Delegates (Visual Basic)

Visual Studio 2015

Delegates are objects that refer to methods. They are sometimes described as *type-safe function pointers* because they are similar to function pointers used in other programming languages. But unlike function pointers, Visual Basic delegates are a reference type based on the class System.Delegate. Delegates can reference both shared methods — methods that can be called without a specific instance of a class — and instance methods.

Delegates and Events

Delegates are useful in situations where you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately, the object raising the events cannot know ahead of time which event handler is handling a specific event. Visual Basic lets you dynamically associate event handlers with events by creating a delegate for you when you use the **AddHandler** statement. At run time, the delegate forwards calls to the appropriate event handler.

Although you can create your own delegates, in most cases Visual Basic creates the delegate and takes care of the details for you. For example, an **Event** statement implicitly defines a delegate class named `<EventName>EventHandler` as a nested class of the class containing the **Event** statement, and with the same signature as the event. The **AddressOf** statement implicitly creates an instance of a delegate that refers to a specific procedure. The following two lines of code are equivalent. In the first line, you see the explicit creation of an instance of `Eventhandler`, with a reference to method `Button1_Click` sent as the argument. The second line is a more convenient way to do the same thing.

VB

```
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
' The following line of code is shorthand for the previous line.
AddHandler Button1.Click, AddressOf Me.Button1_Click
```

You can use the shorthand way of creating delegates anywhere the compiler can determine the delegate's type by the context.

Declaring Events that Use an Existing Delegate Type

In some situations, you may want to declare an event to use an existing delegate type as its underlying delegate. The following syntax demonstrates how:

VB

```
Delegate Sub DelegateType()
Event AnEvent As DelegateType
```

This is useful when you want to route multiple events to the same handler.

Delegate Variables and Parameters

You can use delegates for other, non-event related tasks, such as free threading or with procedures that need to call different versions of functions at run time.

For example, suppose you have a classified-ad application that includes a list box with the names of cars. The ads are sorted by title, which is normally the make of the car. A problem you may face occurs when some cars include the year of the car before the make. The problem is that the built-in sort functionality of the list box sorts only by character codes; it places all the ads starting with dates first, followed by the ads starting with the make.

To fix this, you can create a sort procedure in a class that uses the standard alphabetic sort on most list boxes, but is able to switch at run time to the custom sort procedure for car ads. To do this, you pass the custom sort procedure to the sort class at run time, using delegates.

AddressOf and Lambda Expressions

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 following example shows both **AddressOf** and lambda expressions used to specify the reference for a delegate.

VB

```
Module Module1

    Sub Main()
```

```
        ' Create an instance of InOrderClass and assign values to the properties.
        ' InOrderClass method ShowInOrder displays the numbers in ascending
        ' or descending order, depending on the comparison method you specify.
        Dim inOrder As New InOrderClass
        inOrder.Num1 = 5
        inOrder.Num2 = 4

        ' Use AddressOf to send a reference to the comparison function you want
        ' to use.
        inOrder.ShowInOrder(AddressOf GreaterThan)
        inOrder.ShowInOrder(AddressOf LessThan)

        ' Use lambda expressions to do the same thing.
        inOrder.ShowInOrder(Function(m, n) m > n)
        inOrder.ShowInOrder(Function(m, n) m < n)
    End Sub

    Function GreaterThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
        Return num1 > num2
    End Function

    Function LessThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
        Return num1 < num2
    End Function

    Class InOrderClass
        ' Define the delegate function for the comparisons.
        Delegate Function CompareNumbers(ByVal num1 As Integer, ByVal num2 As Integer)
As Boolean
        ' Display properties in ascending or descending order.
        Sub ShowInOrder(ByVal compare As CompareNumbers)
            If compare(_num1, _num2) Then
                Console.WriteLine(_num1 & "  " & _num2)
            Else
                Console.WriteLine(_num2 & "  " & _num1)
            End If
        End Sub

        Private _num1 As Integer
        Property Num1() As Integer
            Get
                Return _num1
            End Get
            Set(ByVal value As Integer)
                _num1 = value
            End Set
        End Property

        Private _num2 As Integer
        Property Num2() As Integer
            Get
                Return _num2
            End Get
            Set(ByVal value As Integer)
```

```
                _num2 = value
            End Set
        End Property
    End Class
End Module
```

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 examples of lambda expression and **AddressOf** assignments to delegates, see Relaxed Delegate Conversion (Visual Basic).

Related Topics

Title	Description
How to: Invoke a Delegate Method (Visual Basic)	Provides an example that shows how to associate a method with a delegate and then invoke that method through the delegate.
How to: Pass Procedures to Another Procedure in Visual Basic	Demonstrates how to use delegates to pass one procedure to another procedure.
Relaxed Delegate Conversion (Visual Basic)	Describes how you can assign subs and functions to delegates or handlers even when their signatures are not identical
Events (Visual Basic)	Provides an overview of events in Visual Basic.

