Expression Trees (Visual Basic)

Visual Studio 2015

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to: Use Expression Trees to Build Dynamic Queries (Visual Basic).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type Expression(Of TDelegate), the compiler emits code to build an expression tree that represents the lambda expression.

The Visual Basic compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in Visual Basic, see Lambda Expressions (Visual Basic).

The following code examples demonstrate how to have the Visual Basic compiler create an expression tree that represents the lambda expression `Function(num) num < 5`.

VB

```
Dim lambda As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the Expression class. This class contains static factory methods that create expression tree nodes of specific types, for example, ParameterExpression, which represents a variable or parameter, or MethodCallExpression, which represents a method call. ParameterExpression, MethodCallExpression, and the other expression-specific types are also defined in the System.Linq.Expressions namespace. These types derive from the abstract type Expression.

The following code example demonstrates how to create an expression tree that represents the lambda expression `Function(num) num < 5` by using the API.

VB

```
' Import the following namespace to your project: System.Linq.Expressions

' Manually build the expression tree for the lambda expression num => num < 5.
Dim numParam As ParameterExpression = Expression.Parameter(GetType(Integer), "num")
Dim five As ConstantExpression = Expression.Constant(5, GetType(Integer))
Dim numLessThanFive As BinaryExpression = Expression.LessThan(numParam, five)
Dim lambda1 As Expression(Of Func(Of Integer, Boolean)) =
  Expression.Lambda(Of Func(Of Integer, Boolean))(
        numLessThanFive,
        New ParameterExpression() {numParam})
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and **try-catch** blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the Visual Basic compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

VB

```
' Creating a parameter expression.
Dim value As ParameterExpression =
    Expression.Parameter(GetType(Integer), "value")

' Creating an expression to hold a local variable.
Dim result As ParameterExpression =
    Expression.Parameter(GetType(Integer), "result")

' Creating a label to jump to from a loop.
Dim label As LabelTarget = Expression.Label(GetType(Integer))

' Creating a method body.
Dim block As BlockExpression = Expression.Block(
    New ParameterExpression() {result},
    Expression.Assign(result, Expression.Constant(1)),
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(value, Expression.Constant(1)),
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            Expression.Break(label, result)
        ),
        label
    )
)

' Compile an expression tree and return a delegate.
Dim factorial As Integer =
    Expression.Lambda(Of Func(Of Integer, Integer))(block, value).Compile()(5)

Console.WriteLine(factorial)
' Prints 120.
```

For more information, see Generating Dynamic Methods with Expression Trees in Visual Studio 2010 (or later).

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression `Function(num) num < 5` can be decomposed into its parts.

VB

```
' Import the following namespace to your project: System.Linq.Expressions

' Create an expression tree.
Dim exprTree As Expression(Of Func(Of Integer, Boolean)) = Function(num) num < 5

' Decompose the expression tree.
Dim param As ParameterExpression = exprTree.Parameters(0)
Dim operation As BinaryExpression = exprTree.Body
Dim left As ParameterExpression = operation.Left
Dim right As ConstantExpression = operation.Right

Console.WriteLine(String.Format("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value))

' This code produces the following output:
'
' Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see How to: Modify Expression Trees (Visual Basic).

Compiling Expression Trees

The Expression(Of TDelegate) type provides the Compile method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

VB

```
' Creating an expression tree.
Dim expr As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5
```

```
' Compiling the expression tree into a delegate.
Dim result As Func(Of Integer, Boolean) = expr.Compile()

' Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4))

' Prints True.

' You can also use simplified syntax
' to compile and run an expression tree.
' The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4))

