Extension Methods (Visual Basic)

Visual Studio 2015

Extension methods enable developers to add custom functionality to data types that are already defined without creating a new derived type. Extension methods make it possible to write a method that can be called as if it were an instance method of the existing type.

Remarks

An extension method can be only a **Sub** procedure or a **Function** procedure. You cannot define an extension property, field, or event. All extension methods must be marked with the extension attribute `<Extension()>` from the System.Runtime.CompilerServices namespace.

The first parameter in an extension method definition specifies which data type the method extends. When the method is run, the first parameter is bound to the instance of the data type that invokes the method.

Example

Description

The following example defines a `Print` extension to the String data type. The method uses `Console.WriteLine` to display a string. The parameter of the `Print` method, `aString`, establishes that the method extends the String class.

VB

```
Imports System.Runtime.CompilerServices

Module StringExtensions

    <Extension()>
    Public Sub Print(ByVal aString As String)
        Console.WriteLine(aString)
    End Sub

End Module
```

Notice that the extension method definition is marked with the extension attribute `<Extension()>`. Marking the module in which the method is defined is optional, but each extension method must be marked. System.Runtime.CompilerServices must be imported in order to access the extension attribute.

Extension methods can be declared only within modules. Typically, the module in which an extension method is defined is not the same module as the one in which it is called. Instead, the module that contains the extension method is imported, if it needs to be, to bring it into scope. After the module that contains `Print` is in scope, the method can be

called as if it were an ordinary instance method that takes no arguments, such as `ToUpper`:

VB

```
Module Class1

    Sub Main()

        Dim example As String = "Hello"
        ' Call to extension method Print.
        example.Print()

        ' Call to instance method ToUpper.
        example.ToUpper()
        example.ToUpper.Print()

    End Sub

End Module
```

The next example, `PrintAndPunctuate`, is also an extension to String, this time defined with two parameters. The first parameter, `aString`, establishes that the extension method extends String. The second parameter, `punc`, is intended to be a string of punctuation marks that is passed in as an argument when the method is called. The method displays the string followed by the punctuation marks.

VB

```
<Extension()>
Public Sub PrintAndPunctuate(ByVal aString As String,
                             ByVal punc As String)
    Console.WriteLine(aString & punc)
End Sub
```

The method is called by sending in a string argument for `punc`: `example.PrintAndPunctuate(".")`

The following example shows `Print` and `PrintAndPunctuate` defined and called. System.Runtime.CompilerServices is imported in the definition module in order to enable access to the extension attribute.

Code

VB

```
Imports System.Runtime.CompilerServices

Module StringExtensions

    <Extension()>
    Public Sub Print(ByVal aString As String)
        Console.WriteLine(aString)
    End Sub
```

```
    <Extension()>
    Public Sub PrintAndPunctuate(ByVal aString As String,
                                 ByVal punc As String)
        Console.WriteLine(aString & punc)
    End Sub

End Module
```

Next, the extension methods are brought into scope and called.

VB

```
Imports ConsoleApplication2.StringExtensions
Module Module1

    Sub Main()

        Dim example As String = "Example string"
        example.Print()

        example = "Hello"
        example.PrintAndPunctuate(".")
        example.PrintAndPunctuate("!!!!")

    End Sub
End Module
```

Comments

All that is required to be able to run these or similar extension methods is that they be in scope. If the module that contains an extension method is in scope, it is visible in IntelliSense and can be called as if it were an ordinary instance method.

Notice that when the methods are invoked, no argument is sent in for the first parameter. Parameter `aString` in the previous method definitions is bound to `example`, the instance of **String** that calls them. The compiler will use `example` as the argument sent to the first parameter.

If an extension method is called for an object that is set to **Nothing**, the extension method executes. This does not apply to ordinary instance methods. You can explicitly check for **Nothing** in the extension method.

Types That Can Be Extended

You can define an extension method on most types that can be represented in a Visual Basic parameter list, including the following:

- Classes (reference types)
- Structures (value types)
- Interfaces
- Delegates
- ByRef and ByVal arguments
- Generic method parameters
- Arrays

Because the first parameter specifies the data type that the extension method extends, it is required and cannot be optional. For that reason, **Optional** parameters and **ParamArray** parameters cannot be the first parameter in the parameter list.

Extension methods are not considered in late binding. In the following example, the statement `anObject.PrintMe()` raises a MissingMemberException exception, the same exception you would see if the second `PrintMe` extension method definition were deleted.

VB

```
Option Strict Off
Imports System.Runtime.CompilerServices

Module Module4

    Sub Main()
        Dim aString As String = "Initial value for aString"
        aString.PrintMe()

        Dim anObject As Object = "Initial value for anObject"
        ' The following statement causes a run-time error when Option
        ' Strict is off, and a compiler error when Option Strict is on.
        'anObject.PrintMe()
    End Sub

    <Extension()>
    Public Sub PrintMe(ByVal str As String)
        Console.WriteLine(str)
    End Sub

    <Extension()>
    Public Sub PrintMe(ByVal obj As Object)
        Console.WriteLine(obj)
    End Sub

End Module
```

Best Practices

Extension methods provide a convenient and powerful way to extend an existing type. However, to use them successfully, there are some points to consider. These considerations apply mainly to authors of class libraries, but they might affect any application that uses extension methods.

Most generally, extension methods that you add to types that you do not own are more vulnerable than extension methods added to types that you control. A number of things can occur in classes you do not own that can interfere with your extension methods.

- If any accessible instance member exists that has a signature that is compatible with the arguments in the calling statement, with no narrowing conversions required from argument to parameter, the instance method will be used in preference to any extension method. Therefore, if an appropriate instance method is added to a class at some point, an existing extension member that you rely on may become inaccessible.
- The author of an extension method cannot prevent other programmers from writing conflicting extension methods that may have precedence over the original extension.
- You can improve robustness by putting extension methods in their own namespace. Consumers of your library can then include a namespace or exclude it, or select among namespaces, separately from the rest of the library.
- It may be safer to extend interfaces than it is to extend classes, especially if you do not own the interface or class. A change in an interface affects every class that implements it. Therefore, the author may be less likely to add or change methods in an interface. However, if a class implements two interfaces that have extension methods with the same signature, neither extension method is visible.
- Extend the most specific type you can. In a hierarchy of types, if you select a type from which many other types are derived, there are layers of possibilities for the introduction of instance methods or other extension methods that might interfere with yours.

Extension Methods, Instance Methods, and Properties

When an in-scope instance method has a signature that is compatible with the arguments of a calling statement, the instance method is chosen in preference to any extension method. The instance method has precedence even if the extension method is a better match. In the following example, `ExampleClass` contains an instance method named `ExampleMethod` that has one parameter of type **Integer**. Extension method `ExampleMethod` extends `ExampleClass`, and has one parameter of type **Long**.

VB

```
Class ExampleClass
    ' Define an instance method named ExampleMethod.
    Public Sub ExampleMethod(ByVal m As Integer)
        Console.WriteLine("Instance method")
    End Sub
End Class

<Extension()>
Sub ExampleMethod(ByVal ec As ExampleClass,
```

```
                    ByVal n As Long)
    Console.WriteLine("Extension method")
End Sub
```

The first call to `ExampleMethod` in the following code calls the extension method, because `arg1` is **Long** and is compatible only with the **Long** parameter in the extension method. The second call to `ExampleMethod` has an **Integer** argument, `arg2`, and it calls the instance method.

