you define either operator of such a pair, you must define the other as well. The matched pairs are the following:

- **=** and **<>**
- **>** and **<**
- **>=** and **<=**
- **IsTrue** and **IsFalse**

Data Type Restrictions

Every operator you define must involve the class or structure on which you define it. This means that the class or structure must appear as the data type of the following:

- The operand of a unary operator.
- At least one of the operands of a binary operator.
- Either the operand or the return type of a conversion operator.

Certain operators have additional data type restrictions, as follows:

- If you define the **IsTrue** and **IsFalse** operators, they must both return the **Boolean** type.
- If you define the **<<** and **>>** operators, they must both specify the **Integer** type for the *operandtype* of *operand2*.

The return type does not have to correspond to the type of either operand. For example, a comparison operator such as **=** or **<>** can return **Boolean** even if neither operand is **Boolean**.

Logical and Bitwise Operators

The **And**, **Or**, **Not**, and **Xor** operators can perform either logical or bitwise operations in Visual Basic. However, if you define one of these operators on a class or structure, you can define only its bitwise operation.

You cannot define the **AndAlso** operator directly with an **Operator** statement. However, you can use **AndAlso** if you have fulfilled the following conditions:

- You have defined **And** on the same operand types you want to use for **AndAlso**.
- Your definition of **And** returns the same type as the class or structure on which you have defined it.
- You have defined the **IsFalse** operator on the class or structure on which you have defined **And**.

Similarly, you can use **OrElse** if you have defined **Or** on the same operands, with the return type of the class or structure, and you have defined **IsTrue** on the class or structure.

Widening and Narrowing Conversions

A *widening conversion* always succeeds at run time, while a *narrowing conversion* can fail at run time. For more information, see Widening and Narrowing Conversions (Visual Basic).

If you declare a conversion procedure to be **Widening**, your procedure code must not generate any failures. This means the following:

- It must always return a valid value of type *type*.
- It must handle all possible exceptions and other error conditions.
- It must handle any error returns from any procedures it calls.

If there is any possibility that a conversion procedure might not succeed, or that it might cause an unhandled exception, you must declare it to be **Narrowing**.

Example

The following code example uses the **Operator** statement to define the outline of a structure that includes operator procedures for the **And**, **Or**, **IsFalse**, and **IsTrue** operators. **And** and **Or** each take two operands of type `abc` and return type `abc`. **IsFalse** and **IsTrue** each take a single operand of type `abc` and return **Boolean**. These definitions allow the calling code to use **And**, **AndAlso**, **Or**, and **OrElse** with operands of type `abc`.

VB

```
Public Structure abc
    Dim d As Date
    Public Shared Operator And(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate And of x and y.
        Return r
```

```
    End Operator
    Public Shared Operator Or(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate Or of x and y.
        Return r
    End Operator
    Public Shared Operator IsFalse(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of z.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of z.
        Return b
    End Operator
End Structure
```

See Also

IsFalse Operator (Visual Basic)
IsTrue Operator (Visual Basic)
Widening (Visual Basic)
Narrowing (Visual Basic)
Widening and Narrowing Conversions (Visual Basic)
Operator Procedures (Visual Basic)
How to: Define an Operator (Visual Basic)
How to: Define a Conversion Operator (Visual Basic)
How to: Call an Operator Procedure (Visual Basic)
How to: Use a Class that Defines Operators (Visual Basic)

Property Statement

Visual Studio 2015

Declares the name of a property, and the property procedures used to store and retrieve the value of the property.

Syntax

```
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ] [ Iterator ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
    [ <attributelist> ] [ accessmodifier ] Get
        [ statements ]
    End Get
    [ <attributelist> ] [ accessmodifier ] Set
( ByVal value As returntype [, parameterlist ] )
        [ statements ]
    End Set
End Property
- or -
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
```

Parts

- *attributelist*

  Optional. List of attributes that apply to this property or **Get** or **Set** procedure. See Attribute List.

- **Default**

  Optional. Specifies that this property is the default property for the class or structure on which it is defined. Default properties must accept parameters and can be set and retrieved without specifying the property name. If you declare the property as **Default**, you cannot use **Private** on the property or on either of its property procedures.