' Also prints True.
```

For more information, see How to: Execute Expression Trees (Visual Basic).

See Also

System.Linq.Expressions
How to: Execute Expression Trees (Visual Basic)
How to: Modify Expression Trees (Visual Basic)
Lambda Expressions (Visual Basic)
Dynamic Language Runtime Overview
Programming Concepts (Visual Basic)

Dynamic Language Runtime Overview

.NET Framework (current version)

The *dynamic language runtime* (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR). The DLR makes it easier to develop dynamic languages to run on the .NET Framework and to add dynamic features to statically typed languages.

Dynamic languages can identify the type of an object at run time, whereas in statically typed languages such as C# and Visual Basic (when you use **Option Explicit On**) you must specify object types at design time. Examples of dynamic languages are Lisp, Smalltalk, JavaScript, PHP, Ruby, Python, ColdFusion, Lua, Cobra, and Groovy.

Most dynamic languages provide the following advantages for developers:

- The ability to use a rapid feedback loop (REPL, or read-evaluate-print loop). This lets you enter several statements and immediately execute them to see the results.
- Support for both top-down development and more traditional bottom-up development. For example, when you use a top-down approach, you can call functions that are not yet implemented and then add underlying implementations when you need them.
- Easier refactoring and code modifications, because you do not have to change static type declarations throughout the code.

Dynamic languages make excellent scripting languages. Customers can easily extend applications created by using dynamic languages with new commands and functionality. Dynamic languages are also frequently used for creating Web sites and test harnesses, maintaining server farms, developing various utilities, and performing data transformations.

The purpose of the DLR is to enable a system of dynamic languages to run on the .NET Framework and give them .NET interoperability. The DLR introduces dynamic objects to C# and Visual Basic in Visual Studio 2010 to support dynamic behavior in these languages and enable their interoperation with dynamic languages.

The DLR also helps you create libraries that support dynamic operations. For example, if you have a library that uses XML or JavaScript Object Notation (JSON) objects, your objects can appear as dynamic objects to languages that use the DLR. This lets library users write syntactically simpler and more natural code for operating with objects and accessing object members.

For example, you might use the following code to increment a counter in XML in C#.

```
Scriptobj.SetProperty("Count", ((int)GetProperty("Count")) + 1);
```

By using the DLR, you could use the following code instead for the same operation.