How to: Invoke a Delegate Method (Visual Basic)

Visual Studio 2015

This example shows how to associate a method with a delegate and then invoke that method through the delegate.

Create the delegate and matching procedures

1. Create a delegate named `MySubDelegate`.

```
Delegate Sub MySubDelegate(ByVal x As Integer)
```

2. Declare a class that contains a method with the same signature as the delegate.

```
Class class1
    Sub Sub1(ByVal x As Integer)
        MsgBox("The value of x is: " & CStr(x))
    End Sub
End Class
```

3. Define a method that creates an instance of the delegate and invokes the method associated with the delegate by calling the built-in **Invoke** method.

```
Protected Sub DelegateTest()
    Dim c1 As New class1
    ' Create an instance of the delegate.
    Dim msd As MySubDelegate = AddressOf c1.Sub1
    ' Call the method.
    msd.Invoke(10)
End Sub
```

See Also

Delegate Statement
Delegates (Visual Basic)
Events (Visual Basic)
Multithreaded Applications (C# and Visual Basic)

How to: Pass Procedures to Another Procedure in Visual Basic

Visual Studio 2015

This example shows how to use delegates to pass a procedure to another procedure.

A delegate is a type that you can use like any other type in Visual Basic. The **AddressOf** operator returns a delegate object when applied to a procedure name.

This example has a procedure with a delegate parameter that can take a reference to another procedure, obtained with the **AddressOf** operator.

Create the delegate and matching procedures

1. Create a delegate named `MathOperator`.

VB

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

2. Create a procedure named `AddNumbers` with parameters and return value that match those of `MathOperator`, so that the signatures match.

VB

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

3. Create a procedure named `SubtractNumbers` with a signature that matches `MathOperator`.

VB

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

4. Create a procedure named `DelegateTest` that takes a delegate as a parameter.

   This procedure can accept a reference to `AddNumbers` or `SubtractNumbers`, because their signatures match the `MathOperator` signature.

   VB

```
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
```

5. Create a procedure named `Test` that calls `DelegateTest` once with the delegate for `AddNumbers` as a parameter, and again with the delegate for `SubtractNumbers` as a parameter.

   VB

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

   When `Test` is called, it first displays the result of `AddNumbers` acting on `5` and `3`, which is 8. Then the result of `SubtractNumbers` acting on `9` and `3` is displayed, which is 6.

See Also

Delegates (Visual Basic)
AddressOf Operator (Visual Basic)
Delegate Statement
How to: Invoke a Delegate Method (Visual Basic)

Relaxed Delegate Conversion (Visual Basic)

Visual Studio 2015

Relaxed delegate conversion enables you to assign subs and functions to delegates or handlers even when their signatures are not identical. Therefore, binding to delegates becomes consistent with the binding already allowed for method invocations.

Parameters and Return Type

In place of exact signature match, relaxed conversion requires that the following conditions be met when **Option Strict** is set to **On**:

- A widening conversion must exist from the data type of each delegate parameter to the data type of the corresponding parameter of the assigned function or **Sub**. In the following example, the delegate `Del1` has one parameter, an **Integer**. Parameter `m` in the assigned lambda expressions must have a data type for which there is a widening conversion from **Integer**, such as **Long** or **Double**.

  VB

  ```
  ' Definition of delegate Del1.
  Delegate Function Del1(ByVal arg As Integer) As Integer
  ```

  VB

  ```
  ' Valid lambda expression assignments with Option Strict on or off:

  ' Integer matches Integer.
  Dim d1 As Del1 = Function(m As Integer) 3

  ' Integer widens to Long
  Dim d2 As Del1 = Function(m As Long) 3

  ' Integer widens to Double
  Dim d3 As Del1 = Function(m As Double) 3
  ```

  Narrowing conversions are permitted only when **Option Strict** is set to **Off**.

  VB

  ```
  ' Valid only when Option Strict is off:

  Dim d4 As Del1 = Function(m As String) CInt(m)
  Dim d5 As Del1 = Function(m As Short) m
  ```

- A widening conversion must exist in the opposite direction from the return type of the assigned function or **Sub** to the return type of the delegate. In the following examples, the body of each assigned lambda expression must evaluate to a data type that widens to **Integer** because the return type of `del1` is **Integer**.

VB

```vb
' Valid return types with Option Strict on:

' Integer matches Integer.
Dim d6 As Del1 = Function(m As Integer) m

' Short widens to Integer.
Dim d7 As Del1 = Function(m As Long) CShort(m)

' Byte widens to Integer.
Dim d8 As Del1 = Function(m As Double) CByte(m)
```

If **Option Strict** is set to **Off**, the widening restriction is removed in both directions.

