Windows Forms Solutions

SOLUTION 1

ListBox ItemData Is Gone!

PROBLEM Classic VB ListBoxes had an `ItemData` property that let you associate an item in a ListBox with something else, such as an ID value for a row in a database table, or an index for an array of items. But .NET ListBoxes don't have an `ItemData` property. How can I make that association now?

SOLUTION Place your items in a class. When you do that, you often don't need an index or ID number, because the items are directly available from the ListBox's Items collection.

The look of those familiar VB ListBoxes and ComboBoxes hasn't changed, but the way they work has changed dramatically. For those of you just getting started with .NET, dealing with ListBoxes and ComboBoxes is often one of the first sources of serious frustration. But don't worry. In 10 minutes you can absorb the basic workings of the new .NET ListBoxes and ComboBoxes, and you'll never miss `ItemData` again.

NOTE For the rest of this solution, I'll limit the discussion to ListBoxes, but all the information in this solution works with both ComboBoxes and ListBoxes.

The data model for classic VB ListBoxes consisted of the `List` property, which held a simple array of strings, and a parallel `ItemData` array that held Long numeric values. It was convenient to use the two lists in tandem; for example, you might populate a ListBox with a list of strings from a database table, while simultaneously populating the `ItemData` property with a unique numeric value from that table, such as an AutoNumber. When a user selected an item (or items), you could retrieve the `ItemData` value and use it to obtain the associated object, or use the value as a lookup value for a database query. Table 1 shows the classic VB ListBox data model with three items in the `List` array, and three Long integer values in the `ItemData` array.

TABLE 1: The Classic VB ListBox Data Model

List Array (String values)	ItemData Array (Long values)
Item 1	1293
Item 2	2493
Item 3	8271

In VB.NET, when you drag a ListBox onto a form and then try to write the same loop to populate the ListBox, adding a text value and an `ItemData` numeric value for each item, you'll get a compile-time error. ListBoxes in .NET don't have an `ItemData` property. Hmm. It does seem that the ubiquitous VB ListBox lost some backward compatibility. But in doing so, it also gained functionality. Rather than having two separate arrays limited to Strings and Longs, the .NET ListBox has only one collection, called `Items`, which holds objects—meaning you can store any type of object as an item in a ListBox, and not just simple strings and numbers. However, the ListBox still needs a string to display for each item. That's easy. By default, the ListBox calls the `ToString` method to display each item in the `Items` collection.

But wait! What if the `ToString` method doesn't display what you need? That's easy too. ListBoxes now have a `DisplayMember` property. If the `DisplayMember` property is set, the ListBox invokes the item number named by the `DisplayMember` property before displaying the item.

In other words, rather than storing a single set of strings and associated ID values, and then having to do extra work of retrieving the appropriate data when a user clicks on an item, you can now store the entire set of objects—right in the `Items` property.

Still, despite the best efforts of VB.NET experts to convince them otherwise, people aren't always happy with the current ListBox implementation. One reason is that the consumers of a class aren't always the creators of the class—and they may not be satisfied with the class creator's selections. So first, I'll show you how to re-create the functionality of the classic VB ListBox control, and then I'll show you how to move far beyond it—and even beyond the probable intent of the .NET designers—to create an extremely flexible strategy for displaying items in .NET ListBoxes.

Mimicking a Classic VB ListBox

What you're about to do may feel awkward at first, but you'll soon find that as your thinking patterns switch from managing raw data to handling classes, it will become a natural behavior. Because you're trying to mimic an `ItemData` property that doesn't exist, your first inclination might be to subclass the .NET ListBox control and add your own parallel array of Integer values, accessed via an added `ItemData` property. But that carries baggage you don't need, because you'd have to manage the new array in code—which becomes very difficult with a control that can sort items. You'd then have to make sure the arrays stay synchronized across sorts when users modify the `Item` collection—it can be a mess.

Populating a ListBox

Here's an easier way. Rather than adding the `ItemData` property to the control itself, add the `ItemData` *value* to *the items you put into the* `Items` *collection*. When you do that, you don't have to subclass the control or write any special sorting or list modification code. For example,

suppose you have a list of employee names and ID numbers. When a user clicks on an employee name in the ListBox, you want to show a MessageBox with that user's ID number and name. Assume you have the names in a string array called `names`, and the IDs in a Long array called `IDs`. In classic VB, you would write code like this:

```
Dim i As Long
For i = 0 To UBound(names)
   List1.AddItem names(i)
   List1.ItemData(List1.NewIndex) = ids(i)
Next
```

In .NET, however, you create a simple class with two properties, `Text` and `ItemData`, and a constructor to make it easy to assign the two properties when you create the class. Listing 1 shows the code for such a class, named ListItem.

Listing 1 The ListItem class (ListItem.vb)

```
Public Class ListItem
   Private m_Text As String
   Private m_ItemData As Integer

   Public Sub New(ByVal Text As String, _
      ByVal ItemData As String)
      m_Text = Text
      m_ItemData = ItemData
   End Sub

   Public Property Text() As String
      Get
         Return m_Text
      End Get
      Set(ByVal Value As String)
         m_Text = Value
      End Set
   End Property

   Public Property ItemData() As Integer
      Get
         Return m_ItemData
      End Get
      Set(ByVal Value As Integer)
         m_ItemData = Value
      End Set
   End Property
End Class
```

Assuming you have the `names` and `IDs` arrays already populated, you can create instances of your ListItem class and assign them to the ListBox's `Items` collection using a simple loop:

```
Dim i As Integer

For i = 0 To names.Length - 1
   Me.ListBox1.Items.Add(New ListItem(names(i), ids(i)))
Next
```

But if you run this code, you'll find that the ListBox displays a list of items that look like *[Projectname].ListItem* rather than the list of names you were expecting. That's because, by default, the ListBox calls the `ToString` method for each item to get a displayable string. In this case, however, you don't want to use the default; you want the ListBox to display the `Text` property. So, add this line before the loop that populates the ListBox:

```
Me.ListBox1.DisplayMember = "Text"
```

That tells the ListBox to display the `Text` property for each item rather than the results of `ToString`.

TIP You *must* assign a *property* member to the `ListBox.DisplayMember` property—using a public field or a function doesn't work. That's because the display functionality works through reflection—the ListBox dynamically queries the item at runtime for a *property* with the name you assign to the `ListBox.DisplayMember` property.

Of course, it's your class, and you can eliminate the `DisplayMember` assignment by overriding the `ToString` method to show whatever you like. In this case, you want to show the `Text` property. So, add this code to the ListItem class:

```
Public Overrides Function ToString() As String
   Return Me.Text
End Function
```

Now you can remove the `DisplayMember` assignment and the ListBox will still display the results of the `Text` property.

Getting the Data Back

As you've seen, you can use this simple ListItem class to work with exactly the same data you used in classic VB ListBox code. Getting the data back is just as simple. When a user clicks an item, the .NET ListBox fires a `SelectedItemChanged` event. That happens to be the default event for the ListBox, so if you double-click on it in design mode, Visual Studio will insert a stub event handler for you. Fill in the event-handling code as follows:

```
Private Sub ListBox1_SelectedIndexChanged( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles ListBox1.SelectedIndexChanged
```

```
    Dim li As ListItem
    If Me.ListBox1.SelectedIndex >= 0 Then
       li = DirectCast(Me.ListBox1.SelectedItem, ListItem)
       Debug.WriteLine("Selected Item Text: " & _
          li.Text & System.Environment.NewLine & _
          "Selected ItemData: " & li.ItemData)
    End If
End Sub
```

First, test to ensure that an item is selected. If so, even though *you* know that it's a ListItem, the `ListBox.Items` collection doesn't—it's a collection of objects. Therefore, you need to cast the selected item to the correct type, using either the `CType` or `DirectCast` method (`DirectCast` is faster when you know the cast will succeed).

Now that you've seen a way to re-create VB6 ListBox behavior, I'll concentrate on other ways to use the list controls in .NET, including binding the control to a collection type.

The Class Creator Has Control

Suppose you're told to use a Person class (created by a co-worker) that has four properties: `ID` (Long), `LastName`, `FirstName`, and `Status` (see Listing 2). The Person object has an overloaded constructor so you can assign all the values when you create the object. I've included the complete, finished code for the Person class in Listing 2, even though we're assuming your co-worker didn't give you the class in quite this shape. I've highlighted the portions that you'll add in the next section of this solution. The Person class has `ID`, `LastName`, `FirstName`, and `Status` properties. Although it exposes `LastFirst` and `FirstLast` methods, the interesting parts are the `DisplayPersonDelegate`, the `DisplayMethod` property, and the overridden `ToString` method.

Listing 2 (VB.NET) The Person class (Person.vb)

```
Public Class Person

Public Delegate Function DisplayPersonDelegate _
   (ByVal p As Person) As String
   Private mID As Long
   Private mLastName As String
   Private mFirstName As String
   Private mStatus As String
   Private mDisplayMethod As DisplayPersonDelegate

   Public Sub New(ByVal anID As Long, ByVal lname As String, _
      ByVal fname As String, ByVal statusValue As String)
      mID = anID
      mLastName = lname
      mFirstName = fname
```

```
   mStatus = statusValue
End Sub

Public Property ID() As Long
   Get
      Return mID
   End Get
   Set(ByVal Value As Long)
      mID = Value
   End Set
End Property

Public Property LastName() As String
   Get
      Return mLastName
   End Get
   Set(ByVal Value As String)
      mLastName = Value
   End Set
End Property

Public Property FirstName() As String
   Get
      Return mFirstName
   End Get
   Set(ByVal Value As String)
      mFirstName = Value
   End Set
End Property

Public Property Status() As String
   Get
      Return mStatus
   End Get
   Set(ByVal Value As String)
      mStatus = Value
   End Set
End Property

Public Overloads Overrides Function ToString() As String
   Try
      Return Me.DisplayMethod(Me)
   Catch
      Return MyBase.ToString()
   End Try
End Function

Public Property DisplayMethod() As DisplayPersonDelegate
   Get
      Return mDisplayMethod
   End Get
```

```
        Set(ByVal Value As DisplayPersonDelegate)
            mDisplayMethod = Value
        End Set
    End Property

    Public ReadOnly Property LastFirst() As String
        Get
            Return Me.LastName & ", " & Me.FirstName
        End Get
    End Property

    Public ReadOnly Property FirstLast() As String
        Get
            Return Me.FirstName & " " & Me.LastName
        End Get
    End Property

End Class
```

You want to fill a ListBox with Person objects. So you create a Form and drag a ListBox onto it. You want the ListBox to fill when the user clicks a button, so you add a Fill List button to do that (see Figure 1).

VB.NET makes it easy to display items in a ListBox, because you can set the ListBox's `DataSource` property (binding the list) to any collection that implements the IList interface, which represents a collection of objects that you can access individually by index. Note that you don't have to populate the list through binding; you can still write a loop to add items to the ListBox, as you've already seen in the "Populating a ListBox" section of this solution. However, binding is convenient, as long as you understand exactly what the framework does when it displays the list.

FIGURE 1:
The sample form (form2) initially contains a ListBox and a button.

