

Invoking IEnumerable Members

Most of the time, you don't need to call the IEnumerable interface explicitly and you just let Visual Basic call it for you when the object appears in a For Each loop. However, calling GetEnumerator to get the IEnumerator object and then invoking the enumerator's methods directly sometimes makes sense. When you want to retrieve all the elements from an enumerator but can't do that inside a loop would be such an occasion.

The following FileTree class is a nontrivial example of this concept. This class lets you iterate over all the files in a directory tree and internally uses the System.IO.Directory class to enumerate the files and the subdirectory in a given path.

```
Class FileTree
    Implements IEnumerable

    ' The search path
    Public ReadOnly DirPath As String

    ' The constructor
    Sub New(ByVal DirPath As String)
        Me.DirPath = DirPath
    End Sub

    ' Return an Enumerable object (an instance of the inner
    ' class).
    Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator
        Return New FileTreeEnumerator(DirPath)
    End Function

    ' The IEnumerator private object
    Class FileTreeEnumerator
        Implements IEnumerator

        Dim DirPath As String

        ' This variable contains the Enumerator object for the
        ' file list in the directory being scanned.
        Dim FileEnumerator As IEnumerator
        ' This variable contains the stack of the Enumerator
        ' objects for subdirectories of all pending directories.
        Dim DirEnumerators As New System.Collections.Stack()

        ' A simple constructor
        Sub New(ByVal DirPath As String)
            ' Save the directory path.
            Me.DirPath = DirPath
            ' Manually call the Reset method.
            Reset()
        End Sub

        Sub Reset() Implements IEnumerator.Reset
            ' The DirectoryInfo object that represents the
            ' root object
            Dim di As New System.IO.DirectoryInfo(DirPath)
```

```

' Get the Enumerator object for the file list,
' and reset it.
FileEnumerator = di.GetFiles.GetEnumerator
FileEnumerator.Reset()

' Get the Enumerator object for the subdirectory
' list, and reset it.
Dim dirEnum As IEnumerator = _
    di.GetDirectories.GetEnumerator
dirEnum.Reset()
' Push it onto the stack.
DirEnumerators.Push(dirEnum)
End Sub

Function MoveNext() As Boolean Implements _
    IEnumerator.MoveNext
' Simply delegate to the File Enumerator object.
If FileEnumerator.MoveNext Then
    ' It returned True, so we can exit.
    Return True
End If

' If there are no files in the current directory,
' check for another subdirectory in the current
' directory. First get the current directory
' enumerator.
Dim dirEnum As IEnumerator = _
    CType(DirEnumerators.Peek, IEnumerator)
' Check whether current subdirectory enumerator has
' more items.
Do Until dirEnum.MoveNext
    ' There are no more subdirectories at this level,
    ' so we must pop another element of the stack.
    DirEnumerators.Pop()

    If DirEnumerators.Count = 0 Then
        ' Return False if no more subdirectories to
        ' scan.
        Return False
    End If
    ' Get the current enumerator.
    dirEnum = CType(DirEnumerators.Peek, IEnumerator)
Loop

' We can create a DirectoryInfo.
Dim di As System.IO.DirectoryInfo = _
    CType(dirEnum.Current, System.IO.DirectoryInfo)

' Store the file enumerator, and reset it.
FileEnumerator = di.GetFiles.GetEnumerator
FileEnumerator.Reset()
' Get the Enumerator object for the subdirectory
' list, and reset it.
dirEnum = di.GetDirectories.GetEnumerator
dirEnum.Reset()
' Push it onto the stack.
DirEnumerators.Push(dirEnum)

' Recursive call, to process the file enumerator
Return Me.MoveNext
End Function

```

```

        ' The Current property simply delegates to
        ' FileEnumerator.Current.
        ReadOnly Property Current() As Object Implements _
            IEnumerator.Current
            Get
                Return FileEnumerator.Current
            End Get
        End Property
    End Class

End Class

```

Let's analyze the preceding code. The `FileTree` class implements `IEnumerable` and exposes a `GetEnumerator` method, which does nothing but return an instance of the inner, private, `FileTreeEnumerator` class, which in turn implements `IEnumerator`. Each time the client code invokes the `MoveNext` method in `FileTreeEnumerator`, the code in the class must return `True` and be ready to return a filename through the `Current` property or return `False` to stop enumeration when there are no more files to scan in the directory tree.