- *accessmodifier*

  Optional on the **Property** statement and on at most one of the **Get** and **Set** statements. Can be one of the following:

  - Public

  - Protected
  - Friend
  - Private
  - **Protected Friend**

  See Access Levels in Visual Basic.

- *propertymodifiers*

  Optional. Can be one of the following:

  - Overloads
  - Overrides
  - Overridable
  - NotOverridable
  - MustOverride
  - **MustOverride Overrides**
  - **NotOverridable Overrides**

- **Shared**

  Optional. See Shared (Visual Basic).

- **Shadows**

  Optional. See Shadows (Visual Basic).

- **ReadOnly**

  Optional. See ReadOnly (Visual Basic).

- **WriteOnly**

  Optional. See WriteOnly (Visual Basic).

- **Iterator**

  Optional. See Iterator.

- *name*

  Required. Name of the property. See Declared Element Names (Visual Basic).

- *parameterlist*

  Optional. List of local variable names representing the parameters of this property, and possible additional parameters of the **Set** procedure. See Parameter List (Visual Basic).

- *returntype*

  Required if **Option Strict** is **On**. Data type of the value returned by this property.

- **Implements**

  Optional. Indicates that this property implements one or more properties, each one defined in an interface implemented by this property's containing class or structure. See Implements Statement.

- *implementslist*

  Required if **Implements** is supplied. List of properties being implemented.

  ```
  implementedproperty [ , implementedproperty ... ]
  ```

  Each *implementedproperty* has the following syntax and parts:

  ```
  interface.definedname
  ```

Part	Description
interface	Required. Name of an interface implemented by this property's containing class or structure.
definedname	Required. Name by which the property is defined in *interface*.

- **Get**

  Optional. Required if the property is marked **WriteOnly**. Starts a **Get** property procedure that is used to return the value of the property.

- *statements*

  Optional. Block of statements to run within the **Get** or **Set** procedure.

- **End Get**

  Terminates the **Get** property procedure.

- **Set**

  Optional. Required if the property is marked **ReadOnly**. Starts a **Set** property procedure that is used to store the value of the property.

- **End Set**

  Terminates the **Set** property procedure.

- **End Property**

  Terminates the definition of this property.

Remarks

The **Property** statement introduces the declaration of a property. A property can have a **Get** procedure (read only), a **Set** procedure (write only), or both (read-write). You can omit the **Get** and **Set** procedure when using an auto-implemented property. For more information, see Auto-Implemented Properties (Visual Basic).

You can use **Property** only at class level. This means the *declaration context* for a property must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

By default, properties use public access. You can adjust a property's access level with an access modifier on the **Property** statement, and you can optionally adjust one of its property procedures to a more restrictive access level.

Visual Basic passes a parameter to the **Set** procedure during property assignments. If you do not supply a parameter for **Set**, the integrated development environment (IDE) uses an implicit parameter named `value`. This parameter holds the value to be assigned to the property. You typically store this value in a private local variable and return it whenever the **Get** procedure is called.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the **Get** or the **Set** procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared **Friend**, you can declare the **Set** procedure **Private**, but not **Public**.

  If you are defining a **ReadOnly** or **WriteOnly** property, the single property procedure (**Get** or **Set**, respectively) represents all of the property. You cannot declare a different access level for such a procedure, because that would set two access levels for the property.

- **Return Type.** The **Property** statement can declare the data type of the value it returns. You can specify any data type or the name of an enumeration, structure, class, or interface.

  If you do not specify *returntype*, the property returns **Object**.

- **Implementation.** If this property uses the **Implements** keyword, the containing class or structure must have an **Implements** statement immediately following its **Class** or **Structure** statement. The **Implements** statement must include each interface specified in *implementslist*. However, the name by which an interface defines the **Property** (in *definedname*) does not have to be the same as the name of this property (in *name*).

Behavior

- **Returning from a Property Procedure.** When the **Get** or **Set** procedure returns to the calling code, execution continues with the statement following the statement that invoked it.

  The **Exit Property** and **Return** statements cause an immediate exit from a property procedure. Any number of **Exit Property** and **Return** statements can appear anywhere in the procedure, and you can mix **Exit Property** and **Return** statements.