The ArrayList class implements the IList interface, so you can create an ArrayList member variable for the form, called `people`, and fill it with Person objects during the `Form_Load` event.

```
' define an ArrayList at class level
Private people As New ArrayList()

Private Sub Form2_Load( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles MyBase.Load

   Dim p As Person
   Me.ListBox1.Sorted = True
   ListBox1.DisplayMember = "ToString"
   ListBox1.ValueMember = "ID"
   p = New Person(1, "Twain", "Mark", "")
   people.Add(p)
   p = New Person(2, "Austen", "Jane", "")
   people.Add(p)
   p = New Person(3, "Fowles", "John", "")
   people.Add(p)
End Sub
```

Now, when a user clicks the Fill List button, the ListBox displays items automatically because the code sets the ListBox's `DataSource` property to the `people` ArrayList:

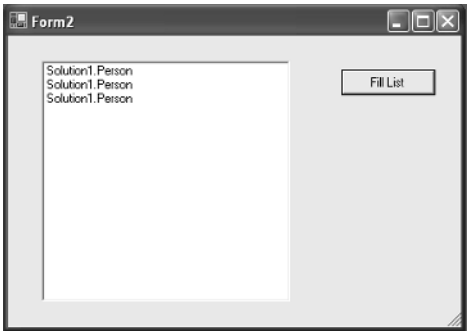
```
Private Sub btnFillList_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) +
   Handles btnFillList.Click

   ListBox1.DataSource = Nothing
   ListBox1.DataSource = people
End Sub
```

Unfortunately, you find that the class creator didn't override the `ToString` implementation or include any additional `LastFirst` method to provide the strings for the ListBox. So the result is that the ListBox calls the default `Person.ToString` implementation, which returns the class name, `Solution1.Person`. The result looks like Figure 2.

OK, no problem. What about using the `DisplayMember` property? Just add the following line to the end of the `Button1_Click` method:

```
ListBox1.DisplayMember = "LastName"
```

FIGURE 2:
The default `Person.ToString` implementation returns only the class name.



Now, run the project again. This time, the result is a little closer to what you want (see Figure 3). Setting the ListBox's `DisplayMember` property to the string `"LastName"` causes the ListBox to invoke the `LastName` method. Unfortunately, this displays only the last names, not the last *and* first names.

FIGURE 3:
Setting the `DisplayMember` property to `"LastName"` displays only the last names.

Now you're stuck. Unless you can get the class creator to add a `LastFirst` property, you'll have to go to a good deal of trouble to get the list to display both names. (At this point, you have to pretend the class creator actually helps and adds a `LastFirst` property to the Person class.)

```
Public ReadOnly Property LastFirst() As String
   Get
      Return Me.LastName & ", " & Me.FirstName
   End Get
End Property
```

Now you can change the `ListBox.DisplayMember` property, and the form will work as expected (see Figure 4):

```
ListBox1.DisplayMember = "LastFirst"
```

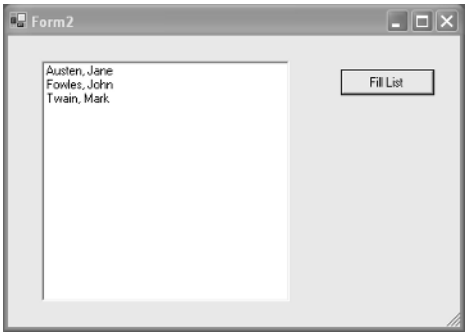


FIGURE 4:
Setting the `ListBox.DisplayMember` property to the `LastFirst` method displays the list in `LastName/FirstName` order.

Just as you get the form working, your manager walks in and says, "Oh, by the way, the clients want to be able to change the list from Last/First to First/Last—both sorted, of course." Now what? You could get the class creator to change the class *again*, but surely there's a better solution.

You *could* inherit the class and add a `FirstLast` method, but then you'd have two classes to maintain. You *could* create a new wrapper class that exposes the `people` ArrayList collection, as well as implements `FirstLast` and `LastFirst` properties. But what if the clients change their minds again? You'd have to keep adding methods to the class, or bite the bullet and beg the class creator for yet more changes. Also, do you really have to create a wrapper for every class you want to display in a ListBox?

This is when you begin to miss the classic VB ListBox's `ItemData` property. If you could assign `Person.ID` as the `ItemData` value, you could concatenate the names yourself, add them to the ListBox, and then look up the Person based on the `ID` when a user selects an item from the ListBox. But `ItemData` is gone. Of course, you can mimic it, as you've seen, but that seems like a lot of trouble when you *already* have a class that you could store directly into the ListBox.

All these possibilities are onerous choices. Things would be a lot easier if you could just control the Person class. What's the answer?

Delegate, Delegate, Delegate

At this point, you need to change roles—take off your reader hat and put on your control creator hat. Here's a completely different approach to displaying custom strings based on some object.

Unless there's a good reason *not* to do so, when you create a class you typically want the class *consumer* to have as much control as possible over the instantiated objects. One way to

increase class consumers' power is to give them control over the method that the ListBox (or other code) calls to get a string representation of your object. In other words, rather than predefining multiple display methods within your class, you provide a public Delegate type, and then add a private member variable and a public property to your class that accept the delegate type. For example:

```
' Public Delegate type definition
Public Delegate Function DisplayPersonDelegate( _
   ByVal p As Person) As String

' Private member variable
Private mDisplayMethod As DisplayPersonDelegate

' Public Property
Public Property DisplayMethod() As DisplayPersonDelegate
    Get
        Return mDisplayMethod
    End Get
    Set (ByVal Value As DisplayPersonDelegate)
        mDisplayMethod=Value
    End Set
End Property
```

The `DisplayPersonDelegate` accepts a Person object and returns a string. The class consumer will create a `DisplayPersonDelegate` object and assign it to the public `DisplayMethod` property.

Next, override the `ToString` method so that it returns the delegate result value. For example:

```
Public Overloads Overrides Function ToString() As String
   Try
      Return Me.DisplayMethod(Me)
   Catch
      Return MyBase.ToString()
   End Try
End Function
```

The advantage of this scheme is that the object consumer gets the best of both worlds—a default `ToString` implementation assignable by the class creator, *and* the ability to call a custom `ToString` method by assigning the delegate. And the class creator doesn't have to worry about all the possible ways that a user may wish to display an object. Finally, it gives the object consumer the ability to set different custom `ToString` methods for *every instance* of the Person class.

The simplest way to use the Person class is to assign a collection of Person objects to some collection, setting the `DisplayMethod` property for each Person to a function matching the

DisplayPersonDelegate signature. For example, to create an ArrayList containing the Person objects, you would first write the display functions:

```
Public Function DisplayPersonFirstLast _
   (byVal p as Person) as String
    Return p.FirstName & " " & p.LastName
End Function

Public Function DisplayPersonLastFirst _
   (byVal p as Person) as String
    Return p.LastName & ", " & p.FirstName
End Function
```

Next, when you create the collection, you assign the DisplayMethod for each Person object:

```
' define an ArrayList at class level
Private people As New ArrayList()

' create Person objects and add them
' to the people ArrayList
Dim p as person
p = New Person(1, "Twain", "Mark", "MT")

' create a DisplayPersonDelegate for the
' DisplayPersonLastFirst method
p.DisplayMethod = New Person.DisplayPersonDelegate _
   (AddressOf DisplayPersonLastFirst)
people.Add(p)

' repeat as necessary
p = New Person(2, "Austen", "Jane", "JA")
p.DisplayMethod = New Person.DisplayPersonDelegate _
   (AddressOf DisplayPersonLastFirst)
people.Add(p)

p = New Person(3, "Fowles", "John", "JF")
p.DisplayMethod = New Person.DisplayPersonDelegate _
   (AddressOf DisplayPersonLastFirst)
people.Add(p)
```

You can see the results by clicking the buttons titled “Last, First” or “First Last” on the sample Form2 form. These buttons switch the display of the names between Last/First and First/Last order without requiring any changes to or using any special display methods in the Person class. Using the DisplayMethod delegate property, Person object consumers can

create custom methods that display the object's data in any format they prefer. But because the scheme defaults to the .NET standard `ToString` method, you haven't changed the base functionality of `ToString` in any other way. In fact, the only reason to override the `ToString` method at all is because that's what the ListBox calls by default. But you could just as easily write a `display` method and have the class consumers call that method explicitly (in this case, by setting the ListBox `DisplayMember` property to `Display`) and leave `ToString` out of the equation altogether. By providing a display method of any kind (`ToString` or otherwise) that accepts a delegate, you have, perhaps accidentally, given class consumers even more power than you may have realized.

Who Needs ItemData?

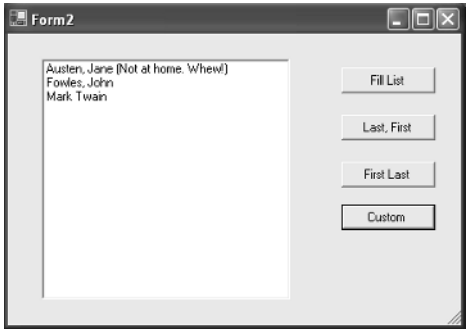
The solution you've just studied accomplishes one other thing that—until now—was impossible without writing customized code, and that's that you can set a different display method for each *instance* of a class. The Custom button illustrates this capability by setting the `Status` property of the "Jane Austen" Person object to a custom string:

```
Private Sub btnCustom_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles btnCustom.Click

   Dim p As Person
   p = CType(people(0), Person) ' Mark Twain
   p.DisplayMethod = New _
      Person.DisplayPersonDelegate( _
       AddressOf DisplayPersonFirstLast)

   p = CType(people(1), Person) ' Jane Austen
   p.Status = "Not at home. Whew!"
   p.DisplayMethod = New  _
      Person.DisplayPersonDelegate( _
      AddressOf DisplayPersonStatus)

   p = CType(people(2), Person) ' John Fowles
   p.DisplayMethod = New _
      Person.DisplayPersonDelegate( _
      AddressOf DisplayPersonLastFirst)
   ListBox1.DataSource = Nothing
   ListBox1.DataSource = people
End Sub
```

```
Public Function DisplayPersonStatus( _
   ByVal p As Person) As String

   Return p.LastName & ", " & p.FirstName & _
      " (" & p.Status & ")"
End Function
```

Now, when you click the button, the results look like Figure 5. In other words, assigning a different `DisplayMethod` delegate to an object instance causes that instance to display differently than other class instances, even within the same ListBox, despite the fact that you don't have to alter the class code to control the text displayed for each item. Figure 5 shows the result when each Person instance has a different display method assigned.

FIGURE 5:
The result of assigning different `DisplayMethod` delegates



While you wouldn't normally want to provide a customized display method for each instance in a ListBox, the capability comes in handy when some people, for example, are comfortable with displaying their nicknames while others aren't, or when the ListBox contains a collection of disparate objects.

Finally, giving class consumers the ability to create customized display strings for your classes goes a long way toward making the missing `ItemData` truly unnecessary. When you click on an item in the ListBox, it displays a MessageBox that shows the selected item and its `ID`, proving yet again that associating an `ID` with an item by using objects works just as well as the older `ItemData` array—and doesn't require the class consumer to write any code.

There's one small downside to this method. If you want to post two ListBoxes side by side, both containing the same objects but with one displaying (for example) LastName/FirstName and the other displaying FirstName/LastName, you need to implement a `Clone` method. Doing so lets you set different display methods for the objects in each list. In this particular case, using a wrapper object (such as the ListItem class) to handle the class display may be a simpler design.

SOLUTION 2

Create Owner-Drawn ListBoxes and ComboBoxes

PROBLEM I want to create ListBoxes and ComboBoxes that can contain icons and special fonts like the ones I see in other Windows applications. How can I do that with .NET?