VB

```
Sub Main()
    Dim example As New ExampleClass
    Dim arg1 As Long = 10
    Dim arg2 As Integer = 5

    ' The following statement calls the extension method.
    example.exampleMethod(arg1)
    ' The following statement calls the instance method.
    example.exampleMethod(arg2)
End Sub
```

Now reverse the data types of the parameters in the two methods:

VB

```
Class ExampleClass
    ' Define an instance method named ExampleMethod.
    Public Sub ExampleMethod(ByVal m As Long)
        Console.WriteLine("Instance method")
    End Sub
End Class

<Extension()>
Sub ExampleMethod(ByVal ec As ExampleClass,
                  ByVal n As Integer)
    Console.WriteLine("Extension method")
End Sub
```

This time the code in `Main` calls the instance method both times. This is because both `arg1` and `arg2` have a widening conversion to **Long**, and the instance method takes precedence over the extension method in both cases.

VB

```
Sub Main()
    Dim example As New ExampleClass
    Dim arg1 As Long = 10
    Dim arg2 As Integer = 5

    ' The following statement calls the instance method.
    example.ExampleMethod(arg1)
    ' The following statement calls the instance method.
```

```
        example.ExampleMethod(arg2)
    End Sub
```

Therefore, an extension method cannot replace an existing instance method. However, when an extension method has the same name as an instance method but the signatures do not conflict, both methods can be accessed. For example, if class `ExampleClass` contains a method named `ExampleMethod` that takes no arguments, extension methods with the same name but different signatures are permitted, as shown in the following code.

VB

```
Imports System.Runtime.CompilerServices

Module Module3

    Sub Main()
        Dim ex As New ExampleClass
        ' The following statement calls the extension method.
        ex.ExampleMethod("Extension method")
        ' The following statement calls the instance method.
        ex.ExampleMethod()
    End Sub

    Class ExampleClass
        ' Define an instance method named ExampleMethod.
        Public Sub ExampleMethod()
            Console.WriteLine("Instance method")
        End Sub
    End Class

    <Extension()>
    Sub ExampleMethod(ByVal ec As ExampleClass,
                      ByVal stringParameter As String)
        Console.WriteLine(stringParameter)
    End Sub

End Module
```

The output from this code is as follows:

```
Extension method

Instance method
```

The situation is simpler with properties: if an extension method has the same name as a property of the class it extends, the extension method is not visible and cannot be accessed.

Extension Method Precedence

When two extension methods that have identical signatures are in scope and accessible, the one with higher precedence will be invoked. An extension method's precedence is based on the mechanism used to bring the method into scope. The

following list shows the precedence hierarchy, from highest to lowest.

1. Extension methods defined inside the current module.
2. Extension methods defined inside data types in the current namespace or any one of its parents, with child namespaces having higher precedence than parent namespaces.
3. Extension methods defined inside any type imports in the current file.
4. Extension methods defined inside any namespace imports in the current file.
5. Extension methods defined inside any project-level type imports.
6. Extension methods defined inside any project-level namespace imports.

If precedence does not resolve the ambiguity, you can use the fully qualified name to specify the method that you are calling. If the `Print` method in the earlier example is defined in a module named `StringExtensions`, the fully qualified name is `StringExtensions.Print(example)` instead of `example.Print()`.

See Also

System.Runtime.CompilerServices
ExtensionAttribute
Extension Methods (C# Programming Guide)
Module Statement
Procedure Parameters and Arguments (Visual Basic)
Optional Parameters (Visual Basic)
Parameter Arrays (Visual Basic)
Attributes (C# and Visual Basic)
Scope in Visual Basic

How to: Write an Extension Method (Visual Basic)

Visual Studio 2015

Extension methods enable you to add methods to an existing class. The extension method can be called as if it were an instance of that class.

To define an extension method

1. Open a new or existing Visual Basic application in Visual Studio.
2. At the top of the file in which you want to define an extension method, include the following import statement:

```
Imports System.Runtime.CompilerServices
```

3. Within a module in your new or existing application, begin the method definition with the extension attribute:

```
<Extension()>
```

4. Declare your method in the ordinary way, except that the type of the first parameter must be the data type you want to extend.

```
<Extension()>
Public Sub subName (ByVal para1 As ExtendedType, <other parameters>)
     ' < Body of the method >
End Sub
```

Example

The following example declares an extension method in module `StringExtensions`. A second module, `Module1`, imports `StringExtensions` and calls the method. The extension method must be in scope when it is called. Extension method `PrintAndPunctuate` extends the String class with a method that displays the string instance followed by a string of punctuation symbols sent in as a parameter.

VB

```
' Declarations will typically be in a separate module.
Imports System.Runtime.CompilerServices
```

```
Module StringExtensions
    <Extension()>
    Public Sub PrintAndPunctuate(ByVal aString As String,
                                 ByVal punc As String)
        Console.WriteLine(aString & punc)
    End Sub

End Module
```

VB

```
' Import the module that holds the extension method you want to use,
' and call it.

Imports ConsoleApplication2.StringExtensions

Module Module1

    Sub Main()
        Dim example = "Hello"
        example.PrintAndPunctuate("?")
        example.PrintAndPunctuate("!!!!")
    End Sub

End Module
```

Notice that the method is defined with two parameters and called with only one. The first parameter, `aString`, in the method definition is bound to `example`, the instance of **String** that calls the method. The output of the example is as follows:

```
Hello?

Hello!!!!
```

See Also

- ExtensionAttribute
- Extension Methods (Visual Basic)
- Module Statement
- Procedure Parameters and Arguments (Visual Basic)
- Scope in Visual Basic

How to: Call an Extension Method (Visual Basic)

Visual Studio 2015

Extension methods enable you to add methods to an existing class. After an extension method is declared and brought into scope, you can call it like an instance method of the type that it extends. For more information about how to write an extension method, see How to: Write an Extension Method (Visual Basic).

The following instructions refer to extension method `PrintAndPunctuate`, which will display the string instance that invokes it, followed by whatever value is sent in for the second parameter, `punc`.

VB

```
Imports System.Runtime.CompilerServices

Module StringExtensions
    <Extension()>
    Public Sub PrintAndPunctuate(ByVal aString As String,
                                 ByVal punc As String)
        Console.WriteLine(aString & punc)
    End Sub

End Module
```

The method must be in scope when it is called.

To call an extension method

1. Declare a variable that has the data type of the first parameter of the extension method. For `PrintAndPunctuate`, you need a String variable:

   ```
   Dim example = "Ready"
   ```

2. That variable will invoke the extension method, and its value is bound to the first parameter, `aString`. The following calling statement will display `Ready?`.

   ```
   example.PrintAndPunctuate("?")
   ```

   Notice that the call to this extension method looks just like a call to any one of the String instance methods that

require one parameter:

```
example.EndsWith("dy")
example.IndexOf("R")
```

3. Declare another string variable and call the method again to see that it works with any string.

```
Dim example2 = " or not"
example2.PrintAndPunctuate("!!!")
```

The result this time is: `or not!!!`.

Example

The following code is a complete example of the creation and use of a simple extension method.

VB

```
Imports System.Runtime.CompilerServices
Imports ConsoleApplication1.StringExtensions

Module Module1

    Sub Main()

        Dim example = "Hello"
        example.PrintAndPunctuate(".")
        example.PrintAndPunctuate("!!!!")

        Dim example2 = "Goodbye"
        example2.PrintAndPunctuate("?")
    End Sub

    <Extension()>
    Public Sub PrintAndPunctuate(ByVal aString As String,
                                 ByVal punc As String)
        Console.WriteLine(aString & punc)
    End Sub
End Module

' Output:
' Hello.
' Hello!!!!
' Goodbye?
```

See Also

How to: Write an Extension Method (Visual Basic)
Extension Methods (Visual Basic)
Scope in Visual Basic

Lambda Expressions (Visual Basic)

Visual Studio 2015

A *lambda expression* is a function or subroutine without a name that can be used wherever a delegate is valid. Lambda expressions can be functions or subroutines and can be single-line or multi-line. You can pass values from the current scope to a lambda expression.