- **Return Value.** To return a value from a **Get** procedure, you can either assign the value to the property name or

include it in a **Return** statement. The following example assigns the return value to the property name `quoteForTheDay` and then uses the **Exit Property** statement to return.

VB

```
Private quoteValue As String = "No quote assigned yet."
```

VB

```
ReadOnly Property quoteForTheDay() As String
    Get
        quoteForTheDay = quoteValue
        Exit Property
    End Get
End Property
```

If you use **Exit Property** without assigning a value to *name*, the **Get** procedure returns the default value for the property's data type.

The **Return** statement at the same time assigns the **Get** procedure return value and exits the procedure. The following example shows this.

VB

```
Private quoteValue As String = "No quote assigned yet."
```

VB

```
ReadOnly Property quoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

Example

The following example declares a property in a class.

VB

```
Class Class1
    ' Define a local variable to store the property value.
    Private propertyValue As String
    ' Define the property.
    Public Property prop1() As String
        Get
            ' The Get property procedure is called when the value
            ' of a property is retrieved.
```

```
            Return propertyValue
        End Get
        Set(ByVal value As String)
            ' The Set property procedure is called when the value
            ' of a property is modified.  The value to be assigned
            ' is passed in the argument to Set.
            propertyValue = value
        End Set
    End Property
End Class
```

See Also

Auto-Implemented Properties (Visual Basic)
Objects and Classes in Visual Basic
Get Statement
Set Statement (Visual Basic)
Parameter List (Visual Basic)
Default (Visual Basic)

Event Statement

Visual Studio 2015

Declares a user-defined event.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname[(parameterlist)] _
[ Implements implementslist ]
' -or-
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname As delegatename _
[ Implements implementslist ]
' -or-
 [ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Custom Event eventname As delegatename _
[ Implements implementslist ]
   [ <attrlist> ] AddHandler(ByVal value As delegatename)
      [ statements ]
   End AddHandler
   [ <attrlist> ] RemoveHandler(ByVal value As delegatename)
      [ statements ]
   End RemoveHandler
   [ <attrlist> ] RaiseEvent(delegatesignature)
      [ statements ]
   End RaiseEvent
End Event
```

Parts

Part	Description
attrlist	Optional. List of attributes that apply to this event. Multiple attributes are separated by commas. You must enclose the Attribute List (Visual Basic) in angle brackets ("<" and ">").
accessmodifier	Optional. Specifies what code can access the event. Can be one of the following: • Public—any code that can access the element that declares it can access it.

	• Protected—only code within its class or a derived class can access it. • Friend—only code in the same assembly can access it. • Private—only code in the element that declares it can access it. You can specify **Protected Friend** to enable access from code in the event's class, a derived class, or the same assembly.
Shared	Optional. Specifies that this event is not associated with a specific instance of a class or structure.
Shadows	Optional. Indicates that this event redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a **Private** element shadows a base-class element, code that does not have permission to access the **Private** element accesses the base-class element instead.
eventname	Required. Name of the event; follows standard variable naming conventions.
parameterlist	Optional. List of local variables that represent the parameters of this event. You must enclose the Parameter List (Visual Basic) in parentheses.
Implements	Optional. Indicates that this event implements an event of an interface.
implementslist	Required if **Implements** is supplied. List of **Sub** procedures being implemented. Multiple procedures are separated by commas: *implementedprocedure* [, *implementedprocedure* ...] Each *implementedprocedure* has the following syntax and parts: *interface.definedname* • *interface* - Required. Name of an interface that this procedure's containing class or structure is implementing. • *Definedname* - Required. Name by which the procedure is defined in *interface*. This does not have to be the same as *name*, the name that this procedure is using to implement the defined procedure.
Custom	Required. Events declared as **Custom** must define custom **AddHandler**, **RemoveHandler**, and **RaiseEvent** accessors.
delegatename	Optional. The name of a delegate that specifies the event-handler signature.
AddHandler	Required. Declares an **AddHandler** accessor, which specifies the statements to execute when an event handler is added, either explicitly by using the **AddHandler** statement or implicitly by using the **Handles** clause.
End AddHandler	Required. Terminates the **AddHandler** block.