SOLUTION Learn to use the `DrawMode` settings with ListBoxes and ComboBoxes to create and display customized items.

You must create an "owner-drawn" ListBox or ComboBox when you want to bypass the controls' automatic item display to do something special, such as display an image for each item or display a list in which the items aren't all the same size. The .NET Framework makes it simple to generate these custom item lists. In this solution, you'll learn how to populate list and ComboBox controls with items you draw yourself.

The only thing you need to do to create an owner-drawn ListBox or ComboBox is to set the `DrawMode` property to either `OwnerDrawFixed` or `OwnerDrawVariable`. The `DrawMode` property has three possible settings:

- `Normal`, in which the system handles displaying the items automatically
- `OwnerDrawFixed`, which you should use when you want to draw the items yourself and all the items are the same height and width
- `OwnerDrawVariable`, which you use to draw items that vary in height or width

The default setting is, of course, `Normal`. When you select the `OwnerDrawFixed` setting, you must implement a `DrawItem` method. The ListBox calls your `DrawItem` whenever it needs to draw an item.

When you select the `OwnerDrawVariable` setting, you must implement both the `DrawItem` and a `MeasureItem` method. The `MeasureItem` method lets you set the size of the item to be drawn.

When you use the `Normal` setting, the system does not fire either the `MeasureItem` or the `DrawItem` method.

NOTE There are some restrictions when you use any setting but `Normal`. You can't create variable-height items for multicolumn ListBoxes, and CheckedListBoxes don't support either of the owner-drawn `DrawMode` settings.

Listing Files and Folders

Suppose you want to list the files and directories in a folder along with the associated system icons appropriate to the type of file. You must follow several steps to accomplish this task:

- Validate the requested directory path.
- Retrieve the files and subfolders from a directory.
- Iterate through them, retrieving their types and names.
- Find the appropriate icon for each file type.
- Draw the items for the ListBox containing the appropriate icon and text.

Figure 1 depicts a Web form with the finished ListBox control that displays all the files in a specified directory along with their corresponding system icons. To use the example, enter a directory path in the first text field. The form ensures that the entered path is valid, and then follows the steps listed here to fill a ListBox shown in a separate dialog box. The user can double-click an item in the list, or select an item and click OK. The constructor for the dialog form (ListFilesAndFolders) requires a path string.

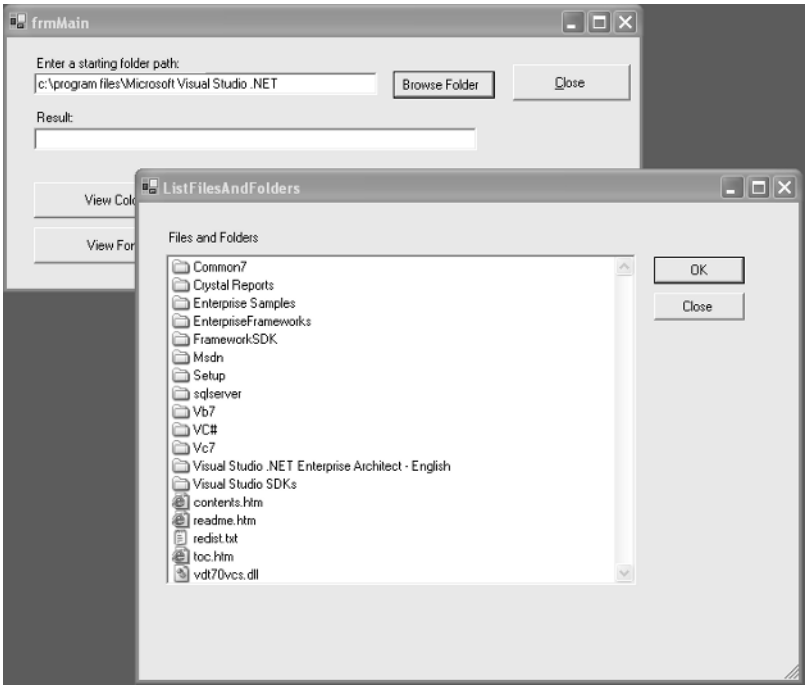


FIGURE 1: The ListFilesAndFolders form contains an owner-drawn ListBox that displays names of folders and files, along with their system-associated icons.

The first step in creating the application logic is to validate the path string users enter in the main form. The easiest way to do that is to use the System.IO.DirectoryInfo class, which has an `Exists` method that returns `True` if the directory exists:

```
Dim di As DirectoryInfo
Me.txtResult.Text = Nothing
di = New DirectoryInfo(Me.txtPath.Text)
If Not di.Exists Then
   txtPath.ForeColor = System.Drawing.Color.Red
   Beep()
   Exit Sub
End If
```

The code turns the TextBox text red and plays a warning sound if the entered path is invalid; otherwise, it creates a new instance of the ListFilesAndFolders form, passing the validated path string to its constructor:

```
Dim frmFiles As New _
   ListFilesAndFolders(Me.txtPath.Text)
```

The ListFilesAndFolders form contains a ListBox, an OK button, and a Close button. The form's constructor calls a `FillList` method that retrieves the files and folders in the specified path and then fills a ListBox control with the icons and names, suspending the control's display until the method completes:

```
Sub FillList(ByVal aPath As String)
   Dim fsi As FileSystemInfo
   lstFiles.BeginUpdate()
   Me.lstFiles.ItemHeight = _
      CInt(lstFiles.Font.GetHeight + 4)
   lstFiles.Items.Clear()
   files = New DirectoryInfo(aPath).GetFileSystemInfos
   For Each fsi In files
      lstFiles.Items.Add(fsi)
   Next
   lstFiles.EndUpdate()
End Sub
```

The `DirectoryInfo.GetFileSystemInfos` method used in this code snippet returns an array of FileSystemInfo objects. The code iterates through the returned array and adds each item to the ListBox's `Items` collection.

Here's where things get interesting. The ListBox's `DrawMode` property is set to `OwnerDrawFixed`, because although you want to draw the items yourself (so you can add the file-type icons), each item will be the same height. When you set `DrawMode` to anything except `Normal`, the act of adding the items to the ListBox doesn't cause the ListBox to draw them; instead, the ListBox fires a `DrawItem` event whenever the ListBox needs to display an item. In this case, every time the `DrawItem` event fires, you want to draw an icon and the name of a FileSystemInfo

object that represents a file or folder. Because this is an owner-drawn control, you must create the `DrawItem` method to display the item:

```
Private Sub lstFiles_DrawItem( _
   ByVal sender As Object, _
   ByVal e As System.Windows.Forms.DrawItemEventArgs) _
   Handles lstFiles.DrawItem

   ' the system sometimes calls this method with
   ' an index of -1. If that happens, exit.
   If e.Index < 0 Then
      e.DrawBackground()
      e.DrawFocusRectangle()
      Exit Sub
   End If

   ' create a brush
   Dim aBrush As Brush = System.Drawing.Brushes.Black

   ' get a reference to the item to be drawn
   Dim fsi As FileSystemInfo = _
      CType(lstFiles.Items(e.Index), FileSystemInfo)

   ' create an icon object
   Dim anIcon As Icon

   ' use a generic string format to draw the filename
   Dim sFormat As StringFormat = _
      StringFormat.GenericTypographic

   ' get the height of each item
   Dim itemHeight As Integer = lstFiles.ItemHeight

   ' call these methods to get items to highlight
   ' properly
   e.DrawBackground()
   e.DrawFocusRectangle()


   ' retrieve the appropriate icon for this file type
   anIcon = IconExtractor.GetSmallIcon(fsi)

   ' draw the icon
   If Not anIcon Is Nothing Then
      e.Graphics.DrawIcon(anIcon, 3, _
      e.Bounds.Top + ((itemHeight - _
      anIcon.Height) \ 2))
```

```
            anIcon.Dispose()
        End If

        ' if the item is selected,
        ' change the text color to white
        If (e.State And _
            Windows.Forms.DrawItemState.Selected) = _
            Windows.Forms.DrawItemState.Selected Then
                aBrush = System.Drawing.Brushes.White
        End If
        sFormat.LineAlignment = StringAlignment.Center
        e.Graphics.DrawString(fsi.Name, lstFiles.Font, _
            aBrush, 22, e.Bounds.Top + _
            (e.Bounds.Height \ 2), sFormat)
    End Sub
```

In the `DrawItem` method shown here, the code calls a shared `GetSmallIcon` method exposed by the IconExtractor class (see Listing 1), which, when passed a FileSystemInfo object, calls the Win32 `SHGetFileInfo` API to extract the icon for the file type represented by that object. The IconExtractor class exposes two public shared methods—`GetLargeIcon` and `GetSmall-Icon`—both of which simply call a private `GetIcon` method that returns the large (32×32) or small (16×16) icon versions, respectively:

```
    Public Shared Function GetSmallIcon( _
        ByVal fsi As FileSystemInfo) As Icon

        Return IconExtractor.GetIcon _
            (fsi, SHGFI_SMALLICON)
    End Function

    Public Shared Function GetLargeIcon( _
        ByVal fsi As FileSystemInfo) As Icon

        Return IconExtractor.GetIcon _
            (fsi, SHGFI_LARGEICON)
    End Function

    Private Shared Function GetIcon( _
        ByVal fsi As FileSystemInfo, _
        ByVal anIconSize As Integer) As Icon

        Dim aSHFileInfo As New SHFILEINFO()
        Dim cbFileInfo As Integer = _
            Marshal.SizeOf(aSHFileInfo)
        Dim uflags As Integer = SHGFI_ICON Or _
            SHGFI_USEFILEATTRIBUTES Or anIconSize
        Try
```

```
      SHGetFileInfo(fsi.FullName, fsi.Attributes, _
         aSHFileInfo, cbFileInfo, uflags)
      Return Icon.FromHandle(aSHFileInfo.hIcon)
   Catch ex As Exception
      Return Nothing
   End Try
End Function
```


Listing 1 **The IconExtractor Class calls the Win32 API to identify and return icons appropriate for a specific file type. (IconExtractor.vb)**

```
Imports System
Imports System.Drawing
Imports System.Runtime.InteropServices
Imports System.Windows.Forms
Imports System.IO

Public Class IconExtractor

   Private Const SHGFI_SMALLICON = &H1
   Private Const SHGFI_LARGEICON = &H0
   Private Const SHGFI_ICON = &H100
   Private Const SHGFI_USEFILEATTRIBUTES = &H10

   Public Enum IconSize
      SmallIcon = SHGFI_SMALLICON
      LargeIcon = SHGFI_LARGEICON
   End Enum

   <StructLayout(LayoutKind.Sequential)> _
   Private Structure SHFILEINFO
      ' pointer to icon handle
      Public hIcon As IntPtr
      ' icon index
      Public iIcon As Integer
      ' not used in this example
      Public dwAttributes As Integer
      ' file pathname--marshal this as
      ' an unmanaged LPSTR of MAX_SIZE
     <MarshalAs(UnmanagedType.LPStr, SizeConst:=260)> _
      Public szDisplayName As String
      ' file type--marshal as unmanaged
      ' LPSTR of 80 chars
      <MarshalAs(UnmanagedType.LPStr, SizeConst:=80)> _
      Public szTypeName As String
   End Structure

   Private Declare Auto Function SHGetFileInfo _
   Lib "shell32" (ByVal pszPath As String, _
   ByVal dwFileAttributes As Integer, _
```

```
      ByRef psfi As SHFILEINFO, _
      ByVal cbFileInfo As Integer, _
      ByVal uFlags As Integer) As Integer

      Public Shared Function GetSmallIcon( _
         ByVal fsi As FileSystemInfo) As Icon
         Return IconExtractor.GetIcon _
            (fsi, SHGFI_SMALLICON)
      End Function

      Public Shared Function GetLargeIcon( _
         ByVal fsi As FileSystemInfo) As Icon
         Return IconExtractor.GetIcon _
            (fsi, SHGFI_LARGEICON)
      End Function

      Private Shared Function GetIcon( _
         ByVal fsi As FileSystemInfo, _
         ByVal anIconSize As Integer) As Icon

         Dim aSHFileInfo As New SHFILEINFO()
         Dim cbFileInfo As Integer = _
            Marshal.SizeOf(aSHFileInfo)
         Dim uflags As Integer = SHGFI_ICON Or _
            SHGFI_USEFILEATTRIBUTES Or anIconSize
         Try
            SHGetFileInfo(fsi.FullName, _
            fsi.Attributes, aSHFileInfo, _
            cbFileInfo, uflags)
            Return Icon.FromHandle(aSHFileInfo.hIcon)
         Catch ex As Exception
            Return Nothing
         End Try
      End Function

   End Class
```

The `GetSmallIcon` and `GetLargeIcon` methods both accept a FileSystemInfo object. Internally, the `GetIcon` method uses the FileSystemInfo object to pass the filename and file attributes to the `SHGetFileInfo` API call. After drawing the icon, the `DrawItem` event handler calls the `Graphics.DrawString` method to place the filename on the image next to the icon. The ListBox calls the `DrawItem` method repeatedly, once for each item in its `Items` collection.