```
scriptobj.Count += 1;
```

Like the CLR, the DLR is a part of the .NET Framework and is provided with the .NET Framework and Visual Studio installation packages. The open-source version of the DLR is also available for download on the CodePlex Web site.

Note

The open-source version of the DLR has all the features of the DLR that is included in Visual Studio and the .NET Framework. It also provides additional support for language implementers. For more information, see the documentation on the CodePlex Web site.

Examples of languages developed by using the DLR include the following:

- IronPython. Available as open-source software from the CodePlex Web site.
- IronRuby. Available as open-source software from the RubyForge Web site.

Primary DLR Advantages

The DLR provides the following advantages.

Simplifies Porting Dynamic Languages to the .NET Framework

The DLR allows language implementers to avoid creating lexical analyzers, parsers, semantic analyzers, code generators, and other tools that they traditionally had to create themselves. To use the DLR, a language needs to produce *expression trees*, which represent language-level code in a tree-shaped structure, runtime helper routines, and optional dynamic objects that implement the IDynamicMetaObjectProvider interface. The DLR and the .NET Framework automate a lot of code analysis and code generation tasks. This enables language implementers to concentrate on unique language features.

Enables Dynamic Features in Statically Typed Languages

Existing .NET Framework languages such as C# and Visual Basic can create dynamic objects and use them together with statically typed objects. For example, C# and Visual Basic can use dynamic objects for HTML, Document Object Model (DOM), and .NET reflection.

Provides Future Benefits of the DLR and .NET Framework

Languages implemented by using the DLR can benefit from future DLR and .NET Framework improvements. For example, if the .NET Framework releases a new version that has an improved garbage collector or faster assembly loading time, languages implemented by using the DLR immediately get the same benefit. If the DLR adds optimizations such as better compilation, the performance also improves for all languages implemented by using the DLR.

Enables Sharing of Libraries and Objects

The objects and libraries implemented in one language can be used by other languages. The DLR also enables interoperation between statically typed and dynamic languages. For example, C# can declare a dynamic object that uses a library that is written in a dynamic language. At the same time, dynamic languages can use libraries from the

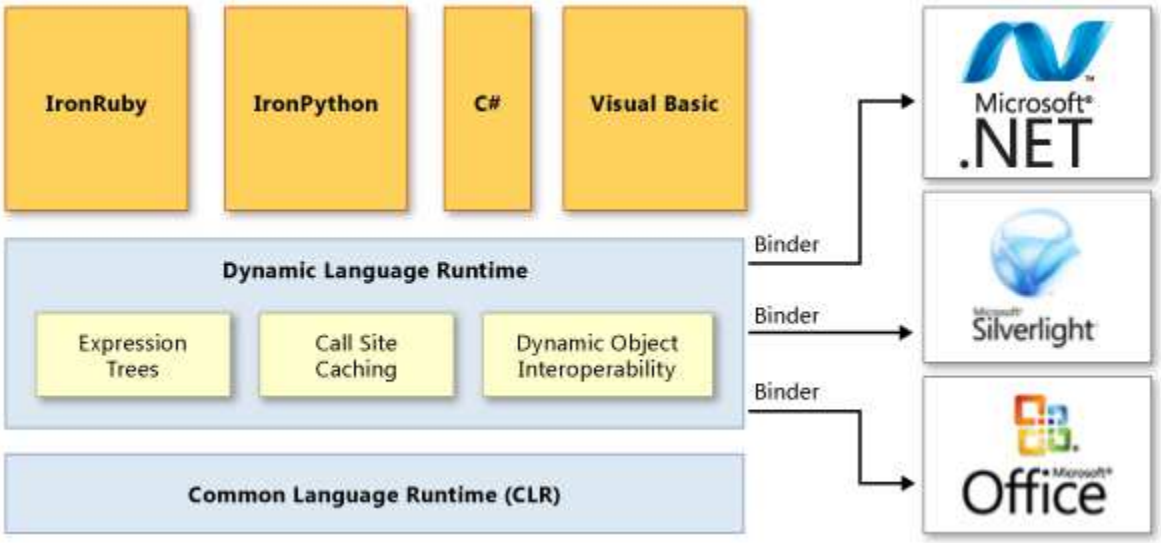
.NET Framework.

Provides Fast Dynamic Dispatch and Invocation

The DLR provides fast execution of dynamic operations by supporting advanced polymorphic caching. The DLR creates rules for binding operations that use objects to the necessary runtime implementations and then caches these rules to avoid resource-exhausting binding computations during successive executions of the same code on the same types of objects.

DLR Architecture

The following illustration shows the architecture of the dynamic language runtime.



DLR architecture

The DLR adds a set of services to the CLR for better supporting dynamic languages. These services include the following:

- Expression trees. The DLR uses expression trees to represent language semantics. For this purpose, the DLR has extended LINQ expression trees to include control flow, assignment, and other language-modeling nodes. For more information, see Expression Trees (C# and Visual Basic).
- Call site caching. A *dynamic call site* is a place in the code where you perform an operation like `a + b` or `a.b()` on dynamic objects. The DLR caches the characteristics of `a` and `b` (usually the types of these objects) and information about the operation. If such an operation has been performed previously, the DLR retrieves all the necessary information from the cache for fast dispatch.
- Dynamic object interoperability. The DLR provides a set of classes and interfaces that represent dynamic objects and operations and can be used by language implementers and authors of dynamic libraries. These classes and interfaces include IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject, and ExpandoObject.

The DLR uses binders in call sites to communicate not only with the .NET Framework, but with other infrastructures and

services, including Silverlight and COM. Binders encapsulate a language's semantics and specify how to perform operations in a call site by using expression trees. This enables dynamic and statically typed languages that use the DLR to share libraries and gain access to all the technologies that the DLR supports.

DLR Documentation

For more information about how to use the open source version of the DLR to add dynamic behavior to a language, or about how to enable the use of a dynamic language with the .NET Framework, see the documentation on the CodePlex Web site.

See Also

ExpandoObject
DynamicObject
Common Language Runtime (CLR)
Expression Trees (C# and Visual Basic)
Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

Visual Studio 2015

Dynamic objects expose members such as properties and methods at run time, instead of in at compile time. This enables you to create objects to work with structures that do not match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the **GetProperty** method of the element. To reference the **id** attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the **id** attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script that is interpreted at run time.

You reference a dynamic object by using late binding. In C#, you specify the type of a late-bound object as **dynamic**. In Visual Basic, you specify the type of a late-bound object as **Object**. For more information, see dynamic (C# Reference) and Early and Late Binding (Visual Basic).

You can create custom dynamic objects by using the classes in the System.Dynamic namespace. For example, you can create an ExpandoObject and specify the members of that object at run time. You can also create your own type that inherits the DynamicObject class. You can then override the members of the DynamicObject class to provide run-time dynamic functionality.

In this walkthrough you will perform the following tasks:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an **IronPython** library.

Prerequisites

You need IronPython 2.6.1 for .NET 4.0 to complete this walkthrough. You can download IronPython 2.6.1 for .NET 4.0 from CodePlex.

Note

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

Creating a Custom Dynamic Object

The first project that you create in this walkthrough defines a custom dynamic object that searches the contents of a text file. Text to search for is specified by the name of a dynamic property. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies `dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and does not remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

To create a custom dynamic class

1. Start Visual Studio.
2. On the **File** menu, point to **New** and then click **Project**.
3. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type **DynamicSample**, and then click **OK**. The new project is created.
4. Right-click the DynamicSample project and point to **Add**, and then click **Class**. In the **Name** box, type **ReadOnlyFile**, and then click **OK**. A new file is added that contains the ReadOnlyFile class.
5. At the top of the ReadOnlyFile.cs or ReadOnlyFile.vb file, add the following code to import the System.IO and System.Dynamic namespaces.

VB