Note

The **RemoveHandler** statement is an exception. You cannot pass a lambda expression in for the delegate parameter of **RemoveHandler**.

You create lambda expressions by using the **Function** or **Sub** keyword, just as you create a standard function or subroutine. However, lambda expressions are included in a statement.

The following example is a lambda expression that increments its argument and returns the value. The example shows both the single-line and multi-line lambda expression syntax for a function.

VB

```
Dim increment1 = Function(x) x + 1
Dim increment2 = Function(x)
                     Return x + 2
                 End Function

' Write the value 2.
Console.WriteLine(increment1(1))

' Write the value 4.
Console.WriteLine(increment2(2))
```

The following example is a lambda expression that writes a value to the console. The example shows both the single-line and multi-line lambda expression syntax for a subroutine.

VB

```
Dim writeline1 = Sub(x) Console.WriteLine(x)
Dim writeline2 = Sub(x)
                     Console.WriteLine(x)
                 End Sub

' Write "Hello".
writeline1("Hello")

' Write "World"
```

```
writeline2("World")
```

Notice that in the previous examples the lambda expressions are assigned to a variable name. Whenever you refer to the variable, you invoke the lambda expression. You can also declare and invoke a lambda expression at the same time, as shown in the following example.

VB

```
Console.WriteLine((Function(num As Integer) num + 1)(5))
```

A lambda expression can be returned as the value of a function call (as is shown in the example in the Context section later in this topic), or passed in as an argument to a parameter that takes a delegate type, as shown in the following example.

VB

```
Module Module2

    Sub Main()
        ' The following line will print Success, because 4 is even.
        testResult(4, Function(num) num Mod 2 = 0)
        ' The following line will print Failure, because 5 is not > 10.
        testResult(5, Function(num) num > 10)
    End Sub

    ' Sub testResult takes two arguments, an integer value and a
    ' delegate function that takes an integer as input and returns
    ' a boolean.
    ' If the function returns True for the integer argument, Success
    ' is displayed.
    ' If the function returns False for the integer argument, Failure
    ' is displayed.
    Sub testResult(ByVal value As Integer, ByVal fun As Func(Of Integer, Boolean))
        If fun(value) Then
            Console.WriteLine("Success")
        Else
            Console.WriteLine("Failure")
        End If
    End Sub

End Module
```

Lambda Expression Syntax

The syntax of a lambda expression resembles that of a standard function or subroutine. The differences are as follows:

- A lambda expression does not have a name.
- Lambda expressions cannot have modifiers, such as **Overloads** or **Overrides**.
- Single-line lambda functions do not use an **As** clause to designate the return type. Instead, the type is inferred

from the value that the body of the lambda expression evaluates to. For example, if the body of the lambda expression is `cust.City = "London"`, its return type is **Boolean**.

- In multi-line lambda functions, you can either specify a return type by using an **As** clause, or omit the **As** clause so that the return type is inferred. When the **As** clause is omitted for a multi-line lambda function, the return type is inferred to be the dominant type from all the **Return** statements in the multi-line lambda function. The *dominant type* is a unique type that all other types can widen to. If this unique type cannot be determined, the dominant type is the unique type that all other types in the array can narrow to. If neither of these unique types can be determined, the dominant type is **Object**. In this case, if **Option Strict** is set to **On**, a compiler error occurs.

  For example, if the expressions supplied to the **Return** statement contain values of type **Integer**, **Long**, and **Double**, the resulting array is of type **Double**. Both **Integer** and **Long** widen to **Double** and only **Double**. Therefore, **Double** is the dominant type. For more information, see Widening and Narrowing Conversions (Visual Basic).

- The body of a single-line function must be an expression that returns a value, not a statement. There is no **Return** statement for single-line functions. The value returned by the single-line function is the value of the expression in the body of the function.

- The body of a single-line subroutine must be single-line statement.

- Single-line functions and subroutines do not include an **End Function** or **End Sub** statement.

- You can specify the data type of a lambda expression parameter by using the **As** keyword, or the data type of the parameter can be inferred. Either all parameters must have specified data types or all must be inferred.

- **Optional** and **Paramarray** parameters are not permitted.

- Generic parameters are not permitted.

Async Lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the Async (Visual Basic) and Await Operator (Visual Basic) keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

VB

```
Public Class Form1

    Async Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ' ExampleMethodAsync returns a Task.
        Await ExampleMethodAsync()
        TextBox1.Text = vbCrLf & "Control returned to button1_Click."
    End Sub

    Async Function ExampleMethodAsync() As Task
        ' The following line simulates a task-returning asynchronous process.
        Await Task.Delay(1000)
    End Function
```

```
End Class
```

You can add the same event handler by using an async lambda in an AddHandler Statement. To add this handler, add an **Async** modifier before the lambda parameter list, as the following example shows.

VB

```
Public Class Form1

    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        AddHandler Button1.Click,
            Async Sub(sender1, e1)
                ' ExampleMethodAsync returns a Task.
                Await ExampleMethodAsync()
                TextBox1.Text = vbCrLf & "Control returned to Button1_ Click."
            End Sub
    End Sub

    Async Function ExampleMethodAsync() As Task
        ' The following line simulates a task-returning asynchronous process.
        Await Task.Delay(1000)
    End Function

End Class
```

For more information about how to create and use async methods, see Asynchronous Programming with Async and Await (C# and Visual Basic).

Context

A lambda expression shares its context with the scope within which it is defined. It has the same access rights as any code written in the containing scope. This includes access to member variables, functions and subs, **Me**, and parameters and local variables in the containing scope.

Access to local variables and parameters in the containing scope can extend beyond the lifetime of that scope. As long as a delegate referring to a lambda expression is not available to garbage collection, access to the variables in the original environment is retained. In the following example, variable `target` is local to `makeTheGame`, the method in which the lambda expression `playTheGame` is defined. Note that the returned lambda expression, assigned to `takeAGuess` in `Main`, still has access to the local variable `target`.

VB

```
Module Module6

    Sub Main()
        ' Variable takeAGuess is a Boolean function. It stores the target
        ' number that is set in makeTheGame.
        Dim takeAGuess As gameDelegate = makeTheGame()
```

```
        ' Set up the loop to play the game.
        Dim guess As Integer
        Dim gameOver = False
        While Not gameOver
            guess = CInt(InputBox("Enter a number between 1 and 10 (0 to quit)",
"Guessing Game", "0"))
            ' A guess of 0 means you want to give up.
            If guess = 0 Then
                gameOver = True
            Else
                ' Tests your guess and announces whether you are correct. Method
takeAGuess
                ' is called multiple times with different guesses. The target value is
not
                ' accessible from Main and is not passed in.
                gameOver = takeAGuess(guess)
                Console.WriteLine("Guess of " & guess & " is " & gameOver)
            End If
        End While

    End Sub

    Delegate Function gameDelegate(ByVal aGuess As Integer) As Boolean

    Public Function makeTheGame() As gameDelegate

        ' Generate the target number, between 1 and 10. Notice that
        ' target is a local variable. After you return from makeTheGame,
        ' it is not directly accessible.
        Randomize()
        Dim target As Integer = CInt(Int(10 * Rnd() + 1))

        ' Print the answer if you want to be sure the game is not cheating
        ' by changing the target at each guess.
        Console.WriteLine("(Peeking at the answer) The target is " & target)

        ' The game is returned as a lambda expression. The lambda expression
        ' carries with it the environment in which it was created. This
        ' environment includes the target number. Note that only the current
        ' guess is a parameter to the returned lambda expression, not the target.

        ' Does the guess equal the target?
        Dim playTheGame = Function(guess As Integer) guess = target

        Return playTheGame

    End Function

End Module
```