The `DrawItemEventArgs` argument to the `DrawItem` event handler exposes an `Index` property whose value is the index of the item to be drawn. Watch out! The system raises the `DrawItem` event with an index value of -1 when the `Items` collection is empty. When that happens, you should call the `DrawItemEventArgs.DrawBackground()` and `DrawFocusRectangle()` methods and then exit. The purpose of raising the event is to let the control draw a focus rectangle so

that users can tell it has the focus, even when no items are present. The code traps for that condition, calls the two methods, and then exits the handler immediately.

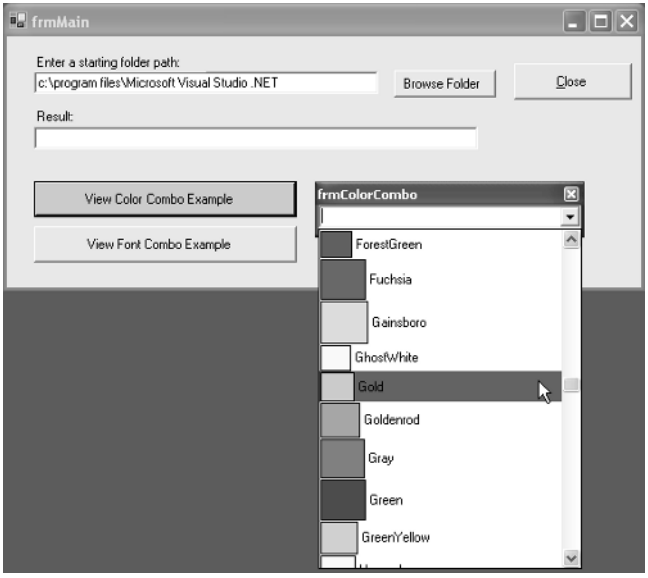
Users can select an item and close the ListFilesAndFolders form either by selecting an item and then clicking the OK button, or by double-clicking an item. Either way, the form sets a public property called `SelectedItem`, sets another public property called `Cancel`, and then closes. The main form then displays the filename of the selected item in the Result field.

Drawing Items with Variable Widths and Heights

This section presents a similar example, but this time the items you'll create won't all be the same width and height. To create an owner-drawn ListBox or ComboBox with items of variable heights and widths, set the `DrawMode` property to `OwnerDrawVariable`. Then, implement a method that handles the `MeasureItem` event, which accepts a `sender` (Object) and a `System.Windows.Forms.MeasureItemEventArgs` argument. The sample form `frmColorCombo` displays all the known system colors and their names in a ComboBox. The items themselves vary between 20 and 40 pixels in height. The result is contrived and ugly (see Figure 2) but serves to illustrate the point.

FIGURE 2:

The frmColorCombo form contains an owner-drawn ListBox that displays all the known color names, accompanied by a variable-height color swatch.



The code does present a couple of interesting problems. The .NET Framework defines the common colors as an *enumeration*. Enumerations expose a `getNames` method that, when passed a Type object for a particular enumeration, returns an array of names in that enumeration. In this case, you want not only the names but the colors themselves. You can create a

Color object if you know the name by using the `Color.FromName` method. So the following `For...Each` loop retrieves the known color names, and then adds Color objects to the ComboBox's `Items` collection:

```
Dim aColorName As String
For Each aColorName In _
   System.Enum.GetNames _
   (GetType(System.Drawing.KnownColor))
   colorCombo.Items.Add(Color.FromName(aColorName))
Next
```

The frmColorCombo class defines a private Random object (`mRand`). Because its `DrawMode` property is set to `OwnerDrawVariable`, the ComboBox control calls the `MeasureItem` event before drawing each item (in other words, before calling the `DrawItem` method):

```
Protected Sub colorCombo_MeasureItem( _
   ByVal sender As Object, ByVal e As _
   System.Windows.Forms.MeasureItemEventArgs) _
   Handles colorCombo.MeasureItem

   e.ItemHeight = mRand.Next(20, 40)

End Sub
```

In this code snippet, the `comboColor_MeasureItem` event handler calls the overloaded `Random.Next` method to get the next random number between 20 and 40, and assigns that to the `ItemHeight` property of the `MeasureItemEventArgs` parameter.

The `DrawItem` event handler used here is similar to the one in the previous example. It retrieves the Color object from the `Items` collection as specified by the `Index` value of the `DrawItemEventArgs` parameter, and then retrieves the color name from that Color object. The method draws a square and fills it with the appropriate color, and then draws the color name to the right of the square. As you can see in Listing 2, the `DrawItem` method uses the bounds set randomly in the `MeasureItem` method for each item.

Listing 2 **The colorCombo_DrawItem method displays a random-height color swatch and the color name for each color shown.**

```
Protected Sub colorCombo_DrawItem( _
   ByVal sender As Object, _
   ByVal e As System.Windows.Forms.DrawItemEventArgs) _
   Handles colorCombo.DrawItem

   If e.Index < 0 Then
      e.DrawBackground()
      e.DrawFocusRectangle()
      Exit Sub
```

```
    End If
    ' Get the Color object from the Items list
    Dim aColor As Color = _
       CType(colorCombo.Items(e.Index), Color)

    ' get a square using the bounds height
    Dim rect As Rectangle = New Rectangle _
       (2, e.Bounds.Top + 2, e.Bounds.Height, _
       e.Bounds.Height - 4)

    Dim br As Brush

    ' call these methods first
    e.DrawBackground()
    e.DrawFocusRectangle()

    ' change brush color if item is selected
    If e.State = _
       Windows.Forms.DrawItemState.Selected Then
       br = Brushes.White
    Else
       br = Brushes.Black
    End If

    ' draw a rectangle and fill it
    e.Graphics.DrawRectangle(New Pen(aColor), rect)
    e.Graphics.FillRectangle(New SolidBrush _
       (aColor), rect)

    ' draw a border
    rect.Inflate(1, 1)
    e.Graphics.DrawRectangle(Pens.Black, rect)

    ' draw the Color name
    e.Graphics.DrawString(aColor.Name, _
       colorCombo.Font, br, e.Bounds.Height + 5, _
       ((e.Bounds.Height - colorCombo.Font.Height) _
        \ 2) + e.Bounds.Top)
End Sub
```

In the sample form, when you select a color from the `frmColorCombo` window, the main form changes the button color to reflect your choice. If you close the `frmColorCombo` window without selecting a color, the main form changes the button back to its default color.

Building a Font Combo That Displays Fonts

Here's another useful example. It's relatively easy to create a ComboBox that lets a user select a font, but in Microsoft Word and other commercial applications, you sometimes see font selection ComboBoxes that display the names of the fonts using the fonts themselves, rather

than using a single fixed typeface. To do that, you first need to retrieve the list of font families installed on the machine—a process called *enumerating* fonts—and add them to an owner-drawn ComboBox. Then, in the `MeasureItem` event, you create an instance of that font and use it to measure the font name. Similarly, in the `DrawItem` event, you create an instance of the font and use that to draw the font name. Because most of the code is identical to that of the `frmColorCombo` form, I'll only show the relevant portion in Listing 3, although the accompanying code (available on the Sybex Web site, `www.sybex.com`) contains the complete implementation.

The system maintains a list of the installed font families, which you can retrieve either by using the `FontFamilies.Families` property or by creating a new InstalledFontsCollection object and calling its `Families` method:

```
Dim installedFonts As New InstalledFontCollection()
```

In that way, you can retrieve a complete list of installed fonts—including the line-drawing and non-character WingDings fonts. Some of those don't look very good in a ComboBox, so I've eliminated the worst offenders by testing the height of the letter A at a font size of 9 points. If the font height measurement is greater than 20 pixels, the code in Listing 3 doesn't add it to the ComboBox.

Listing 3 **The Form_Load event handler fills the ComboBox's Items collection with a list of fonts. (frmFontCombo.vb)**

```
Private Sub frmFontCombo_Load( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

   Me.Size = New Size(New Point(240, 60))
   Me.ControlBox = True
   Me.FormBorderStyle = _
      Windows.Forms.FormBorderStyle.FixedToolWindow
   fontCombo = _
      New System.Windows.Forms.ComboBox()
   fontCombo.DrawMode = _
      Windows.Forms.DrawMode.OwnerDrawVariable
   fontCombo.Location = New Point(0, 0)
   fontCombo.Width = Me.Width - 5
   fontCombo.MaxDropDownItems = 20
   fontCombo.IntegralHeight = True
   Dim aFontFamily As FontFamily
   Dim installedFonts As New InstalledFontCollection()
   Dim g As Graphics = Me.CreateGraphics
   Dim families() As FontFamily = FontFamily.GetFamilies(g)
   For Each aFontFamily In families 'installedFonts.Families
```

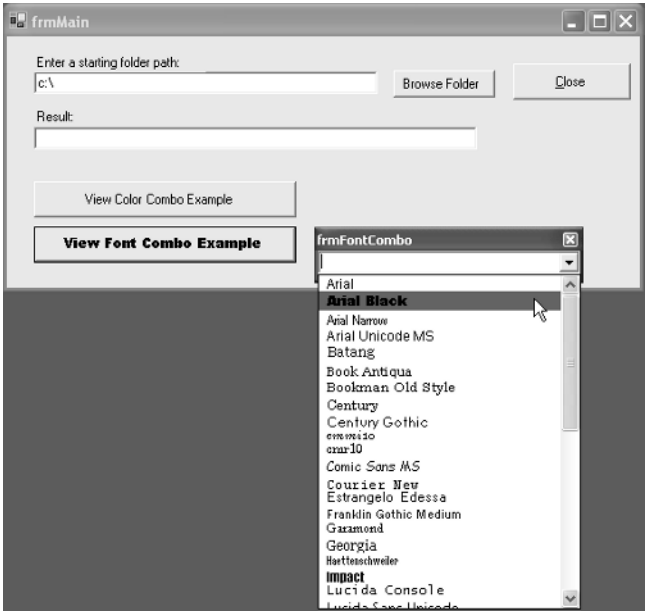
```
            If aFontFamily.IsStyleAvailable _
               (FontStyle.Regular) Then
               If g.MeasureString("A", New Font(aFontFamily, 9, _
                  FontStyle.Regular, GraphicsUnit.Point)).Height _
                  < 20 Then
                  fontCombo.Items.Add(aFontFamily)
               End If
            End If
         Next
         g.Dispose()
         Me.Controls.Add(fontCombo)

      End Sub
```

Similar to the previous example, after you select a font from the ComboBox, the sample changes the View Font Combo Example button font to match your selection (see Figure 3). Closing the window without selecting a font switches the button text back to the default font.