```
Imports System.IO
Imports System.Dynamic
```

6. The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

VB

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. Update the class statement to inherit the **DynamicObject** class, as shown in the following code example.

VB

```
Public Class ReadOnlyFile
```

```
    Inherits DynamicObject
```

8. Add the following code to the **ReadOnlyFile** class to define a private field for the file path and a constructor for the **ReadOnlyFile** class.

VB

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

9. Add the following **GetPropertyValue** method to the **ReadOnlyFile** class. The **GetPropertyValue** method takes, as input, search criteria and returns the lines from a text file that match that search criteria. The dynamic methods provided by the **ReadOnlyFile** class call the **GetPropertyValue** method to retrieve their respective results.

VB

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As
StringSearchOption = StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As
List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search
options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then
results.Add(line)
```

```
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then
results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then
results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

10. After the **GetPropertyValue** method, add the following code to override the TryGetMember method of the DynamicObject class. The TryGetMember method is called when a member of a dynamic class is requested and no arguments are specified. The **binder** argument contains information about the referenced member, and the **result** argument references the result returned for the specified member. The TryGetMember method returns a Boolean value that returns **true** if the requested member exists; otherwise it returns **false**.

VB

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member
calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                        ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

11. After the **TryGetMember** method, add the following code to override the TryInvokeMember method of the DynamicObject class. The TryInvokeMember method is called when a member of a dynamic class is requested with arguments. The **binder** argument contains information about the referenced member, and the **result** argument references the result returned for the specified member. The **args** argument contains an array of the arguments that are passed to the member. The TryInvokeMember method returns a Boolean value that returns **true** if the requested member exists; otherwise it returns **false**.

    The custom version of the **TryInvokeMember** method expects the first argument to be a value from the **StringSearchOption** enum that you defined in a previous step. The **TryInvokeMember** method expects the second argument to be a Boolean value. If one or both arguments are valid values, they are passed to the **GetPropertyValue** method to retrieve the results.

VB

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
```

```
                                        ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0),
StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

12. Save and close the file.

To create a sample text file

1. Right-click the DynamicSample project and point to **Add**, and then click **New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of TextFile1.txt in the **Name** box, and then click **Add**. A new text file is added to the project.

2. Copy the following text to the TextFile1.txt file.

```
List of customers and suppliers

Supplier: Lucerne Publishing (http://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (http://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (http://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Save and close the file.

To create a sample application that uses the custom dynamic object

1. In **Solution Explorer**, double-click the Module1.vb file if you are using Visual Basic or the Program.cs file if you are using Visual C#.

2. Add the following code to the Main procedure to create an instance of the **ReadOnlyFile** class for the TextFile1.txt file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

VB

```
Dim rFile As Object = New ReadOnlyFile("..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Save the file and press CTRL+F5 to build and run the application.

Calling a Dynamic Language Library

The next project that you create in this walkthrough accesses a library that is written in the dynamic language IronPython. Before you create this project, you must have IronPython 2.6.1 for .NET 4.0 installed. You can download IronPython 2.6.1 for .NET 4.0 from CodePlex.

To create a custom dynamic class

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.

2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type **DynamicIronPythonSample**, and then click **OK**. The new project is created.

3. If you are using Visual Basic, right-click the DynamicIronPythonSample project and then click **Properties**. Click the **References** tab. Click the **Add** button. If you are using Visual C#, in **Solution Explorer**, right-click the **References** folder and then click **Add Reference**.

4. On the **Browse** tab, browse to the folder where the IronPython libraries are installed. For example, C:\Program Files\IronPython 2.6 for .NET 4.0. Select the **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll**, and **Microsoft.Dynamic.dll** libraries. Click **OK**.

5. If you are using Visual Basic, edit the Module1.vb file. If you are using Visual C#, edit the Program.cs file.

6. At the top of the file, add the following code to import the **Microsoft.Scripting.Hosting** and **IronPython.Hosting** namespaces from the IronPython libraries.

VB

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
```

7. In the Main method, add the following code to create a new **Microsoft.Scripting.Hosting.ScriptRuntime** object to host the IronPython libraries. The **ScriptRuntime** object loads the IronPython library module random.py.