The following example demonstrates the wide range of access rights of the nested lambda expression. When the returned lambda expression is executed from `Main` as `aDel`, it accesses these elements:

- A field of the class in which it is defined: `aField`
- A property of the class in which it is defined: `aProp`
- A parameter of method `functionWithNestedLambda`, in which it is defined: `level1`
- A local variable of `functionWithNestedLambda`: `localVar`
- A parameter of the lambda expression in which it is nested: `level2`

VB

```
Module Module3

    Sub Main()
        ' Create an instance of the class, with 1 as the value of
        ' the property.
        Dim lambdaScopeDemoInstance =
            New LambdaScopeDemoClass With {.Prop = 1}

        ' Variable aDel will be bound to the nested lambda expression
        ' returned by the call to functionWithNestedLambda.
        ' The value 2 is sent in for parameter level1.
        Dim aDel As aDelegate =
            lambdaScopeDemoInstance.functionWithNestedLambda(2)

        ' Now the returned lambda expression is called, with 4 as the
        ' value of parameter level3.
        Console.WriteLine("First value returned by aDel:   " & aDel(4))

        ' Change a few values to verify that the lambda expression has
        ' access to the variables, not just their original values.
        lambdaScopeDemoInstance.aField = 20
        lambdaScopeDemoInstance.Prop = 30
        Console.WriteLine("Second value returned by aDel: " & aDel(40))
    End Sub

    Delegate Function aDelegate(
        ByVal delParameter As Integer) As Integer

    Public Class LambdaScopeDemoClass
        Public aField As Integer = 6
        Dim aProp As Integer

        Property Prop() As Integer
            Get
                Return aProp
            End Get
            Set(ByVal value As Integer)
                aProp = value
            End Set
        End Property
```

```
        Public Function functionWithNestedLambda(
            ByVal level1 As Integer) As aDelegate

            Dim localVar As Integer = 5

            ' When the nested lambda expression is executed the first
            ' time, as aDel from Main, the variables have these values:
            ' level1 = 2
            ' level2 = 3, after aLambda is called in the Return statement
            ' level3 = 4, after aDel is called in Main
            ' locarVar = 5
            ' aField = 6
            ' aProp = 1
            ' The second time it is executed, two values have changed:
            ' aField = 20
            ' aProp = 30
            ' level3 = 40
            Dim aLambda = Function(level2 As Integer) _
                              Function(level3 As Integer) _
                                  level1 + level2 + level3 + localVar +
                                    aField + aProp

            ' The function returns the nested lambda, with 3 as the
            ' value of parameter level2.
            Return aLambda(3)
        End Function

    End Class
End Module
```

Converting to a Delegate Type

A lambda expression can be implicitly converted to a compatible delegate type. For information about the general requirements for compatibility, see Relaxed Delegate Conversion (Visual Basic). For example, the following code example shows a lambda expression that implicitly converts to `Func(Of Integer, Boolean)` or a matching delegate signature.

VB

```
' Explicitly specify a delegate type.
Delegate Function MultipleOfTen(ByVal num As Integer) As Boolean

' This function matches the delegate type.
Function IsMultipleOfTen(ByVal num As Integer) As Boolean
    Return num Mod 10 = 0
End Function

' This method takes an input parameter of the delegate type.
' The checkDelegate parameter could also be of
' type Func(Of Integer, Boolean).
Sub CheckForMultipleOfTen(ByVal values As Integer(),
                          ByRef checkDelegate As MultipleOfTen)
```

```
    For Each value In values
        If checkDelegate(value) Then
            Console.WriteLine(value & " is a multiple of ten.")
        Else
            Console.WriteLine(value & " is not a multiple of ten.")
        End If
    Next
End Sub

' This method shows both an explicitly defined delegate and a
' lambda expression passed to the same input parameter.
Sub CheckValues()
    Dim values = {5, 10, 11, 20, 40, 30, 100, 3}
    CheckForMultipleOfTen(values, AddressOf IsMultipleOfTen)
    CheckForMultipleOfTen(values, Function(num) num Mod 10 = 0)
End Sub
```

The following code example shows a lambda expression that implicitly converts to `Sub(Of Double, String, Double)` or a matching delegate signature.

VB

```
Module Module1
    Delegate Sub StoreCalculation(ByVal value As Double,
                                  ByVal calcType As String,
                                  ByVal result As Double)

    Sub Main()
        ' Create a DataTable to store the data.
        Dim valuesTable = New DataTable("Calculations")
        valuesTable.Columns.Add("Value", GetType(Double))
        valuesTable.Columns.Add("Calculation", GetType(String))
        valuesTable.Columns.Add("Result", GetType(Double))

        ' Define a lambda subroutine to write to the DataTable.
        Dim writeToValuesTable = Sub(value As Double, calcType As String, result As
Double)
                                     Dim row = valuesTable.NewRow()
                                     row(0) = value
                                     row(1) = calcType
                                     row(2) = result
                                     valuesTable.Rows.Add(row)
                                 End Sub

        ' Define the source values.
        Dim s = {1, 2, 3, 4, 5, 6, 7, 8, 9}

        ' Perform the calculations.
        Array.ForEach(s, Sub(c) CalculateSquare(c, writeToValuesTable))
        Array.ForEach(s, Sub(c) CalculateSquareRoot(c, writeToValuesTable))

        ' Display the data.
        Console.WriteLine("Value" & vbTab & "Calculation" & vbTab & "Result")
```

```
        For Each row As DataRow In valuesTable.Rows
            Console.WriteLine(row(0).ToString() & vbTab &
                              row(1).ToString() & vbTab &
                              row(2).ToString())
        Next

    End Sub


    Sub CalculateSquare(ByVal number As Double, ByVal writeTo As StoreCalculation)
        writeTo(number, "Square     ", number ^ 2)
    End Sub

    Sub CalculateSquareRoot(ByVal number As Double, ByVal writeTo As StoreCalculation)
        writeTo(number, "Square Root", Math.Sqrt(number))
    End Sub
End Module
```

When you assign lambda expressions to delegates or pass them as arguments to procedures, you can specify the parameter names but omit their data types, letting the types be taken from the delegate.

Examples

- The following example defines a lambda expression that returns **True** if the nullable argument has an assigned value, and **False** if its value is **Nothing**.

VB

```
Dim notNothing =
  Function(num? As Integer) num IsNot Nothing
Dim arg As Integer = 14
Console.WriteLine("Does the argument have an assigned value?")
Console.WriteLine(notNothing(arg))
```

- The following example defines a lambda expression that returns the index of the last element in an array.

VB

```
Dim numbers() = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Dim lastIndex =
  Function(intArray() As Integer) intArray.Length - 1
For i = 0 To lastIndex(numbers)
    numbers(i) += 1
Next
```

See Also

Procedures in Visual Basic
Introduction to LINQ in Visual Basic
Delegates (Visual Basic)
Function Statement (Visual Basic)
Sub Statement (Visual Basic)
Nullable Value Types (Visual Basic)
How to: Pass Procedures to Another Procedure in Visual Basic
How to: Create a Lambda Expression (Visual Basic)
Relaxed Delegate Conversion (Visual Basic)

How to: Create a Lambda Expression (Visual Basic)

Visual Studio 2015

A *lambda expression* is a function or subroutine that does not have a name. A lambda expression can be used wherever a delegate type is valid.

To create a single-line lambda expression function

1. In any situation where a delegate type could be used, type the keyword **Function**, as in the following example:

   ```
   Dim add1 = Function
   ```

2. In parentheses, directly after `Function`, type the parameters of the function. Notice that you do not specify a name after `Function`.