By selecting the appropriate `DrawMode` setting for your ListBoxes and ComboBoxes and implementing the `MeasureItem` and `DrawItem` event handlers, you gain complete control over the contents of ListBoxes and ComboBoxes within the .NET Framework. You can extend the owner-drawn techniques shown here to create complex interactive ComboBoxes.

FIGURE 3:

Selecting a font from the ComboBox changes the View Font Combo button's font to match the selection.



SOLUTION 3

Upgrade Your INI Files to XML

PROBLEM: The .NET Framework wraps many underlying Windows API calls, but doesn't provide an easy way to get to legacy data and configuration settings stored in application initialization (INI) files.

SOLUTION: Migrate your legacy INI data to XML using the Windows API and this XML INI file wrapper class.

The INI (application Initialization) file format became popular because it provides a convenient way to store values that might change (such as file locations and user preferences) in a standard format accessible to but outside of compiled application code. INI files are text-based—meaning you can read and change most values manually if necessary—and logically arranged—meaning it's easy even for nontechnical personnel to understand the contents. In addition, the functions for reading and modifying the files are built into Windows.

You can still use the existing Win32 API calls to read and write from standard INI files using the DllImport attribute with C#, or with the Declare Function statement in VB.NET; however, there are a couple of tricks.

WARNING The API calls to interact with INI files have been obsolete since the release of Windows 95, and are supported in Win32 for backward compatibility only. The INIWrapper class shown here wraps the most important API calls for interacting with INI files.

The API INI Functions

The API functions that deal with INI files are usually paired; there's a Get and a Write version for most of the functions. The API contains special functions to read and write the `win.ini` file in the Windows folder (which aren't discussed in this solution); however, if you need to modify win.ini through .NET, you'll see enough here to declare the function prototypes yourself. For INI files associated with individual applications, the most important Win32 API INI-related functions are:

GetPrivateProfileString Retrieves an individual value associated with a named section and key.

WritePrivateProfileString Sets an individual value associated with a named section and key.

GetPrivateProfileInt Retrieves an integer value associated with a named section and key.

WritePrivateProfileInt Sets an integer value associated with a named section and key.

GetPrivateProfileSection Retrieves all the keys and values associated with a named section.

WritePrivateProfileSection Sets all the keys and values associated with a named section.

GetPrivateProfileSectionNames Retrieves all the section names in an INI file.

For example, the `GetPrivateProfileString` API function retrieves an individual value from an INI file. You specify the file, the section, the key, a default value, a string buffer for the returned information, and the size of the buffer. In classic VB, you use a Declare Function statement to declare the API function:

```
' Classic VB declaration
Public Declare Function _
   GetPrivateProfileString _
   Lib "kernel32" _
   Alias "GetPrivateProfileStringA" _
   (ByVal lpApplicationName As String, _
   ByVal lpKeyName As Any, _
   ByVal lpDefault As String, _
   ByVal lpReturnedString As String, _
   ByVal nSize As Long, _
   ByVal lpFileName As String) As Long
```

In .NET the equivalent declaration is

```
' VB.NET declaration
Private Declare Ansi Function _
   GetPrivateProfileString _
   Lib "KERNEL32.DLL"
   Alias "GetPrivateProfileStringA" _
   (ByVal lpAppName As String, _
   ByVal lpKeyName As String, _
   ByVal lpDefault As String, _
   ByVal lpReturnedString As StringBuilder, _
   ByVal nSize As Integer, _
   ByVal lpFileName As String) As Integer
```

C# handles things a little differently. In C#, you use the DllImport attribute to declare function prototypes, so the equivalent declaration is

```
// C# function prototype
[ DllImport("KERNEL32.DLL",
   EntryPoint="GetPrivateProfileString")]
   protected internal static extern int
GetPrivateProfileString(string lpAppName,
   string lpKeyName, string lpDefault,
   StringBuilder lpReturnedString, int nSize,
   string lpFileName);
```

The interesting point here is that the `lpReturnedString` parameter expects a string buffer `nSize` in length. In .NET, you pass a StringBuilder object for the `lpReturnedString` parameter rather than a String object (remember to add a `using System.Text;` line to your class file in C#; in VB.NET use the `Imports System.Text` statement). That's because strings in .NET are immutable, so while you can pass them into unmanaged code without errors, any changes made to the string buffer in unmanaged code aren't visible in your .NET code when the .NET Framework marshals the data back into managed code. Fortunately, StringBuilder objects act as mutable strings and you can create them with a fixed buffer size. When calling unmanaged code that needs a fixed-size string buffer, try a StringBuilder first.

The `EntryPoint` parameter in the C# declaration in the previous snippet isn't strictly required. The `EntryPoint` parameter contains the name (or the index) of the function you want to declare. You need to include this parameter only if the name of your .NET function isn't the same, because .NET looks for a function named identically to the .NET function if you don't include the parameter. However, if you were to rename the .NET function to `getAppInitValue`, you would have to include the `EntryPoint` parameter:

```
[ DllImport("KERNEL32.DLL",
   EntryPoint="GetPrivateProfileString")]
   protected internal static extern int
getAppInitValue(string lpAppName,
   string lpKeyName, string lpDefault,
   StringBuilder lpReturnedString, int nSize,
   string lpFileName);
```

NOTE I declared the prototypes in the C# version of this class using the protected internal static accessibility level, which means they're only visible from this project and any derived classes. You may want to change the accessibility level, depending on how you want to use the functions.

Suppose you had a simple INI file that looks like this:

```
[textvalues]
1=item1
2=item2
3=item3

[intvalues]
1=101
2=102
3=103
```

After setting up the imported `GetPrivateProfileString` function definition, you can call it just like any other function. For example, using the INI file shown earlier, and assuming it was saved as `c:\INIinterop.ini`, the following code would retrieve the value of the item in

the [textvalues] section with the key "1"—the string "item1"—and write it to the output window. Note that you don't have to instantiate an instance of the INIWrapper class, because all the methods are class-level methods (static methods in C# and shared methods in VB.NET).

```
Dim buffer As StringBuilder = New StringBuilder(256)
Dim sDefault As String = ""
Dim bufLen As Integer = _
  INIWrapper.GetPrivateProfileString _
  ("textvalues", "1", "", buffer, buffer.Capacity, _
  "c:\INIinterop.ini") <> 0)
Debug.WriteLine(buffer.ToString())
```

In contrast, you can write a new value without using a StringBuilder, because you don't need a return value:

```
INIFileInterop.WritePrivateProfileString
   ("textvalues", "1", "new Item 1",
   "c:\INIinterop.ini")
```

You can retrieve and write integer values with the GetPrivateProfileInt and WritePrivate-ProfileInt methods. (See Listing 1 later in this solution for the full declarations.) To call the GetPrivateProfileInt function, pass the section name, key name, a default integer value (which is returned if the key doesn't exist), and the name of the INI file. For example, the following code writes "101" to the output window:

```
int result = INIWrapper.GetPrivateProfileString
   ("intvalues", "1", 0, "c:\INIinterop.ini");
Debug.WriteLine(result.ToString());

dim result as Integer = _
INIWrapper.GetPrivateProfileString _
   ("intvalues", "1", 0, "c:\INIinterop.ini")
Debug.WriteLine(result.ToString())
```

Unfortunately calling the GetPrivateProfileSection function isn't quite as easy. The function returns a buffer filled with a null-delimited list of all the keys and values (items) in a specified section, with an additional trailing null character after the last item, so the returned buffer looks like this, where the \0 characters denote nulls:

```
1=item1\02=item2\03=item3\0\0
```

You would expect to declare the function using a StringBuilder object with a predefined length for the lpReturnedString buffer parameter, just as with the GetPrivateProfileString function—but that doesn't work. When you call the function, it returns the proper number of characters, but the StringBuilder contains only the first item, "1=item1". However, the return value of the function contains 24, which is correct—the length of the text of the three

items in the [textvalues] section plus one null character after each item. In other words, the StringBuilder buffer *contains* the second and third items—but you can't reach them; the first null character in the StringBuilder buffer determines the length of the contents available, and the StringBuilder throws an error if you attempt to index a character past that point. Obviously, you need to pass a managed type that isn't quite so sensitive to null-delimited strings.

Using a Char array doesn't work either—the function doesn't alter the array, even though it still returns the correct number of characters. Instead, after much fiddling with the problem, you'll find that you can use a byte array. You can see the full declaration in Listing 1.

When the function call returns, the byte array contains the entire set of items, separated with null characters, as expected. I haven't found a truly simple way to convert the byte array to a set of strings; the best method I've found is to iterate through the byte array creating the individual strings using a StringBuilder object. The sample GetINISection method here wraps the call to the GetPrivateProfileSection API, converts the returned items to strings, collects them in a StringCollection, and returns that to the calling code:

```
' ********************************************
' * VB.NET code                              *
' ********************************************
Public Shared Function GetINISection(ByVal filename _
   As String, ByVal section As String) _
   As StringCollection
   Dim items As StringCollection = New _
      StringCollection()
   Dim buffer(32768) As Byte
   Dim bufLen As Integer = 0
   Dim sb As StringBuilder
   Dim i As Integer
   bufLen = GetPrivateProfileSection(section, _
      buffer, buffer.GetUpperBound(0), filename)
   If bufLen > 0 Then
      sb = New StringBuilder()
      For i = 0 To bufLen - 1
         If buffer(i) <> 0 Then
            sb.Append(ChrW(buffer(i)))
         Else
            If sb.Length > 0 Then
               items.Add(sb.ToString())
               sb = New StringBuilder()
            End If
         End If
      Next
   End If
```

```
   Return items
End Function


// ********************************************
// * C# code                                  *
// ********************************************
public static StringCollection GetINISection
   (String filename, String section) {
   StringCollection items = new StringCollection();
   byte[] buffer = new byte[32768];
   int bufLen=0;
   bufLen = GetPrivateProfileSection(section,
      buffer, buffer.GetUpperBound(0), filename);
   if (bufLen > 0) {
      StringBuilder sb = new StringBuilder();
      for(int i=0; i < bufLen; i++) {
         if (buffer[i] != 0) {
            sb.Append((char) buffer[i]);
         }
         else {
            if (sb.Length > 0) {
               items.Add(sb.ToString());
               sb = new StringBuilder();
            }
         }
      }
   }
   return items;
}
```

To use the method, add the line `using System.Collections.Specialized;` (in C#) or `Imports System.Collections.Specialized` (in VB.NET) to the top of the calling class, and then you can write code such as this:

```
' ********************************************
' * VB.NET code                              *
' ********************************************
Dim s As String
Dim items as StringCollection = _
   INIFileInterop.GetINISection _
   ("c:\INIinterop.ini", "textvalues")
For Each s In items
   Debug.WriteLine(s)
Next

// ********************************************
```

```
// * C# code                                          *
// ***************************************************
StringCollection items =
   INIFileInterop.GetINISection
   ("c:\INIinterop.ini", "textvalues");
foreach(String s in items) {
   Debug.WriteLine(s);
}
```

The sample code in Listing 1 (downloadable from the Sybex Web site) includes an `INIWrapper` class (`INIWrapper.cs` in C#, `INIWrapper.vb` in VB.NET) that contains the API function prototypes and some wrapper methods to simplify calling the APIs. The classes work with any standard INI file.