VB

```
' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
    My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
    "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

8. After the code to load the random.py module, add the following code to create an array of integers. The array is passed to the **shuffle** method of the random.py module, which randomly sorts the values in the array.

VB

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("--------------------")
Next
```

9. Save the file and press CTRL+F5 to build and run the application.

See Also

System.Dynamic
System.Dynamic.DynamicObject
Using Type dynamic (C# Programming Guide)
Early and Late Binding (Visual Basic)
dynamic (C# Reference)
Implementing Dynamic Interfaces (external blog)

Early and Late Binding (Visual Basic)

Visual Studio 2015

The Visual Basic compiler performs a process called *binding* when an object is assigned to an object variable. An object is *early bound* when it is assigned to a variable declared to be of a specific object type. Early bound objects allow the compiler to allocate memory and perform other optimizations before an application executes. For example, the following code fragment declares a variable to be of type FileStream:

VB

```
'  Create a variable to hold a new object.
Dim FS As System.IO.FileStream
' Assign a new object to the variable.
FS = New System.IO.FileStream("C:\tmp.txt",
    System.IO.FileMode.Open)
```

Because FileStream is a specific object type, the instance assigned to FS is early bound.

By contrast, an object is *late bound* when it is assigned to a variable declared to be of type **Object**. Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects. For example, the following code fragment declares an object variable to hold an object returned by the **CreateObject** function:

VB

```
' To use this example, you must have Microsoft Excel installed on your computer.
' Compile with Option Strict Off to allow late binding.
Sub TestLateBinding()
    Dim xlApp As Object
    Dim xlBook As Object
    Dim xlSheet As Object
    xlApp = CreateObject("Excel.Application")
    ' Late bind an instance of an Excel workbook.
    xlBook = xlApp.Workbooks.Add
    ' Late bind an instance of an Excel worksheet.
    xlSheet = xlBook.Worksheets(1)
    xlSheet.Activate()
    ' Show the application.
    xlSheet.Application.Visible = True
    ' Place some text in the second row of the sheet.
    xlSheet.Cells(2, 2) = "This is column B row 2"
End Sub
```

Advantages of Early Binding

You should use early-bound objects whenever possible, because they allow the compiler to make important

optimizations that yield more efficient applications. Early-bound objects are significantly faster than late-bound objects and make your code easier to read and maintain by stating exactly what kind of objects are being used. Another advantage to early binding is that it enables useful features such as automatic code completion and Dynamic Help because the Visual Studio integrated development environment (IDE) can determine exactly what type of object you are working with as you edit the code. Early binding reduces the number and severity of run-time errors because it allows the compiler to report errors when a program is compiled.

Note

Late binding can only be used to access type members that are declared as **Public**. Accessing members declared as **Friend** or **Protected Friend** results in a run-time error.

See Also

CreateObject
Object Lifetime: How Objects Are Created and Destroyed (Visual Basic)
Object Data Type

How to: Execute Expression Trees (Visual Basic)

Visual Studio 2015

This topic shows you how to execute an expression tree. Executing an expression tree may return a value, or it may just perform an action such as calling a method.

Only expression trees that represent lambda expressions can be executed. Expression trees that represent lambda expressions are of type LambdaExpression or Expression(Of TDelegate). To execute these expression trees, call the Compile method to create an executable delegate, and then invoke the delegate.

Note

If the type of the delegate is not known, that is, the lambda expression is of type LambdaExpression and not Expression(Of TDelegate), you must call the DynamicInvoke method on the delegate instead of invoking it directly.

If an expression tree does not represent a lambda expression, you can create a new lambda expression that has the original expression tree as its body, by calling the Lambda(Of TDelegate)(Expression, IEnumerable(Of ParameterExpression)) method. Then, you can execute the lambda expression as described earlier in this section.

Example

The following code example demonstrates how to execute an expression tree that represents raising a number to a power by creating a lambda expression and executing it. The result, which represents the number raised to the power, is displayed.

VB

```
' The expression tree to execute.
Dim be As BinaryExpression = Expression.Power(Expression.Constant(2.0R),
Expression.Constant(3.0R))

' Create a lambda expression.
Dim le As Expression(Of Func(Of Double)) = Expression.Lambda(Of Func(Of Double))(be)

' Compile the lambda expression.
Dim compiledExpression As Func(Of Double) = le.Compile()

' Execute the lambda expression.
Dim result As Double = compiledExpression()

' Display the result.
MsgBox(result)

' This code produces the following output:
```

```
' 8
```

Compiling the Code

- Add a project reference to System.Core.dll if it is not already referenced.
- Include the System.Linq.Expressions namespace.