   ```
   Dim add1 = Function (num As Integer)
   ```

3. Following the parameter list, type a single expression as the body of the function. The value that the expression evaluates to is the value returned by the function. You do not use an **As** clause to specify the return type.

   VB

   ```
   Dim add1 = Function(num As Integer) num + 1
   ```

   You call the lambda expression by passing in an integer argument.

   VB

   ```
   ' The following line prints 6.
   Console.WriteLine(add1(5))
   ```

4. Alternatively, the same result is accomplished by the following example:

   VB

   ```
   Console.WriteLine((Function(num As Integer) num + 1)(5))
   ```

To create a single-line lambda expression subroutine

1. In any situation where a delegate type could be used, type the keyword **Sub**, as shown in the following example.

   ```
   Dim add1 = Sub
   ```

2. In parentheses, directly after `Sub`, type the parameters of the subroutine. Notice that you do not specify a name after `Sub`.

```
Dim add1 = Sub (msg As String)
```

3. Following the parameter list, type a single statement as the body of the subroutine.

VB

```
Dim writeMessage = Sub(msg As String) Console.WriteLine(msg)
```

You call the lambda expression by passing in a string argument.

VB

```
' The following line prints "Hello".
writeMessage("Hello")
```

To create a multiline lambda expression function

1. In any situation where a delegate type could be used, type the keyword **Function**, as shown in the following example.

```
Dim add1 = Function
```

2. In parentheses, directly after `Function`, type the parameters of the function. Notice that you do not specify a name after `Function`.

```
Dim add1 = Function (index As Integer)
```

3. Press ENTER. The **End Function** statement is automatically added.

4. Within the body of the function, add the following code to create an expression and return the value. You do not use an **As** clause to specify the return type.

VB

```
Dim getSortColumn = Function(index As Integer)
                        Select Case index
                            Case 0
                                Return "FirstName"
                            Case 1
                                Return "LastName"
                            Case 2
                                Return "CompanyName"
                            Case Else
                                Return "LastName"
                        End Select
                    End Function
```

You call the lambda expression by passing in an integer argument.

VB

```
Dim sortColumn = getSortColumn(0)
```

To create a multiline lambda expression subroutine

1. In any situation where a delegate type could be used, type the keyword **Sub**, as shown in the following example:

   ```
   Dim add1 = Sub
   ```

2. In parentheses, directly after `Sub`, type the parameters of the subroutine. Notice that you do not specify a name after `Sub`.

   ```
   Dim add1 = Sub(msg As String)
   ```

3. Press ENTER. The **End Sub** statement is automatically added.

4. Within the body of the function, add the following code to execute when the subroutine is invoked.

   VB

   ```
   Dim writeToLog = Sub(msg As String)
                        Dim log As New EventLog()
                        log.Source = "Application"
                        log.WriteEntry(msg)
                        log.Close()
                    End Sub
   ```

   You call the lambda expression by passing in a string argument.

   VB

   ```
   writeToLog("Application started.")
   ```

Example

A common use of lambda expressions is to define a function that can be passed in as the argument for a parameter whose type is **Delegate**. In the following example, the GetProcesses method returns an array of the processes running on the local computer. The Where(Of TSource) method from the Enumerable class requires a **Boolean** delegate as its argument. The lambda expression in the example is used for that purpose. It returns **True** for each process that has only one thread, and those are selected in `filteredList`.

VB

```
Sub Main()

    ' Create an array of running processes.
    Dim procList As Process() = Diagnostics.Process.GetProcesses

    ' Return the processes that have one thread. Notice that the type
```

```
        ' of the parameter does not have to be explicitly stated.
        Dim filteredList = procList.Where(Function(p) p.Threads.Count = 1)

        ' Display the name of each selected process.
        For Each proc In filteredList
            MsgBox(proc.ProcessName)
        Next

    End Sub
```

The previous example is equivalent to the following code, which is written in Language-Integrated Query (LINQ) syntax:

VB

```
    Sub Main()

        Dim filteredQuery = From proc In Diagnostics.Process.GetProcesses
                            Where proc.Threads.Count = 1
                            Select proc

        For Each proc In filteredQuery
            MsgBox(proc.ProcessName)
        Next
    End Sub
```

See Also

- Enumerable
- Lambda Expressions (Visual Basic)
- Function Statement (Visual Basic)
- Sub Statement (Visual Basic)
- Delegates (Visual Basic)
- How to: Pass Procedures to Another Procedure in Visual Basic
- Delegate Statement
- Introduction to LINQ in Visual Basic

Property Procedures (Visual Basic)

Visual Studio 2015

A property procedure is a series of Visual Basic statements that manipulate a custom property on a module, class, or structure. Property procedures are also known as *property accessors*.

Visual Basic provides for the following property procedures:

- A **Get** procedure returns the value of a property. It is called when you access the property in an expression.
- A **Set** procedure sets a property to a value, including an object reference. It is called when you assign a value to the property.

You usually define property procedures in pairs, using the **Get** and **Set** statements, but you can define either procedure alone if the property is read-only (Get Statement) or write-only (Set Statement (Visual Basic)).

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 define properties in classes, structures, and modules. Properties are **Public** by default, which means you can call them from anywhere in your application that can access the property's container.

For a comparison of properties and variables, see Differences Between Properties and Variables in Visual Basic.

Declaration Syntax

A property itself is defined by a block of code enclosed within the Property Statement and the **End Property** statement. Inside this block, each property procedure appears as an internal block enclosed within a declaration statement (**Get** or **Set**) and the matching **End** declaration.

The syntax for declaring a property and its procedures is as follows:

```
[Default] [Modifiers] Property PropertyName[(ParameterList)] [As DataType]
    [AccessLevel] Get
        ' Statements of the Get procedure.
        ' The following statement returns an expression as the property's value.
        Return Expression
    End Get
    [AccessLevel] Set[(ByVal NewValue As DataType)]
        ' Statements of the Set procedure.
        ' The following statement assigns newvalue as the property's value.
        LValue = NewValue
    End Set
End Property
- or -
```

```
[Default] [Modifiers] Property PropertyName [(ParameterList)] [As DataType]
```

The *Modifiers* can specify access level and information regarding overloading, overriding, sharing, and shadowing, as well as whether the property is read-only or write-only. The *AccessLevel* on the **Get** or **Set** procedure can be any level that is more restrictive than the access level specified for the property itself. For more information, see Property Statement.

Data Type

A property's data type and principal access level are defined in the **Property** statement, not in the property procedures. A property can have only one data type. For example, you cannot define a property to store a **Decimal** value but retrieve a **Double** value.

Access Level

However, you can define a principal access level for a property and further restrict the access level in one of its property procedures. For example, you can define a **Public** property and then define a **Private Set** procedure. The **Get** procedure remains **Public**. You can change the access level in only one of a property's procedures, and you can only make it more restrictive than the principal access level. For more information, see How to: Declare a Property with Mixed Access Levels (Visual Basic).

Parameter Declaration

You declare each parameter the same way you do for Sub Procedures (Visual Basic), except that the passing mechanism must be **ByVal**.

The syntax for each parameter in the parameter list is as follows:

```
[Optional] ByVal [ParamArray] parametername As datatype
```

If the parameter is optional, you must also supply a default value as part of its declaration. The syntax for specifying a default value is as follows:

```
Optional ByVal parametername As datatype = defaultvalue
```

Property Value

In a **Get** procedure, the return value is supplied to the calling expression as the value of the property.

In a **Set** procedure, the new property value is passed to the parameter of the **Set** statement. If you explicitly declare a parameter, you must declare it with the same data type as the property. If you do not declare a parameter, the compiler uses the implicit parameter `Value` to represent the new value to be assigned to the property.