There are a few other Win32 API calls that work with INI files, and over the years, I've found it useful to add wrapper functions that, for example, return just a list of keys in a section, or just the list of values from a section. I've also found it useful to write wrapper functions to insert comments at various places. You can probably think of many more extensions to these simple classes.

Finally, the code in Listing 1 is meant for example use only. You should add error trapping and checking. See the `DllImportAttribute.SetLastError` field and the `Marshal.GetLastWin32Error` method in the .NET Framework documentation for more information.

Listing 1 The VB.NET INIWrapper class (INIWrapper.vb)

```
Option Strict On
Imports System
Imports System.Runtime.InteropServices
Imports System.Collections.Specialized
Imports System.Text
Imports System.IO
Public Class INIWrapper
   Private Declare Ansi Function GetPrivateProfileString _
      Lib "KERNEL32.DLL" Alias "GetPrivateProfileStringA" _
      (ByVal lpAppName As String, ByVal lpKeyName As _
      String, ByVal lpDefault As String, _
      ByVal lpReturnedString As StringBuilder, _
      ByVal nSize As Integer, _
      ByVal lpFileName As String) As Integer
   Private Declare Ansi Function GetPrivateProfileInt _
      Lib "KERNEL32.DLL" (ByVal lpAppName As String, _
      ByVal lpKeyName As String, ByVal iDefault As _
      Integer, ByVal lpFileName As String) As Integer
   Private Declare Ansi Function _
      WritePrivateProfileString Lib "KERNEL32.DLL" _
```

```
   Alias "WritePrivateProfileStringA" _
   (ByVal lpAppName As String, ByVal lpKeyName As _
   String, ByVal lpString As String, ByVal lpFileName _
   As String) As Boolean
Private Declare Ansi Function GetPrivateProfileSection _
   Lib "KERNEL32.DLL" Alias _
   "GetPrivateProfileSectionA" _
   (ByVal lpAppName As String, ByVal lpReturnedString _
   As Byte(), ByVal nSize As Integer, ByVal lpFileName _
   As String) As Integer
Private Declare Ansi Function _
   WritePrivateProfileSection _
   Lib "KERNEL32.DLL" Alias _
   "WritePrivateProfileSectionA" _
   (ByVal lpAppName As String, ByVal data As Byte(), _
   ByVal lpFileName As String) As Boolean
Private Declare Ansi Function _
   GetPrivateProfileSectionNames _
   Lib "KERNEL32.DLL" Alias _
   "GetPrivateProfileSectionNamesA" _
   (ByVal lpReturnedString As Byte(), ByVal nSize As _
   Integer, ByVal lpFileName As String) As Integer

Public Shared Function GetINIValue _
   (ByVal filename As String, ByVal section As String, _
   ByVal key As String) As String
   Dim buffer As StringBuilder = New StringBuilder(256)
   Dim sDefault As String = ""
   If (GetPrivateProfileString(section, key, sDefault, _
      buffer, buffer.Capacity, filename) <> 0) Then
      Return buffer.ToString()
   Else
      Return Nothing
   End If
End Function

Public Shared Function WriteINIValue _
   (ByVal filename As String, ByVal section As String, _
   ByVal key As String, ByVal sValue As String) _
   As Boolean
   Return WritePrivateProfileString(section, key, _
      sValue, filename)
End Function

Public Shared Function GetINIInt _
   (ByVal filename As String, ByVal section As String, _
   ByVal key As String) As Integer
   Dim iDefault As Integer = -1
   Return GetPrivateProfileInt(section, key, iDefault, _
      filename)
End Function
```

```
Public Shared Function GetINISection _
   (ByVal filename As String, ByVal section As String) _
   As StringCollection
   Dim items As StringCollection = New _
      StringCollection()
   Dim buffer(32768) As Byte
   Dim bufLen As Integer = 0
   Dim sb As StringBuilder
   Dim i As Integer
   bufLen = GetPrivateProfileSection(section, buffer, _
      buffer.GetUpperBound(0), filename)
   If bufLen > 0 Then
      sb = New StringBuilder()
      For i = 0 To bufLen - 1
         If buffer(i) <> 0 Then
            sb.Append(ChrW(buffer(i)))
         Else
            If sb.Length > 0 Then
               items.Add(sb.ToString())
               sb = New StringBuilder()
            End If
         End If
      Next
   End If
   Return items
End Function

Public Shared Function WriteINISection _
   (ByVal filename As String, ByVal section As String, _
   ByVal items As StringCollection) As Boolean
   Dim b(32768) As Byte
   Dim j As Integer = 0
   Dim s As String
   For Each s In items
      ASCIIEncoding.ASCII.GetBytes(s, 0, s.Length, b, j)
      j += s.Length
      b(j) = 0
      j += 1
   Next
   b(j) = 0
   Return WritePrivateProfileSection(section, _
      b, filename)
End Function

Public Shared Function GetINISectionNames _
   (ByVal filename As String) As StringCollection
   Dim sections As StringCollection = New _
      StringCollection()
   Dim buffer(32768) As Byte
   Dim bufLen As Integer = 0
   Dim sb As StringBuilder
   Dim i As Integer
   bufLen = GetPrivateProfileSectionNames(buffer, _
```

```
            buffer.GetUpperBound(0), filename)
         If bufLen > 0 Then
            sb = New StringBuilder()
            For i = 0 To bufLen - 1
               If buffer(i) <> 0 Then
                  sb.Append(ChrW(buffer(i)))
               Else
                  If sb.Length > 0 Then
                     sections.Add(sb.ToString())
                     sb = New StringBuilder()
                  End If
               End If
            Next
         End If
         Return sections
      End Function
   End Class
```

Moving Beyond Interop

All that COM Interop code is interesting, but simply reading existing INI files doesn't help you move them into XML. You've seen how to read and write INI files with .NET using DllImport to access the Windows API functions from within a C# or VB.NET class. Wrapping the Windows API functions lets you use existing INI files from .NET, but doesn't address the problems inherent in the INI file format itself—and INI files have a number of deficiencies. For example, total file size is limited to 64 KB total, individual values cannot exceed 256 characters, and the Windows API provides no programmatic way to read and write comments. Translating the files to XML solves these problems. The first task is to analyze exactly how INI files are constructed and decide how best to preserve their advantage (a text format that's easy to read and modify) while maintaining and extending the programmatic capabilities.

The Ubiquitous INI File

An INI file has three types of information: sections, keys, and values. A section is a string enclosed in square brackets; the keys and values are paired. A key does not have to have a value, but when present, an equal sign (=) separates the key from the value. The keys and values together create an *item*. The items are always "children" of a section header. Each section can have any number of child items. For example, here's a simple INI file structure:

```
[Section 1]
key=value

[Section 2]
key=value
key=value
```

INI files first appeared in Windows 3.*x*, and were originally intended to hold global Windows settings for various applications. Therefore, the *section* items of the INI file format were initially called *application*—a name that persists in the Win32 API function calls to this day, even though the later documentation uses the section/key/value terminology. Windows provides special APIs to read and write the `win.ini` file in the `Windows` folder, as well as functionally identical "private" versions that read and write values from named INI files specific to one application. Windows provides a number of functions in `kernel32.dll` to read and write INI files—all of which are marked as obsolete (see `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconupgraderecommendation-adjustdatatypesforwin32apis.asp` for more information).

Microsoft began calling the API functions obsolete *seven years ago* with the release of Windows 95, when it began touting the Registry as the perfect place to store application-level settings. The current MDSN documentation states that the functions are supported only for backward compatibility with Win16. Nevertheless, INI files have remained popular among developers, partly because Microsoft never made the Registry easy to use programmatically and partly for many of the same reasons that XML became popular: INI files are easy to understand; easy to modify, either manually or programmatically; and you can simply copy them from one machine to another. The strongest evidence in favor of INI files is that—despite Microsoft's insistence that INI files are obsolete—they're ubiquitous even in Microsoft software (search your local `Documents and Settings\accountName\Local Settings\Application Data` folder for examples).

Why Not Use .NET Configuration Files?

Your .NET applications are supposed to use the `AppSettings` section of configuration files to store key-value pair information. Unfortunately, the INI file format doesn't translate well to a simple list of key-value pairs, because you lose a "level" of information. Items in INI files exist as children of named sections; therefore, many INI files repeat key names for each section. Any line in an INI file that starts with a semicolon is a comment. For example, the sample code for this chapter uses this INI file:

```
; Company employees
[Employee1]
name=Bob Johnson
department=Accounting

[Employee2]
name=Susan Fielding
department=Sales
```

If you were to remove the section names and attempt to place this information into a configuration file as key-value pairs, you would have duplicate key names. In other words, the

default `AppSettings` data provides no way to group the items into something equivalent to INI sections. You *could* write a custom configuration handler to handle more levels of structure than configuration files support by default, but using a separate file is easier.

The XML INI File Structure

The `IniFileReader` project sample code for this solution reads standard INI files or XML-formatted INI files. Table 1 shows the `IniFileReaderNotInitialized` exception property, and Table 2 shows a complete list of methods and properties for the IniFileReader class.

TABLE 1: Class: IniFileReaderNotInitializedException

Property Type	Description	
Message (read-only)	String	The IniFileReader class throws this error when it can't successfully read the file passed to the constructor.

TABLE 2: Class: IniFileReader

Property/Method	Type	Description
Message (read-only)	String	The IniFileReader class throws this error when it can't successfully read the file passed to the constructor.
SetIniSection	Boolean	Updates a section name.
SetIniValue	Boolean	Updates the value associated with a specified section and key. When the keyName parameter is null, the method deletes the section. When the value parameter is null, the method deletes the specified key. Otherwise, if the specified sectionName or keyName do not already exist, the method creates them. If both the specified sectionName and keyName exist, the method updates the value with the string contained in the value parameter.
SetIniKey	Boolean	Updates a key name.
GetIniValue	String	Retrieves the value associated with a specified section and key.
GetIniComments	StringCollection	Retrieves a StringCollection filled with all the comments associated with a specified section.
SetIniComments	Boolean	Writes a set of comments contained in a StringCollection parameter to the specified section. The method first removes any existing comments. In this implementation, you must set and retrieve all comments at one time.