See Also

Expression Trees (Visual Basic)
How to: Modify Expression Trees (Visual Basic)

How to: Modify Expression Trees (Visual Basic)

Visual Studio 2015

This topic shows you how to modify an expression tree. Expression trees are immutable, which means that they cannot be modified directly. To change an expression tree, you must create a copy of an existing expression tree and when you create the copy, make the required changes. You can use the ExpressionVisitor class to traverse an existing expression tree and to copy each node that it visits.

To modify an expression tree

1. Create a new **Console Application** project.

2. Add an **Imports** statement to the file for the `System.Linq.Expressions` namespace.

3. Add the `AndAlsoModifier` class to your project.

VB

```
Public Class AndAlsoModifier
    Inherits ExpressionVisitor

    Public Function Modify(ByVal expr As Expression) As Expression
        Return Visit(expr)
    End Function

    Protected Overrides Function VisitBinary(ByVal b As BinaryExpression) As
Expression
        If b.NodeType = ExpressionType.AndAlso Then
            Dim left = Me.Visit(b.Left)
            Dim right = Me.Visit(b.Right)

            ' Make this binary expression an OrElse operation instead
            ' of an AndAlso operation.
            Return Expression.MakeBinary(ExpressionType.OrElse, left, right, _
                                         b.IsLiftedToNull, b.Method)
        End If

        Return MyBase.VisitBinary(b)
    End Function
End Class
```

This class inherits the ExpressionVisitor class and is specialized to modify expressions that represent conditional **AND** operations. It changes these operations from a conditional **AND** to a conditional **OR**. To do this, the class overrides the VisitBinary method of the base type, because conditional **AND** expressions are represented as binary expressions.

In the `VisitBinary` method, if the expression that is passed to it represents a conditional **AND** operation, the code constructs a new expression that contains the conditional **OR** operator instead of the conditional **AND** operator. If the expression that is passed to `VisitBinary` does not represent a conditional **AND** operation, the method defers to the base class implementation. The base class methods construct nodes that are like the expression trees that are passed in, but the nodes have their sub trees replaced with the expression trees that are produced recursively by the visitor.

4. Add an **Imports** statement to the file for the `System.Linq.Expressions` namespace.

5. Add code to the `Main` method in the Module1.vb file to create an expression tree and pass it to the method that will modify it.

VB

```
Dim expr As Expression(Of Func(Of String, Boolean)) = _
    Function(name) name.Length > 10 AndAlso name.StartsWith("G")

Console.WriteLine(expr)

Dim modifier As New AndAlsoModifier()
Dim modifiedExpr = modifier.Modify(CType(expr, Expression))

Console.WriteLine(modifiedExpr)

' This code produces the following output:
' name => ((name.Length > 10) && name.StartsWith("G"))
' name => ((name.Length > 10) || name.StartsWith("G"))
```

The code creates an expression that contains a conditional **AND** operation. It then creates an instance of the `AndAlsoModifier` class and passes the expression to the `Modify` method of this class. Both the original and the modified expression trees are outputted to show the change.

6. Compile and run the application.

See Also

How to: Execute Expression Trees (Visual Basic)
Expression Trees (Visual Basic)

How to: Use Expression Trees to Build Dynamic Queries (Visual Basic)

Visual Studio 2015

In LINQ, expression trees are used to represent structured queries that target sources of data that implement IQueryable(Of T). For example, the LINQ provider implements the IQueryable(Of T) interface for querying relational data stores. The Visual Basic compiler compiles queries that target such data sources into code that builds an expression tree at runtime. The query provider can then traverse the expression tree data structure and translate it into a query language appropriate for the data source.

Expression trees are also used in LINQ to represent lambda expressions that are assigned to variables of type Expression(Of TDelegate).

This topic describes how to use expression trees to create dynamic LINQ queries. Dynamic queries are useful when the specifics of a query are not known at compile time. For example, an application might provide a user interface that enables the end user to specify one or more predicates to filter the data. In order to use LINQ for querying, this kind of application must use expression trees to create the LINQ query at runtime.

Example

The following example shows you how to use expression trees to construct a query against an **IQueryable** data source and then execute it. The code builds an expression tree to represent the following query:

```
companies.Where(Function(company) company.ToLower() = "coho winery" OrElse company.Length >
16).OrderBy(Function(company) company)
```

The factory methods in the System.Linq.Expressions namespace are used to create expression trees that represent the expressions that make up the overall query. The expressions that represent calls to the standard query operator methods refer to the Queryable implementations of these methods. The final expression tree is passed to the CreateQuery(Of TElement)(Expression) implementation of the provider of the **IQueryable** data source to create an executable query of type **IQueryable**. The results are obtained by enumerating that query variable.