The `FileTreeEnumerator` class delegates its job to two different types of enumerator objects, one for iterating over all the files in a given directory and the other for enumerating all the subdirectories in a given directory. There is only one file enumerator object in any given moment, and it's stored in the `FileEnumerator` variable. However, you need to store several directory enumerator objects because you're traversing a directory tree, and you need to save the enumerator for all the directories above the one you're parsing currently. For this reason, instead of a single variable, you maintain directory enumerators in a `Stack` structure named `DirEnumerators`.

When the client calls the `MoveNext` method—either explicitly or implicitly through a `For Each` loop—the `FileTreeEnumerator` class attempts to delegate its job to the `FileEnumerator.MoveNext` method. If this method returns `True`, there's nothing more to do, and you can exit:

```

Function MoveNext() As Boolean Implements IEnumerator.MoveNext
    ' Simply delegate to the File Enumerator object.
    If FileEnumerator.MoveNext Then
        ' It returned True, so we can exit.
        Return True
    End If

```

If the `FileEnumerator.MoveNext` method returns `False`, there are no more files in the directory being scanned, so you must retrieve the directory enumerator at the top of the `DirEnumerators` stack (but without popping it off the stack) and invoke its `MoveNext` method. If this method returns `False`, it means that you've already scanned all the subdirectories in the current directory, so you can pop one element off the `DirEnumerators` stack and try again:

```

    ' First get the current directory enumerator.
    Dim dirEnum As IEnumerator = _
        CType(DirEnumerators.Peek, IEnumerator)
    ' Check whether current subdirectory enumerator has
    ' more items.
    Do Until dirEnum.MoveNext

```

```

    ' There are no more subdirectories at this level,
    ' so we must pop another element off the stack.
    DirEnumerators.Pop()

    If DirEnumerators.Count = 0 Then
        ' Return False if no more subdirectories to scan.
        Return False
    End If
    ' Get the current enumerator.
    dirEnum = CType(DirEnumerators.Peek, IEnumerator)
Loop

```

You can exit the preceding loop for only two reasons: there are no more items in `DirEnumerators` (in which case you've visited all the directories in the tree and can return `False` to the client program), or you've found a directory enumerator whose `MoveNext` method returned `True`, which means that you've found a directory that hasn't been scanned yet. In the latter case, you can reinitialize `FileEnumerator` with the enumerator returned by the `GetFiles` method and then get the enumerator returned by the `GetDirectories` method to push onto the `DirEnumerators` stack:

```

    ' Store the file enumerator, and reset it.
    FileEnumerator = di.GetFiles.GetEnumerator
    FileEnumerator.Reset()
    ' Get the Enumerator object for the subdirectory list,
    ' and reset it.
    dirEnum = di.GetDirectories.GetEnumerator
    dirEnum.Reset()
    ' Push it onto the stack.
    DirEnumerators.Push(dirEnum)

```

The last thing to do is recursively call the `MoveNext` method to correctly process the enumerator, which is now contained in the `FileEnumerator` variable:

```

    ' Recursive call, to process the file enumerator
    Return Me.MoveNext
End Function

```

By comparison, the code in the `Current` property is really simple because it only has to delegate to the `FileEnumerator.Current` property:

```

ReadOnly Property Current() As Object Implements _
    IEnumerator.Current
    Get
        Return FileEnumerator.Current
    End Get
End Property

```

The `FileTree` class uses the `Directory` and `DirectoryInfo` classes in the `System.IO` namespace, which I describe only in Chapter 9, so you might not understand every single detail of its inner workings. Using the `FileTree` class, on the other hand, couldn't be simpler:

```

Sub TestGetEnumerator()
    Dim f As System.IO.FileInfo
    ' Enumerate all files in the C:\DOCS directory tree.
    For Each f In New FileTree("C:\DOCS")
        Console.WriteLine(f.FullName)
    Next
End Sub

```