Continued on next page

TABLE 2: CONTINUED Class: IniFileReader

Property/Method	Type	Description
AllKeysInSection	StringCollection	Returns a StringCollection object filled with the key names associated with a specified section.
AllValuesInSection	StringCollection	Returns a StringCollection object filled with all the values associated with all keys in the specified section.
AllItemsInSection	StringCollection	Returns a StringCollection object filled with all the keys and values from the specific section. Each item has the form "key=value".
GetCustomIniAttribute	String	Returns the value of a specified attribute associated with a specified key and section.
SetCustomIniAttribute	Boolean	Updates the value of a specified attribute associated with a specified section and key. If the attribute does not exist, the method creates it.
Save	void	Saves the XML-formatted INI file to the file specified using the SaveAs property.
AsIniFile	String	Performs an XSLT transform to translate the loaded XML-formatted INI file back to a standard text INI file and returns the results as a string.
IniFilename (read-only)	String	Returns the name of the file passed to the new constructor.
Initialized (read-only)	Boolean	Returns true when the class has been properly initialized with a standard INI or XML-formatted INI file.
CaseSensitive (read-only)	Boolean	Returns a Boolean indicating whether the instance is case-sensitive or case-insensitive. When case-sensitive, all section, key, and attribute name parameters to the methods must match those in the source file exactly. The default is false (not case sensitive).
AllSections (read-only)	StringCollection	Returns a StringCollection object filled with the section names that exist in the file.
SaveAs	String	The filename to which the class will save the XML-formatted INI file during the next Save operation.
XmlDoc	XmlDocument	Returns the XmlDocument object containing the current XML-formatted INI file.
XML	String	Returns a string containing the XML-formatted contents of the current INI file.

The IniFileReader class constructor requires a filename. The class first tries to open the specified file as an XML file, using the `XmlDocument.Load` method. If that fails, the class assumes the file is in the INI format. It then creates a very simple default XML string and loads that, using the `XmlDocument.LoadXml` method, after which it opens the file in text mode and parses the lines of the file, adding elements for the sections, items, and comments in the order they appear (see Listing 2).

Listing 2 The ParseLineXML (IniFileReader.vb)

```
Private Sub ParseLineXml(ByVal s As String, _
   ByVal doc As XmlDocument)
   Dim key As String
   Dim value As String
   Dim N As XmlElement
   Dim Natt As XmlAttribute
   Dim parts() As String
   s.TrimStart()

   If s.Length = 0 Then
      Return
   End If
   Select Case (s.Substring(0, 1))
      Case "["
         ' this is a section
         ' trim the first and last characters
         s = s.TrimStart("[")
         s = s.TrimEnd("]")
         ' create a new section element
         CreateSection(s)
      Case ";"
         ' new comment
         N = doc.CreateElement("comment")
         N.InnerText = s.Substring(1)
         GetLastSection().AppendChild(N)
      Case Else
         ' split the string on the "=" sign, if present
         If (s.IndexOf("=") > 0) Then
            parts = s.Split("=")
            key = parts(0).Trim()
            value = parts(1).Trim()
         Else
            key = s
            value = ""
         End If
         N = doc.CreateElement("item")
         Natt = doc.CreateAttribute("key")
         Natt.Value = SetNameCase(key)
         N.Attributes.SetNamedItem(Natt)
         Natt = doc.CreateAttribute("value")
         Natt.Value = value
         N.Attributes.SetNamedItem(Natt)
         GetLastSection().AppendChild(N)
   End Select
End Sub
```

The sample INI file looks like this after the IniFileReader finishes loading it:

```
<?xml version="1.0" encoding="UTF-8"?>
<sections>
  <comment> Company employees</comment>
  <section name="employee1">
    <item key="name" value="Bob Johnson" />
    <item key="department" value="Accounting" />
  </section>
  <section name="employee2">
    <item key="name" value="Susan Fielding" />
    <item key="department" value="Sales" />
  </section>
</sections>
```

Getting the INI file contents into the simple XML structure makes it easy to mimic and extend the actions that you can perform with a standard INI file—and makes them easier to remember. The root `<sections>` element can contain any number of child `<comment>` or `<section>` elements, each of which can contain any number of `<item>` or `<comment>` elements.

You may be wondering why the project doesn't use the standard API functions through DllImport—as discussed in the article "Use COM Interop to Read and Write to INI Files with .NET" (see `www.devx.com/dotnet/discussions/040902/cominterop.asp`). This is because the standard API functions provide no way to read comments, so you can't get a complete translation using the API functions alone. Instead, for this particular purpose, it's better to parse the file line by line.

Retrieving Values

To retrieve the value of an item, you use the `GetIniValue` method, which accepts `sectionName` and `keyName` parameters. The method creates an XPath query that searches for the section with a `name` attribute matching the supplied section name, and then searches within that section for an item with a `key` attribute matching the supplied key name. If the XPath query matches an item, the function returns the text value of the `value` attribute of that item; otherwise it returns `Nothing` (`null` in C#).

```
' ********************************************
' * VB.NET code                              *
' ********************************************
Public Function GetIniValue( _
   ByVal sectionName As String, _
   ByVal keyName As String) As String
   If Not Initialized Then Throw New _
      IniFileReaderNotInitializedException()
   Dim N As XmlNode = GetItem(sectionName, keyName)
```

```
    If Not N Is Nothing Then
       Return (N.Attributes.GetNamedItem("value").Value)
    End If
    Return Nothing
End Function
```

There is one major difference between XML-formatted files and INI files—XML files are case sensitive, while standard INI files aren't. The INIFileReader class deals with this potential problem by treating all data as case-insensitive by default. The following section provides a more detailed discussion of the problem and the solution.

Dealing with XML's Case Sensitivity

Case sensitivity is an intrinsic feature of many languages, and XML is no exception. Unfortunately, case sensitivity is also one common source of bugs in code. Because the old INI files were not case-sensitive, translating them directly to XML risks breaking existing code logic because of the difference in the way the Windows API function treats case and the way an XML parser treats case. Therefore, the IniFileReader class uses only lowercase tags by default; however, it treats queries in a case-insensitive manner by first converting the section or key names to lowercase and then performing the query. While the change in case sensitivity might cause problems for some file types due to section or key name conflicts where the only difference was in case, it's not a problem for INI files—the API functions for INI file update and retrieval aren't case-sensitive either. Therefore, translating all the section and key names to lowercase has no effect on file modifications—it just makes the names look different.

If you prefer to use the IniFileReader in case-sensitive mode, you can set the `CaseSensitive` property to False in VB.NET (false in C#). If you prefer this, remember that you must set the `CaseSensitive` property when you instantiate an IniFileReader using the optional overloaded constructor as shown in the following code snippet. After creating an instance of the class, there's no way to change the `CaseSensitive` property value.

```
ifr = new IniFilereader(someFilename, True);
```

All methods that retrieve nodes apply the results of the private `setNameCase` method to parameters. The method ensures that names are shifted to lowercase when `CaseSensitive` is true.

```
Private Function SetNameCase( _
   ByVal aName As String) As String
   If (CaseSensitive) Then
      Return aName
   Else
      Return aName.ToLower()
   End If
End Function
```

So to retrieve a section, for example, the `GetSection` code checks to ensure that the `sectionName` argument is not Nothing or an empty string, and then calls `SetNameCase` before constructing the XPath query that searches for the section node:

```
Private Function GetSection( _
   ByVal sectionName As String) As XmlElement
   If (Not (sectionName = Nothing)) AndAlso _
      (sectionName <> String.Empty) Then
      sectionName = SetNameCase(sectionName)
      Return CType(m_XmlDoc.SelectSingleNode _
         ("//section[@name='" & sectionName & "']"), _
         XmlElement)
   End If
   Return Nothing
End Function
```

Updating and Adding Items

Updating individual values is similar to retrieving them. You provide a section name, a key name, and the new value. The class uses the `GetItem` method to locate the appropriate `<item>` element, and then updates that item's `value` attribute with the specified new value. The Windows API function `WritePrivateProfileString` creates new sections and items if you call it with a section or key name that doesn't already exist.

Although it's not good object-oriented design, for consistency the IniFileReader class acts identically, meaning that you can create a new section simply by passing a nonexistent section name. To update a section name, key name, or value for existing sections or items, select the section and item you want to update, enter the new values in the appropriate fields, and then click the Update button to apply the changes. To create a new section or key on the sample form, first click the New Section or New Key button to clear the current selections, and then enter the new section or key name and click the Update button to apply your changes.

To delete a section using the API, you pass the section name and a null key value to the `WritePrivateProfileString` function—and you do the same with the IniFileReader class, except that you use the `SetIniValue` method. For example, the following code would delete the section named `section1`.

```
SetIniValue("section1", Nothing, Nothing)
```

Similarly, to delete an item within a section, you pass the section name, the key name, and a null value for the `value` argument. The following code would delete the item with the key `key1` in the section named `section1`.

```
SetIniValue("section1", "key1", Nothing)
```

On the sample form (see Figure 1), you can delete a section or value by selecting the appropriate item in either the section or item list, and then pressing the Delete key.

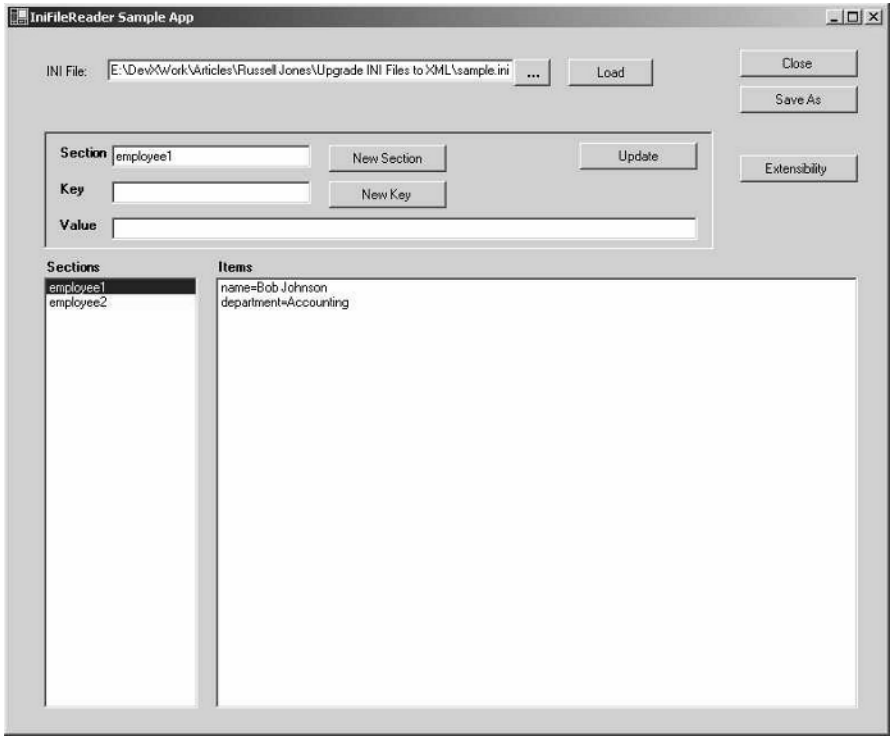


FIGURE 1: The sample form lets you test the INIFileReader methods and edit INI files.

The class has several other properties that may interest you. First, the `Save` method saves the file to a filename you specify using the `OutputFilename` property. The `Save` method checks whether the specified directory exists and then uses the `Save` method of the underlying XmlDocument to save the file.

Second, the `XmlDoc` property gives the calling code direct access to the underlying XmlDocument object. The IniFileReader class has a number of properties that extend the standard API functionality. For example, the `GetAllSections` method retrieves all the section names. Whenever the class returns multiple values, it returns a StringCollection object rather than a string array. While this marginally affects the class's performance, from the caller point of view StringCollections are more convenient than simple arrays.