Calling Syntax

You invoke a property procedure implicitly by making reference to the property. You use the name of the property the same way you would use the name of a variable, except that you must provide values for all arguments that are not optional, and you must enclose the argument list in parentheses. If no arguments are supplied, you can optionally omit the parentheses.

The syntax for an implicit call to a **Set** procedure is as follows:

```
propertyname[(argumentlist)] = expression
```

The syntax for an implicit call to a **Get** procedure is as follows:

```
lvalue = propertyname[(argumentlist)]

Do While (propertyname[(argumentlist)] > expression)
```

Illustration of Declaration and Call

The following property stores a full name as two constituent names, the first name and the last name. When the calling code reads `fullName`, the **Get** procedure combines the two constituent names and returns the full name. When the calling code assigns a new full name, the **Set** procedure attempts to break it into two constituent names. If it does not find a space, it stores it all as the first name.

VB

```
Dim firstName, lastName As String
Property fullName() As String
    Get
      If lastName = "" Then
          Return firstName
      Else
          Return firstName & " " & lastName
      End If

    End Get
    Set(ByVal Value As String)
        Dim space As Integer = Value.IndexOf(" ")
        If space < 0 Then
            firstName = Value
            lastName = ""
        Else
            firstName = Value.Substring(0, space)
            lastName = Value.Substring(space + 1)
        End If
    End Set
End Property
```

The following example shows typical calls to the property procedures of `fullName`.

VB

```
fullName = "MyFirstName MyLastName"
MsgBox(fullName)
```

See Also

- Procedures in Visual Basic
- Function Procedures (Visual Basic)
- Operator Procedures (Visual Basic)
- Procedure Parameters and Arguments (Visual Basic)
- Differences Between Properties and Variables in Visual Basic
- How to: Create a Property (Visual Basic)
- How to: Call a Property Procedure (Visual Basic)
- How to: Declare and Call a Default Property in Visual Basic
- How to: Put a Value in a Property (Visual Basic)
- How to: Get a Value from a Property (Visual Basic)

Differences Between Properties and Variables in Visual Basic

Visual Studio 2015

Variables and properties both represent values that you can access. However, there are differences in storage and implementation.

Variables

A *variable* corresponds directly to a memory location. You define a variable with a single declaration statement. A variable can be a *local variable*, defined inside a procedure and available only within that procedure, or it can be a *member variable*, defined in a module, class, or structure but not inside any procedure. A member variable is also called a *field*.

Properties

A *property* is a data element defined on a module, class, or structure. You define a property with a code block between the **Property** and **End Property** statements. The code block contains a **Get** procedure, a **Set** procedure, or both. These procedures are called *property procedures* or *property accessors*. In addition to retrieving or storing the property's value, they can also perform custom actions, such as updating an access counter.

Differences

The following table shows some important differences between variables and properties.

Point of difference	**Variable**	**Property**
Declaration	Single declaration statement	Series of statements in a code block
Implementation	Single storage location	Executable code (property procedures)
Storage	Directly associated with variable's value	Typically has internal storage not available outside the property's containing class or module Property's value might or might not exist as a stored element [1]
Executable code	None	Must have at least one procedure

Read and write access	Read/write or read-only	Read/write, read-only, or write-only
Custom actions (in addition to accepting or returning value)	Not possible	Can be performed as part of setting or retrieving property value

[1] Unlike a variable, the value of a property might not correspond directly to a single item of storage. The storage might be split into pieces for convenience or security, or the value might be stored in an encrypted form. In these cases the **Get** procedure would assemble the pieces or decrypt the stored value, and the **Set** procedure would encrypt the new value or split it into the constituent storage. A property value might be ephemeral, like time of day, in which case the **Get** procedure would calculate it on the fly each time you access the property.

See Also

Property Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Property Statement
Dim Statement (Visual Basic)
How to: Create a Property (Visual Basic)
How to: Declare a Property with Mixed Access Levels (Visual Basic)
How to: Call a Property Procedure (Visual Basic)
How to: Declare and Call a Default Property in Visual Basic
How to: Put a Value in a Property (Visual Basic)
How to: Get a Value from a Property (Visual Basic)

Auto-Implemented Properties (Visual Basic)

Visual Studio 2015

Auto-implemented properties enable you to quickly specify a property of a class without having to write code to **Get** and **Set** the property. When you write code for an auto-implemented property, the Visual Basic compiler automatically creates a private field to store the property variable in addition to creating the associated **Get** and **Set** procedures.

With auto-implemented properties, a property, including a default value, can be declared in a single line. The following example shows three property declarations.

VB

```
Public Property Name As String
Public Property Owner As String = "DefaultName"
Public Property Items As New List(Of String) From {"M", "T", "W"}
Public Property ID As New Guid()
```

An auto-implemented property is equivalent to a property for which the property value is stored in a private field. The following code example shows an auto-implemented property.

VB

```
Property Prop2 As String = "Empty"
```

The following code example shows the equivalent code for the previous auto-implemented property example.

VB

```
Private _Prop2 As String = "Empty"
Property Prop2 As String
    Get
        Return _Prop2
    End Get
    Set(ByVal value As String)
        _Prop2 = value
    End Set
End Property
```

The following code show implementing readonly properties:

VB

```
Class Customer
   Public ReadOnly Property Tags As New List(Of String)
   Public ReadOnly Property Name As String = ""
   Public ReadOnly Property File As String
```

```
    Sub New(file As String)
       Me.File = file
    End Sub
End Class
```

You can assign to the property with initialization expressions as shown in the example, or you can assign to the properties in the containing type's constructor. You can assign to the backing fields of readonly properties at any time.

Backing Field

When you declare an auto-implemented property, Visual Basic automatically creates a hidden private field called the *backing field* to contain the property value. The backing field name is the auto-implemented property name preceded by an underscore (_). For example, if you declare an auto-implemented property named `ID`, the backing field is named `_ID`. If you include a member of your class that is also named `_ID`, you produce a naming conflict and Visual Basic reports a compiler error.

The backing field also has the following characteristics:

- The access modifier for the backing field is always **Private**, even when the property itself has a different access level, such as **Public**.
- If the property is marked as **Shared**, the backing field also is shared.
- Attributes specified for the property do not apply to the backing field.
- The backing field can be accessed from code within the class and from debugging tools such as the Watch window. However, the backing field does not show in an IntelliSense word completion list.

Initializing an Auto-Implemented Property

Any expression that can be used to initialize a field is valid for initializing an auto-implemented property. When you initialize an auto-implemented property, the expression is evaluated and passed to the **Set** procedure for the property. The following code examples show some auto-implemented properties that include initial values.

VB

```
Property FirstName As String = "James"
Property PartNo As Integer = 44302
Property Orders As New List(Of Order)(500)
```

You cannot initialize an auto-implemented property that is a member of an **Interface**, or one that is marked **MustOverride**.

When you declare an auto-implemented property as a member of a **Structure**, you can only initialize the auto-implemented property if it is marked as **Shared**.

When you declare an auto-implemented property as an array, you cannot specify explicit array bounds. However, you can

supply a value by using an array initializer, as shown in the following examples.

VB

```
Property Grades As Integer() = {90, 73}
Property Temperatures As Integer() = New Integer() {68, 54, 71}
```

Property Definitions That Require Standard Syntax

Auto-implemented properties are convenient and support many programming scenarios. However, there are situations in which you cannot use an auto-implemented property and must instead use standard, or *expanded*, property syntax.