value	Required. Parameter name.
RemoveHandler	Required. Declares a **RemoveHandler** accessor, which specifies the statements to execute when an event handler is removed using the **RemoveHandler** statement.
End RemoveHandler	Required. Terminates the **RemoveHandler** block.
RaiseEvent	Required. Declares a **RaiseEvent** accessor, which specifies the statements to execute when the event is raised using the **RaiseEvent** statement. Typically, this invokes a list of delegates maintained by the **AddHandler** and **RemoveHandler** accessors.
End RaiseEvent	Required. Terminates the **RaiseEvent** block.
delegatesignature	Required. List of parameters that matches the parameters required by the *delegatename* delegate. You must enclose the Parameter List (Visual Basic) in parentheses.
statements	Optional. Statements that contain the bodies of the **AddHandler**, **RemoveHandler**, and **RaiseEvent** methods.
End Event	Required. Terminates the **Event** block.

Remarks

Once the event has been declared, use the **RaiseEvent** statement to raise the event. A typical event might be declared and raised as shown in the following fragments:

VB

```
Public Class EventSource
    ' Declare an event.
    Public Event LogonCompleted(ByVal UserName As String)
    Sub CauseEvent()
        ' Raise an event on successful logon.
        RaiseEvent LogonCompleted("AustinSteele")
    End Sub
End Class
```

Note

You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, **ParamArray** arguments, or **Optional** arguments. Events do not have return values.

To handle an event, you must associate it with an event handler subroutine using either the **Handles** or **AddHandler** statement. The signatures of the subroutine and the event must match. To handle a shared event, you must use the

AddHandler statement.

You can use **Event** only at module level. This means the *declaration context* for an event must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

In most circumstances, you can use the first syntax in the Syntax section of this topic for declaring events. However, some scenarios require that you have more control over the detailed behavior of the event. The last syntax in the Syntax section of this topic, which uses the **Custom** keyword, provides that control by enabling you to define custom events. In a custom event, you specify exactly what occurs when code adds or removes an event handler to or from the event, or when code raises the event. For examples, see How to: Declare Custom Events To Conserve Memory (Visual Basic) and How to: Declare Custom Events To Avoid Blocking (Visual Basic).

Example

The following example uses events to count down seconds from 10 to 0. The code illustrates several of the event-related methods, properties, and statements. This includes the **RaiseEvent** statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Button1`) and a text box (`TextBox1`). When you click the button, the first text box displays a countdown from 10 to 0 seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Forms project. Then add a button named `Button1` and a text box named `TextBox1` to the main form, named `Form1`. Then right-click the form and click **View Code** to open the code editor.

Add a **WithEvents** variable to the declarations section of the `Form1` class:

VB

```
Private WithEvents mText As TimerState
```

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load` or `Button_Click`.

VB

```
Private Sub Form1_Load() Handles MyBase.Load
    Button1.Text = "Start"
    mText = New TimerState
End Sub
Private Sub Button1_Click() Handles Button1.Click
    mText.StartCountdown(10.0, 0.1)
End Sub

Private Sub mText_ChangeText() Handles mText.Finished
```

```
        TextBox1.Text = "Done"
    End Sub

    Private Sub mText_UpdateTime(ByVal Countdown As Double
      ) Handles mText.UpdateTime

        TextBox1.Text = Format(Countdown, "##0.0")
        ' Use DoEvents to allow the display to refresh.
        My.Application.DoEvents()
    End Sub

    Class TimerState
        Public Event UpdateTime(ByVal Countdown As Double)
        Public Event Finished()
        Public Sub StartCountdown(ByVal Duration As Double,
                                  ByVal Increment As Double)
            Dim Start As Double = DateAndTime.Timer
            Dim ElapsedTime As Double = 0

            Dim SoFar As Double = 0
            Do While ElapsedTime < Duration
                If ElapsedTime > SoFar + Increment Then
                    SoFar += Increment
                    RaiseEvent UpdateTime(Duration - SoFar)
                End If
                ElapsedTime = DateAndTime.Timer - Start
            Loop
            RaiseEvent Finished()
        End Sub
    End Class
```

Press F5 to run the previous example, and click the button labeled **Start**. The first text box starts to count down the seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

Note

The **My.Application.DoEvents** method does not process events in the same way the form does. To enable the form to handle the events directly, you can use multithreading. For more information, see Threading (C# and Visual Basic).