VB

```
' Add an Imports statement for System.Linq.Expressions.

Dim companies =
    {"Consolidated Messenger", "Alpine Ski House", "Southridge Video", "City Power &
Light",
     "Coho Winery", "Wide World Importers", "Graphic Design Institute", "Adventure
Works",
     "Humongous Insurance", "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
     "Blue Yonder Airlines", "Trey Research", "The Phone Company",
     "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"}

' The IQueryable data to query.
Dim queryableData As IQueryable(Of String) = companies.AsQueryable()
```

```
' Compose the expression tree that represents the parameter to the predicate.
Dim pe As ParameterExpression = Expression.Parameter(GetType(String), "company")

' ***** Where(Function(company) company.ToLower() = "coho winery" OrElse company.Length >
16) *****
' Create an expression tree that represents the expression: company.ToLower() = "coho
winery".
Dim left As Expression = Expression.Call(pe, GetType(String).GetMethod("ToLower",
System.Type.EmptyTypes))
Dim right As Expression = Expression.Constant("coho winery")
Dim e1 As Expression = Expression.Equal(left, right)

' Create an expression tree that represents the expression: company.Length > 16.
left = Expression.Property(pe, GetType(String).GetProperty("Length"))
right = Expression.Constant(16, GetType(Integer))
Dim e2 As Expression = Expression.GreaterThan(left, right)

' Combine the expressions to create an expression tree that represents the
' expression: company.ToLower() = "coho winery" OrElse company.Length > 16).
Dim predicateBody As Expression = Expression.OrElse(e1, e2)

' Create an expression tree that represents the expression:
' queryableData.Where(Function(company) company.ToLower() = "coho winery" OrElse
company.Length > 16)
Dim whereCallExpression As MethodCallExpression = Expression.Call(
        GetType(Queryable),
        "Where",
        New Type() {queryableData.ElementType},
        queryableData.Expression,
        Expression.Lambda(Of Func(Of String, Boolean))(predicateBody, New
ParameterExpression() {pe}))
' ***** End Where *****

' ***** OrderBy(Function(company) company) *****
' Create an expression tree that represents the expression:
' whereCallExpression.OrderBy(Function(company) company)
Dim orderByCallExpression As MethodCallExpression = Expression.Call(
        GetType(Queryable),
        "OrderBy",
        New Type() {queryableData.ElementType, queryableData.ElementType},
        whereCallExpression,
        Expression.Lambda(Of Func(Of String, String))(pe, New ParameterExpression() {pe}))
' ***** End OrderBy *****

' Create an executable query from the expression tree.
Dim results As IQueryable(Of String) = queryableData.Provider.CreateQuery(Of String)
(orderByCallExpression)

' Enumerate the results.
For Each company As String In results
    Console.WriteLine(company)
Next
```

```
' This code produces the following output:
'
' Blue Yonder Airlines
' City Power & Light
' Coho Winery
' Consolidated Messenger
' Graphic Design Institute
' Humongous Insurance
' Lucerne Publishing
' Northwind Traders
' The Phone Company
' Wide World Importers
```

This code uses a fixed number of expressions in the predicate that is passed to the `Queryable.Where` method. However, you can write an application that combines a variable number of predicate expressions that depends on the user input. You can also vary the standard query operators that are called in the query, depending on the input from the user.

Compiling the Code

- Create a new **Console Application** project.
- Add a reference to System.Core.dll if it is not already referenced.
- Include the System.Linq.Expressions namespace.
- Copy the code from the example and paste it into the `Main` **Sub** procedure.

See Also

Expression Trees (Visual Basic)
How to: Execute Expression Trees (Visual Basic)

Debugging Expression Trees in Visual Studio (Visual Basic)

Visual Studio 2015

You can analyze the structure and content of expression trees when you debug your applications. To get a quick overview of the expression tree structure, you can use the **DebugView** property, which is available only in debug mode. For more information about debugging, see Debugging in Visual Studio.

To better represent the content of expression trees, the **DebugView** property uses Visual Studio visualizers. For more information, see Create Custom Visualizers of Data.

To open a visualizer for an expression tree

1. Click the magnifying glass icon that appears next to the **DebugView** property of an expression tree in **DataTips**, a **Watch** window, the **Autos** window, or the **Locals** window.