You have to use expanded property-definition syntax if you want to do any one of the following:

- Add code to the **Get** or **Set** procedure of a property, such as code to validate incoming values in the **Set** procedure. For example, you might want to verify that a string that represents a telephone number contains the required number of numerals before setting the property value.
- Specify different accessibility for the **Get** and **Set** procedure. For example, you might want to make the **Set** procedure **Private** and the **Get** procedure **Public**.
- Create properties that are **WriteOnly**.
- Use parameterized properties (including **Default** properties). You must declare an expanded property in order to specify a parameter for the property, or to specify additional parameters for the **Set** procedure.
- Place an attribute on the backing field, or change the access level of the backing field.
- Provide XML comments for the backing field.

Expanding an Auto-Implemented Property

If you have to convert an auto-implemented property to an expanded property that contains a **Get** or **Set** procedure, the Visual Basic Code Editor can automatically generate the **Get** and **Set** procedures and **End Property** statement for the property. The code is generated if you put the cursor on a blank line following the **Property** statement, type a **G** (for **Get**) or an **S** (for **Set**) and press ENTER. The Visual Basic Code Editor automatically generates the **Get** or **Set** procedure for read-only and write-only properties when you press ENTER at the end of a **Property** statement.

See Also

How to: Declare and Call a Default Property in Visual Basic
How to: Declare a Property with Mixed Access Levels (Visual Basic)
Property Statement
ReadOnly (Visual Basic)
WriteOnly (Visual Basic)

Objects and Classes in Visual Basic

Procedure Parameters and Arguments (Visual Basic)

Visual Studio 2015

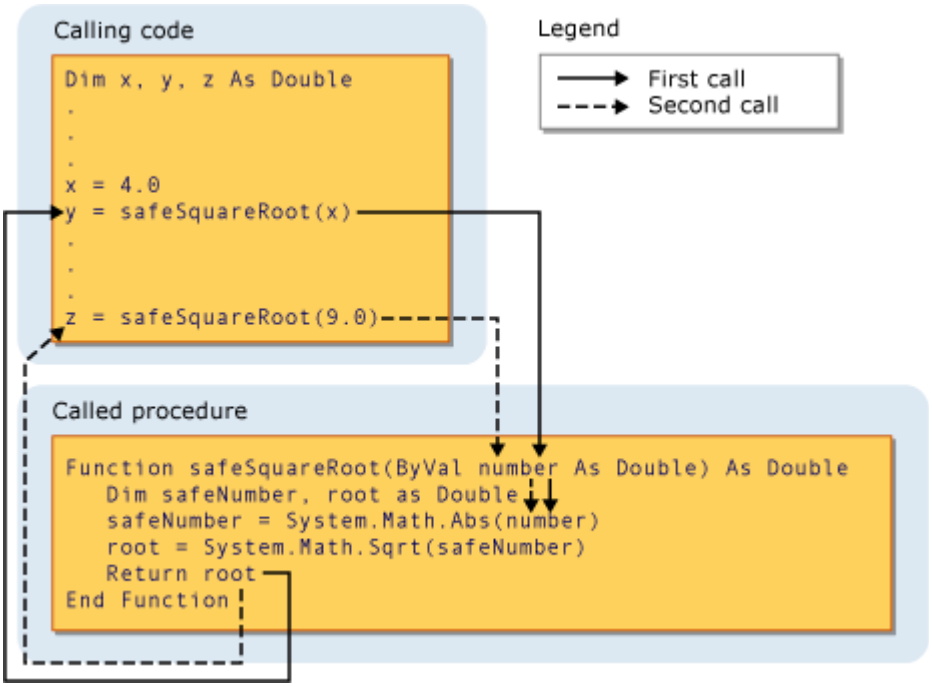
In most cases, a procedure needs some information about the circumstances in which it has been called. A procedure that performs repeated or shared tasks uses different information for each call. This information consists of variables, constants, and expressions that you pass to the procedure when you call it.

A *parameter* represents a value that the procedure expects you to supply when you call it. The procedure's declaration defines its parameters.

You can define a procedure with no parameters, one parameter, or more than one. The part of the procedure definition that specifies the parameters is called the *parameter list*.

An *argument* represents the value you supply to a procedure parameter when you call the procedure. The calling code supplies the arguments when it calls the procedure. The part of the procedure call that specifies the arguments is called the *argument list*.

The following illustration shows code calling the procedure `safeSquareRoot` from two different places. The first call passes the value of the variable `x` (4.0) to the parameter `number`, and the return value in `root` (2.0) is assigned to the variable `y`. The second call passes the literal value 9.0 to `number`, and assigns the return value (3.0) to variable `z`.



Passing an argument to a parameter

For more information, see Differences Between Parameters and Arguments (Visual Basic).

Parameter Data Type

You define a data type for a parameter by using the **As** clause in its declaration. For example, the following function accepts a string and an integer.

VB

```
Function appointment(ByVal day As String, ByVal hour As Integer) As String
    ' Insert code to return any appointment for the given day and time.
    Return "appointment"
End Function
```

If the type checking switch (Option Strict Statement) is **Off,** the **As** clause is optional, except that if any one parameter uses it, all parameters must use it. If type checking is **On**, the **As** clause is required for all procedure parameters.

If the calling code expects to supply an argument with a data type different from that of its corresponding parameter, such as **Byte** to a **String** parameter, it must do one of the following:

- Supply only arguments with data types that widen to the parameter data type;
- Set **Option Strict Off** to allow implicit narrowing conversions; or
- Use a conversion keyword to explicitly convert the data type.

Type Parameters

A *generic procedure* also defines one or more *type parameters* in addition to its normal parameters. A generic procedure allows the calling code to pass different data types each time it calls the procedure, so it can tailor the data types to the requirements of each individual call. See Generic Procedures in Visual Basic.

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
How to: Define a Parameter for a Procedure (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Procedure Overloading (Visual Basic)
Type Conversions in Visual Basic

How to: Declare a Property with Mixed Access Levels (Visual Basic)

Visual Studio 2015

If you want the **Get** and **Set** procedures on a property to have different access levels, you can use the more permissive level in the **Property** statement and the more restrictive level in either the **Get** or **Set** statement. You use mixed access levels on a property when you want certain parts of the code to be able to get the property's value, and certain other parts of the code to be able to change the value.

For more information on access levels, see Access Levels in Visual Basic.

To declare a property with mixed access levels

1. Declare the property in the normal way, and specify the less restrictive access level (such as **Public**) in the **Property** statement.

2. Declare either the **Get** or the **Set** procedure specifying the more restrictive access level (such as **Friend**).

3. Do not specify an access level on the other property procedure. It assumes the access level declared in the **Property** statement. You can restrict access on only one of the property procedures.

VB

```
Public Class employee
    Private salaryValue As Double
    Protected Property salary() As Double
        Get
            Return salaryValue
        End Get
        Private Set(ByVal value As Double)
            salaryValue = value
        End Set
    End Property
End Class
```

In the preceding example, the **Get** procedure has the same **Protected** access as the property itself, while the **Set** procedure has **Private** access. A class derived from `employee` can read the `salary` value, but only the `employee` class can set it.

See Also

Procedures in Visual Basic
Property Procedures (Visual Basic)

- Procedure Parameters and Arguments (Visual Basic)
- Property Statement
- Differences Between Properties and Variables in Visual Basic
- How to: Create a Property (Visual Basic)
- How to: Call a Property Procedure (Visual Basic)
- How to: Declare and Call a Default Property in Visual Basic
- How to: Put a Value in a Property (Visual Basic)
- How to: Get a Value from a Property (Visual Basic)

How to: Call a Property Procedure (Visual Basic)

Visual Studio 2015

You call a property procedure by storing a value in the property or retrieving its value. You access a property the same way you access a variable.

The property's **Set** procedure stores a value, and its **Get** procedure retrieves the value. However, you do not explicitly call these procedures by name. You use the property in an assignment statement or an expression, just as you would store or retrieve the value of a variable. Visual Basic makes the calls to the property's procedures.