See Also

RaiseEvent Statement
Implements Statement
Events (Visual Basic)
AddHandler Statement
RemoveHandler Statement
Handles Clause (Visual Basic)
Delegate Statement
How to: Declare Custom Events To Conserve Memory (Visual Basic)

How to: Declare Custom Events To Avoid Blocking (Visual Basic)
Shared (Visual Basic)
Shadows (Visual Basic)

Delegate Statement

Visual Studio 2015

Used to declare a delegate. A delegate is a reference type that refers to a **Shared** method of a type or to an instance method of an object. Any procedure with matching parameter and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _
[ Shadows ] Delegate [ Sub | Function ] name [( Of typeparamlist )] [([ parameterlist
])] [ As type ]
```

Parts

Term	Definition
attrlist	Optional. List of attributes that apply to this delegate. Multiple attributes are separated by commas. You must enclose the Attribute List (Visual Basic) in angle brackets ("**<**" and "**>**").
accessmodifier	Optional. Specifies what code can access the delegate. Can be one of the following: • Public. Any code that can access the element that declares the delegate can access it. • Protected. Only code within the delegate's class or a derived class can access it. • Friend. Only code within the same assembly can access the delegate. • Private. Only code within the element that declares the delegate can access it. You can specify **Protected Friend** to enable access from code within the delegate's class, a derived class, or the same assembly.
Shadows	Optional. Indicates that this delegate redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a **Private** element shadows a base class element, code that does not have permission to access the **Private** element accesses the base class element instead.

Sub	Optional, but either **Sub** or **Function** must appear. Declares this procedure as a delegate **Sub** procedure that does not return a value.
Function	Optional, but either **Sub** or **Function** must appear. Declares this procedure as a delegate **Function** procedure that returns a value.
name	Required. Name of the delegate type; follows standard variable naming conventions.
typeparamlist	Optional. List of type parameters for this delegate. Multiple type parameters are separated by commas. Optionally, each type parameter can be declared variant by using **In** and **Out** generic modifiers. You must enclose the Type List (Visual Basic) in parentheses and introduce it with the **Of** keyword.
parameterlist	Optional. List of parameters that are passed to the procedure when it is called. You must enclose the Parameter List (Visual Basic) in parentheses.
type	Required if you specify a **Function** procedure. Data type of the return value.

Remarks

The **Delegate** statement defines the parameter and return types of a delegate class. Any procedure with matching parameters and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance, by calling the delegate's **Invoke** method.

Delegates can be declared at the namespace, module, class, or structure level, but not within a procedure.

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be a reference to a method, or a lambda expression.

To specify a reference to a method, use the following syntax:

AddressOf [*expression*.]*methodname*

The compile-time type of the *expression* must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The *methodname* can be either a shared method or an instance method. The *methodname* is not optional, even if you create a delegate for the default method of the class.

To specify a lambda expression, use the following syntax:

Function ([*parm* As *type*, *parm2* As *type2*, ...]) *expression*

The signature of the function must match that of the delegate type. For more information about lambda expressions, see Lambda Expressions (Visual Basic).

For more information about delegates, see Delegates (Visual Basic).

Example

The following example uses the **Delegate** statement to declare a delegate for operating on two numbers and returning a number. The `DelegateTest` method takes an instance of a delegate of this type and uses it to operate on pairs of numbers.

VB

```
Delegate Function MathOperator(
    ByVal x As Double,
    ByVal y As Double
) As Double

Function AddNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x + y
End Function

Function SubtractNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x - y
End Function

Sub DelegateTest(
    ByVal x As Double,
    ByVal op As MathOperator,
    ByVal y As Double
)
    Dim ret As Double
    ret = op.Invoke(x, y) ' Call the method.
    MsgBox(ret)
End Sub

Protected Sub Test()
    DelegateTest(5, AddressOf AddNumbers, 3)
    DelegateTest(9, AddressOf SubtractNumbers, 3)
End Sub
```

See Also

AddressOf Operator (Visual Basic)
Of Clause (Visual Basic)
Delegates (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Covariance and Contravariance (C# and Visual Basic)
In (Generic Modifier) (Visual Basic)
Out (Generic Modifier) (Visual Basic)