VB

```vb
' Valid only when Option Strict is set to Off.

' Integer does not widen to Short in the parameter.
Dim d9 As Del1 = Function(n As Short) n

' Long does not widen to Integer in the return type.
Dim d10 As Del1 = Function(n As Integer) CLng(n)
```

Omitting Parameter Specifications

Relaxed delegates also allow you to completely omit parameter specifications in the assigned method:

VB

```vb
' Definition of delegate Del2, which has two parameters.
Delegate Function Del2(ByVal arg1 As Integer, ByVal arg2 As String) As Integer
```

VB

```vb
' The assigned lambda expression specifies no parameters, even though
' Del2 has two parameters. Because the assigned function in this
' example is a lambda expression, Option Strict can be on or off.
' Compare the declaration of d16, where a standard function is assigned.
Dim d11 As Del2 = Function() 3

' The parameters are still there, however, as defined in the delegate.
Console.WriteLine(d11(5, "five"))

' Not valid.
' Console.WriteLine(d11())
' Console.WriteLine(d11(5))
```

Note that you cannot specify some parameters and omit others.

VB

```
' Not valid.
'Dim d12 As Del2 = Function(p As Integer) p
```

The ability to omit parameters is helpful in a situation such as defining an event handler, where several complex parameters are involved. The arguments to some event handlers are not used. Instead, the handler directly accesses the state of the control on which the event is registered, and ignores the arguments. Relaxed delegates allow you to omit the arguments in such declarations when no ambiguities result. In the following example, the fully specified method `OnClick` can be rewritten as `RelaxedOnClick`.

VB

```
Sub OnClick(ByVal sender As Object, ByVal e As EventArgs) Handles b.Click
    MessageBox.Show("Hello World from" + b.Text)
End Sub

Sub RelaxedOnClick() Handles b.Click
    MessageBox.Show("Hello World from" + b.Text)
End Sub
```

AddressOf Examples

Lambda expressions are used in the previous examples to make the type relationships easy to see. However, the same relaxations are permitted for delegate assignments that use **AddressOf**, **Handles**, or **AddHandler**.

In the following example, functions `f1`, `f2`, `f3`, and `f4` can all be assigned to `Del1`.

VB

```
' Definition of delegate Del1.
Delegate Function Del1(ByVal arg As Integer) As Integer
```

VB

```
' Definitions of f1, f2, f3, and f4.
Function f1(ByVal m As Integer) As Integer
End Function

Function f2(ByVal m As Long) As Integer
End Function

Function f3(ByVal m As Integer) As Short
```

```
End Function

Function f4() As Integer
End Function
```

VB

```
' Assignments to function delegate Del1.

' Valid AddressOf assignments with Option Strict on or off:

' Integer parameters of delegate and function match.
Dim d13 As Del1 = AddressOf f1

' Integer delegate parameter widens to Long.
Dim d14 As Del1 = AddressOf f2

' Short return in f3 widens to Integer.
Dim d15 As Del1 = AddressOf f3
```

The following example is valid only when **Option Strict** is set to **Off**.

VB

```
' If Option Strict is Off, parameter specifications for f4 can be omitted.
Dim d16 As Del1 = AddressOf f4

' Function d16 still requires a single argument, however, as specified
' by Del1.
Console.WriteLine(d16(5))

' Not valid.
'Console.WriteLine(d16())
'Console.WriteLine(d16(5, 3))
```

Dropping Function Returns

Relaxed delegate conversion enables you to assign a function to a **Sub** delegate, effectively ignoring the return value of the function. However, you cannot assign a **Sub** to a function delegate. In the following example, the address of function `doubler` is assigned to **Sub** delegate `Del3`.

VB

```
' Definition of Sub delegate Del3.
Delegate Sub Del3(ByVal arg1 As Integer)

' Definition of function doubler, which both displays and returns the
' value of its integer parameter.
Function doubler(ByVal p As Integer) As Integer
```

```
        Dim times2 = 2 * p
        Console.WriteLine("Value of p: " & p)
        Console.WriteLine("Double p:   " & times2)
        Return times2
    End Function
```

VB

```
    ' You can assign the function to the Sub delegate:
    Dim d17 As Del3 = AddressOf doubler

    ' You can then call d17 like a regular Sub procedure.
    d17(5)

    ' You cannot call d17 as a function. It is a Sub, and has no
    ' return value.
    ' Not valid.
    'Console.WriteLine(d17(5))
```

See Also

Lambda Expressions (Visual Basic)
Widening and Narrowing Conversions (Visual Basic)
Delegates (Visual Basic)
How to: Pass Procedures to Another Procedure in Visual Basic
Local Type Inference (Visual Basic)
Option Strict Statement

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)