   A list of visualizers is displayed.

2. Click the visualizer you want to use.

Each expression type is displayed in the visualizer as described in the following sections.

ParameterExpressions

ParameterExpression variable names are displayed with a "$" symbol at the beginning.

If a parameter does not have a name, it is assigned an automatically generated name, such as `$var1` or `$var2`.

Examples

- **Expression**

  VB

  ```
  Dim numParam As ParameterExpression =
  Expression.Parameter(GetType(Integer), "num")
  ```

  DebugView property

  ```
  $num
  ```

- **Expression**

  VB

```
Dim numParam As ParameterExpression =
Expression.Parameter(GetType(Integer))
```

DebugView property

```
$var1
```

ConstantExpressions

For ConstantExpression objects that represent integer values, strings, and **null**, the value of the constant is displayed.

Examples

- **Expression**

  VB

  ```
  Dim num as Integer= 10
  Dim expr As ConstantExpression = Expression.Constant(num)
  ```

  DebugView property

  ```
  10
  ```

- **Expression**

  VB

  ```
  Dim num As Double = 10
  Dim expr As ConstantExpression = Expression.Constant(num)
  ```

  DebugView property

  ```
  10D
  ```

BlockExpression

If the type of a BlockExpression object differs from the type of the last expression in the block, the type is displayed in the **DebugInfo** property in angle brackets (< and >). Otherwise, the type of the BlockExpression object is not displayed.

Examples

- **Expression**

```
VB
Dim block As BlockExpression = Expression.Block(Expression.Constant("test"))
```

DebugView property

```
.Block() {

"test"

}
```

- **Expression**

```
VB
Dim block As BlockExpression =
Expression.Block(GetType(Object), Expression.Constant("test"))
```

DebugView property

```
.Block<System.Object>() {

"test"

}
```

LambdaExpression

LambdaExpression objects are displayed together with their delegate types.

If a lambda expression does not have a name, it is assigned an automatically generated name, such as `#Lambda1` or `#Lambda2`.

Examples

- **Expression**

```
VB
Dim lambda As LambdaExpression =
Expression.Lambda(Of Func(Of Integer))(Expression.Constant(1))
```

DebugView property

```
.Lambda #Lambda1<System.Func'1[System.Int32]>() {

1

}
```

- **Expression**

  VB

  ```
  Dim lambda As LambdaExpression =
  Expression.Lambda(Of Func(Of Integer))(Expression.Constant(1), "SampleLamda",
  Nothing)
  ```

  DebugView property

  ```
  .Lambda SampleLambda<System.Func'1[System.Int32]>() {

  1


  }
  ```

LabelExpression

If you specify a default value for the LabelExpression object, this value is displayed before the LabelTarget object.

The `.Label` token indicates the start of the label. The `.LabelTarget` token indicates the destination of the target to jump to.

If a label does not have a name, it is assigned an automatically generated name, such as `#Label1` or `#Label2`.

Examples

- **Expression**

  VB

  ```
  Dim target As LabelTarget = Expression.Label(GetType(Integer), "SampleLabel")
  Dim label1 As BlockExpression = Expression.Block(
  Expression.Goto(target, Expression.Constant(0)),
  Expression.Label(target, Expression.Constant(-1)))
  ```

  DebugView property

  ```
  .Block() {
  ```

```
.Goto SampleLabel { 0 };

.Label

-1

.LabelTarget SampleLabel:

}
```

- **Expression**

  VB

  ```
  Dim target As LabelTarget = Expression.Label()
  Dim block As BlockExpression = Expression.Block(
  Expression.Goto(target), Expression.Label(target))
  ```

  DebugView property

  ```
  .Block() {

  .Goto #Label1 { };

  .Label

  .LabelTarget #Label1:

  }
  ```

Checked Operators

Checked operators are displayed with the "#" symbol in front of the operator. For example, the checked addition operator is displayed as `#+`.

Examples

- **Expression**

  VB

  ```
  Dim expr As Expression = Expression.AddChecked(
  Expression.Constant(1), Expression.Constant(2))
  ```

  DebugView property

  ```
  1 #+ 2
  ```

- **Expression**

```vb
Dim expr As Expression = Expression.ConvertChecked(
Expression.Constant(10.0), GetType(Integer))
```

DebugView property

```
#(System.Int32)10D
```

See Also

Expression Trees (Visual Basic)
Debugging in Visual Studio
Create Custom Visualizers of Data