To call a property's Get procedure

1. Use the property name in an expression the same way you would use a variable name. You can use a property anywhere you can use a variable or a constant.

   -or-

   Use the property name following the equal (**=**) sign in an assignment statement.

   The following example reads the value of the Now property, implicitly calling its **Get** procedure.

   VB

   ```
   Dim ThisMoment As Date
   ' The following statement calls the Get procedure of the Visual Basic Now property.
   ThisMoment = Now
   ```

2. If the property takes arguments, follow the property name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the property defines the corresponding parameters.

The value of the property participates in the expression just as a variable or constant would, or it is stored in the variable or property on the left side of the assignment statement.

To call a property's Set procedure

1. Use the property name on the left side of an assignment statement.

   The following example sets the value of the TimeOfDay property, implicitly calling the **Set** procedure.

   VB

```
' The following statement calls the Set procedure of the Visual Basic TimeOfDay
property.
TimeOfDay = #12:00:00 PM#
```

2. If the property takes arguments, follow the property name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the property defines the corresponding parameters.

The value generated on the right side of the assignment statement is stored in the property.

See Also

Property Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Property Statement
Differences Between Properties and Variables in Visual Basic
How to: Create a Property (Visual Basic)
How to: Declare a Property with Mixed Access Levels (Visual Basic)
How to: Declare and Call a Default Property in Visual Basic
How to: Put a Value in a Property (Visual Basic)
How to: Get a Value from a Property (Visual Basic)
Get Statement
Set Statement (Visual Basic)

How to: Declare and Call a Default Property in Visual Basic

Visual Studio 2015

A *default property* is a class or structure property that your code can access without specifying it. When calling code names a class or structure but not a property, and the context allows access to a property, Visual Basic resolves the access to that class or structure's default property if one exists.

A class or structure can have at most one default property. However, you can overload a default property and have more than one version of it.

For more information, see Default (Visual Basic).

To declare a default property

1. Declare the property in the normal way. Do not specify the **Shared** or **Private** keyword.
2. Include the **Default** keyword in the property declaration.
3. Specify at least one parameter for the property. You cannot define a default property that does not take at least one argument.

VB

```
Default Property myProperty(ByVal index As Integer) As String
```

To call a default property

1. Declare a variable of the containing class or structure type.

VB

```
Dim x As New class1(3)
```

2. Use the variable name alone in an expression where you would normally include the property name.

VB

```
MsgBox(x)
```

3. Follow the variable name with an argument list in parentheses. A default property must take at least one argument.

VB

```
MsgBox(x(1))
```

4. To retrieve the default property value, use the variable name, with an argument list, in an expression or following the equal (=) sign in an assignment statement.

VB

```
MsgBox(x(1) & x(2) & x(3))
```

5. To set the default property value, use the variable name, with an argument list, on the left side of an assignment statement.

VB

```
x(1) = "Hello"
x(2) = " "
x(3) = "World"
```

6. You can always specify the default property name together with the variable name, just as you would do to access any other property.

VB

```
x.myProperty(1) = "Hello"
x.myProperty(2) = " "
x.myProperty(3) = "World"
```

Example

The following example declares a default property on a class.

VB

```
Public Class class1
    Private myStrings() As String
    Sub New(ByVal size As Integer)
        ReDim myStrings(size)
    End Sub
    Default Property myProperty(ByVal index As Integer) As String
        Get
            ' The Get property procedure is called when the value
            ' of the property is retrieved.
            Return myStrings(index)
        End Get
        Set(ByVal Value As String)
            ' The Set property procedure is called when the value
            ' of the property is modified.
            ' The value to be assigned is passed in the argument
            ' to Set.
            myStrings(index) = Value
```

```
            End Set
        End Property
    End Class
```

Example

The following example demonstrates how to call the default property `myProperty` on class `class1`. The three assignment statements store values in `myProperty`, and the MsgBox call reads the values.

VB

```
Sub Test()
    Dim x As New class1(3)
    x(1) = "Hello"
    x(2) = " "
    x(3) = "World"
    MsgBox(x(1) & x(2) & x(3))
End Sub
```

The most common use of a default property is the Item property on various collection classes.

Robust Programming

Default properties can result in a small reduction in source code-characters, but they can make your code more difficult to read. If the calling code is not familiar with your class or structure, when it makes a reference to the class or structure name it cannot be certain whether that reference accesses the class or structure itself, or a default property. This can lead to compiler errors or subtle run-time logic errors.

You can somewhat reduce the chance of default property errors by always using the Option Strict Statement to set compiler type checking to **On**.

If you are planning to use a predefined class or structure in your code, you must determine whether it has a default property, and if so, what its name is.

Because of these disadvantages, you should consider not defining default properties. For code readability, you should also consider always referring to all properties explicitly, even default properties.

See Also

- Property Procedures (Visual Basic)
- Procedure Parameters and Arguments (Visual Basic)
- Property Statement
- Default (Visual Basic)
- Differences Between Properties and Variables in Visual Basic
- How to: Create a Property (Visual Basic)
- How to: Declare a Property with Mixed Access Levels (Visual Basic)
- How to: Call a Property Procedure (Visual Basic)
- How to: Put a Value in a Property (Visual Basic)
- How to: Get a Value from a Property (Visual Basic)

How to: Put a Value in a Property (Visual Basic)

Visual Studio 2015

You store a value in a property by putting the property name on the left side of an assignment statement.

The property's **Set** procedure stores a value, but you do not explicitly call it by name. You use the property just as you would use a variable. Visual Basic makes the calls to the property's procedures.

To store a value in a property

1. Use the property name on the left side of an assignment statement.

   The following example sets the value of the Visual Basic **TimeOfDay** property to noon, implicitly calling its **Set** procedure.

   VB

   ```
   ' The following statement calls the Set procedure of the Visual Basic TimeOfDay
   property.
   TimeOfDay = #12:00:00 PM#
   ```

2. If the property takes arguments, follow the property name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the property defines the corresponding parameters.

4. The value generated on the right side of the assignment statement is stored in the property.

See Also

TimeOfDay
Property Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Property Statement
Differences Between Properties and Variables in Visual Basic
How to: Create a Property (Visual Basic)
How to: Declare a Property with Mixed Access Levels (Visual Basic)
How to: Call a Property Procedure (Visual Basic)
How to: Declare and Call a Default Property in Visual Basic
How to: Get a Value from a Property (Visual Basic)

How to: Get a Value from a Property (Visual Basic)

Visual Studio 2015

You retrieve a property's value by including the property name in an expression.

The property's **Get** procedure retrieves the value, but you do not explicitly call it by name. You use the property just as you would use a variable. Visual Basic makes the calls to the property's procedures.

To retrieve a value from a property

1. Use the property name in an expression the same way you would use a variable name. You can use a property anywhere you can use a variable or a constant.

   -or-

   Use the property name following the equal (=) sign in an assignment statement.

   The following example reads the value of the Visual Basic **Now** property, implicitly calling its **Get** procedure.

   VB

   ```
   Dim ThisMoment As Date
   ' The following statement calls the Get procedure of the Visual Basic Now property.
   ThisMoment = Now
   ```

2. If the property takes arguments, follow the property name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the property defines the corresponding parameters.

The value of the property participates in the expression just as a variable or constant would, or it is stored in the variable or property on the left side of the assignment statement.

See Also

Procedures in Visual Basic
Property Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Property Statement
Differences Between Properties and Variables in Visual Basic
How to: Create a Property (Visual Basic)
How to: Declare a Property with Mixed Access Levels (Visual Basic)

- How to: Call a Property Procedure (Visual Basic)
- How to: Declare and Call a Default Property in Visual Basic
- How to: Put a Value in a Property (Visual Basic)