Extensibility

The three types of information in a standard INI file often did not suffice to store the information needed by the application. I've seen (and built) applications that rely on custom string formats to perform tasks beyond the INI file format's native capabilities. For example, it's not uncommon to see INI files containing strings that use separators to "squash" more values into an INI file:

```
[testing]
emp1=Bob Johnson|Accounting|04/03/2001 8:23:14|85
emp2=Susan Fielding|Sales|03/23/2001 15:41:48|92
```

Not only are such files difficult to read and edit—and maintain—but they also require custom code to retrieve the item values and assign them to variables within the program. The data in such files is unusable by other programs without re-creating the code to retrieve the data. In contrast, you can add new items to an XML-formatted INI file with few problems. At the most simplistic level, you can use the `GetCustomIniAttribute` and `SetCustomIniAttribute` methods to read, write, and delete custom strings, stored as attributes within the `<item>` elements. For example, the following XML document shows the same data shown in the preceding INI file added as custom attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<sections>
  <comment>Company employees</comment>
  <section name="employees">
    <item key="employee1" value="Bob Johnson"
       department="Accounting"
       testedon="04/03/2001
       8:23:14" score="85"/>
    <item key="employee2"
       value="Susan Fielding"
       department="Sales"
       testedon="03/23/2001 15:41:48"
       score="92" />
  </section>
</sections>
```

It's much easier to discover the *meaning* of the data in the XML version.

At a more complex level, although I haven't implemented it in the sample code, you could add `GetCustomIniElement` and `SetCustomIniElement` methods to add custom child elements and values to the `<item>` elements. These methods would be overloaded to accept an element name and value, an XmlElement instance, or an XmlDocumentFragment instance, so you

could make the file as complicated as necessary. The Extensibility button on the sample form contains code that shows how to use the extensibility methods.

Beyond the built-in extensibility methods, you can, of course, subclass the IniFileReader and override or add methods to do anything you like.

Dealing with Comments

You use the `SetIniComments` and `GetIniComments` methods to add and retrieve comments. The `GetIniComments` method returns a StringCollection containing the text of comments at either the file or section level. The `SetIniComments` method accepts a StringCollection containing a list of comments you want to add at either the file or section level. While this implementation is very crude, and could be greatly improved—for example, you could extend the class to attach comments directly to individual items—it's already an improvement over standard INI files, which provide no way to create or remove comments automatically. You can also add XML-formatted comments manually or use the DOM directly to add comments to an XML-formatted INI file.

Where's My INI File?

For straightforward (unextended) files, you can "get the original INI file back." In some cases, you will want to read and modify the INI file with a .NET application, but you still need access to the file in its original format usable by pre-.NET Windows applications. An XSLT transform performs the work of turning the file back into a standard INI file. You initiate the transform using the `SaveAsIniFile` method. However, extensibility comes at a price. If you make *custom* modifications to the file using the `writeCustomIniAttribute` method, those changes will *not* appear in the results of the transform; however, there's little reason to translate the files back to standard INI format, so that restriction seems reasonable.

Having a separate file for storing application initialization information is a good idea. The .NET Framework contains built-in methods for handling configuration data, but—as delivered—they aren't suitable for complex information structures, nor are they dynamically updatable using the Configuration classes in the Framework. As you migrate existing applications into .NET, the INIFileReader class described in this solution lets you use and even extend your existing INI files. Nonetheless, .NET configuration files have some advantages even over these custom external XML-formatted initialization files, and you should study their capabilities.

SOLUTION 4

Build Your Own XML-Enabled Windows Forms TreeView Control

PROBLEM Displaying XML in a TreeView seems like it would be a no-brainer—and for some simple, regular XML documents, it is. But for complex, mixed-content documents, or documents with content split between attributes and elements, or documents containing unneeded nodes, displaying the XML becomes much more complicated.

SOLUTION Because of XML's simple, repeating syntax, you can treat any well-formed XML document generically. Create this XML-enabled TreeView control and display customized views of any well-formed XML document.

The .NET Windows Forms TreeView control lets you display a hierarchical view of information; therefore, it's a perfect match for displaying data in XML documents, which is innately hierarchical. But the Windows Forms TreeView control can't display an XML document natively—if you want to display XML in a TreeView, you have to add the functionality yourself.

The basic process is simple; iterate through the nodes in the XML document. For each node, add a new TreeNode and set its text to the text of the current XML node—and that's the idea behind most of the examples you'll find about filling TreeView controls with XML data. But those examples use XML that's *conducive* to TreeView display; when you try to use them directly on many XML documents, they work—but not the way you want them to work.

Iterating through Nodes

The iteration process is recursive. A method such as the `AddNode` method shown in the following code snippet (from `http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q308063`) expects an XmlNode argument and a TreeNode argument. The method tests to see if the XML node has child nodes. If so, it gets a list of the child nodes and iterates through them, creating a new TreeNode and calling itself recursively for each child node. Finally, it sets the text of the TreeNode to the `OuterXML` property of the XML node. The comments in the code indicate that you might want to change that based on the type of node.

```
Private Sub AddNode(ByRef inXmlNode As XmlNode, _
   ByRef inTreeNode As TreeNode)
```

```
    Dim xNode As XmlNode
    Dim tNode As TreeNode
    Dim nodeList As XmlNodeList
    Dim i As Long

    ' Loop through the XML nodes until the leaf is reached.
    ' Add the nodes to the TreeView during the looping
    ' process.
    If inXmlNode.HasChildNodes() Then
       nodeList = inXmlNode.ChildNodes
       For i = 0 To nodeList.Count - 1
          xNode = inXmlNode.ChildNodes(i)
          inTreeNode.Nodes.Add(New TreeNode(xNode.Name))
          tNode = inTreeNode.Nodes(i)
          AddNode(xNode, tNode)
       Next
    Else
       ' Here you need to pull the data from the XmlNode
       ' based on the type of node, whether attribute
       ' values are required, and so forth.
       inTreeNode.Text = (inXmlNode.OuterXml).Trim
    End If
End Sub
```

So basically, if you pass the `AddNode` method the root element of an XML document, it will fill the TreeView control with the XML of each node in the document. Here's another example. Suppose you have a simple XML document that looks like Listing 1.

Listing 1 **A simple XML document (SimpleXML.xml)**

```
<?xml version="1.0" encoding="Windows-1252"?>
<departments>
   <department id="d3" name="IT" expanded="False">
      <employees>
         <employee id="e50">
            <lastname>Chen</lastname>
            <firstname>Kelly</firstname>
            <address><![CDATA[37118 Second Hill
                Dr.]]></address>
            <email>kchen@company.com</email>
            <phones>
               <work>(253) 703-7277</work>
               <home>(253) 703-3168</home>
               <fax>(253) 703-5633</fax>
            </phones>
            <active>True</active>
         </employee>
         <employee id="e45">
            <lastname>Chen</lastname>
```

```
            <firstname>William</firstname>
            <address>133 S. Chilhowee Dr.</address>
            <email>wchen@company.com</email>
            <phones>
               <work>(221) 410-3615</work>
               <home>(221) 410-0402</home>
               <fax>(221) 410-3788</fax>
            </phones>
            <active>False</active>
         </employee>
       <!-- more employees here -->
      </employees>
   </department>
   <!-- more departments here -->
</departments>
```

To see the results, you create an XmlDocument object and load the XML file into it, create a root TreeView node, and call the `AddNode` method, passing the `XmlDocument.DocumentElement` and the root TreeNode as arguments. For example, the following code loads the sample `employee.xml` file:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
   Dim doc As New XmlDocument()
   Dim execFile As FileInfo = New _
      FileInfo(Application.ExecutablePath)
   Dim fldr As DirectoryInfo = execFile.Directory
   fldr = fldr.Parent
Dim filename As String = "\employees.xml"
   doc.Load(fldr.FullName & filename)
   Dim tNode As TreeNode = Me.TreeView1.Nodes.Add _
      (doc.DocumentElement.Name)
   AddNode(doc.DocumentElement, tNode)
   Me.TreeView1.ExpandAll()
End Sub
```

The results look like Figure 1.

The first thing you'll probably notice is that it looks "different" than you might expect. Sure, it contains the data, but it also has a number of little problems. First, every node has a child node. Because the final line in the `Form1_Load` code calls the TreeView's `ExpandAll` method, it looks slightly less onerous than it is. But if you collapse all the nodes in one of the employee nodes, for example, you'll find that to see any value, your users will have to drill down one more level than you might think. In addition, your users probably don't care about the XML document element names—and you might not even *want* your users to see those names.

FIGURE 1:

The results of loading a TreeView control with XML using the AddNode method

To be fair, the comment at the bottom of the `AddNode` method states that you would probably want to customize it based on your needs. In this case, it would be much better if you could ignore the `<employees>` nodes and simply show the names of each department, with the names of the employees below that.

Similarly, note that the `AddNode` method doesn't show attributes. You would probably want to show the name of each department, which, in the `employees.xml` file, resides in an attribute of each `<department>` node. To do that, you'd need to add code in the `Else` block, such as

```
If inXmlNode.LocalName = "department" Then
   inTreeNode.Text = inXmlNode.getAttribute("name")
End If
```

It's easy to create such custom code and make it work for *this* document. But if you take that route for every XML document you want to display, you'll have to customize an `AddNode` implementation each and every time. There's a better way.

What Capabilities Do You Need?

Rather than writing custom code, consider the functionality you might *like* to have in a TreeView that displays XML. The simplest way to extend the functionality of controls within the .NET Framework is to subclass them, so one solution is to create a TreeViewXml control that's capable of displaying XML in a TreeView control in customized ways.

Right off the bat, you'll probably decide that it's usually much more intuitive to show the text string for a node than to show the tag names. The tag names may (or may not) help users understand exactly what they're looking at. For example, there's little point in showing this in a TreeView…

```
<departments>
   <department>
      IT
   <department>
      Marketing
   <department>
      Manufacturing
```

…when you could show this:

```
departments
   IT
   Marketing
   Manufacturing
```

The second version displays only the data, not the element names, but the intent is clear, and it's far easier to read. To display the data without the names, you must be able to identify elements based on their relative hierarchical position in the document.

You'll also want to be able to *hide* elements based on their name or their position relative to the document itself—in other words, their *path*. Here's the full path for the department name attribute and the type:

```
departments/department/name
departments/department/employees/employee/name
```

Using the node path rather than a simple name clearly differentiates between two identically named nodes in different locations within the node tree.

While many XML documents already have exactly the content you want in exactly the format you need, many don't. The goal is to identify the nodes you want to change (using their path) and then alter the contents. For example, you don't have to show the `<department>` tag; you want to show the `name` attribute value instead. Similarly, the `<employee>` nodes have `<lastname>` and `<firstname>` children, but rather than displaying them separately, it would be far friendlier to concatenate them, displaying the name as a single `lastname, firstname` string.

In other words, you want the capability to "surface" attributes and child elements and display them at the parent level. In the sample code for this solution, I've termed that functionality "display rules." The code that defines each rule is localized to the DisplayRule class.

The DisplayRule Class

The following code contains five types of display rules, exemplified by the DisplayRuleContentType enumeration in the class:

```
Public Enum DisplayRuleContentType
   ShowOnlyNodesNamed
   HideNodesNamed
   ShowOnlyNodesWithPath
   HideNodesWithPath
   XpathQuery
   XslTemplate
End Enum
```

These five are by no means comprehensive, and you may find many ways to extend the class to provide additional customizing capabilities. To do that, you can add new types to the enumeration. However, these five are sufficient for many purposes. Table 1 shows the effect of each DisplayRuleContentType.

TABLE 1: Display Rules

DisplayRuleContentType	Rule Effect
`ShowOnlyNodesNamed`	Causes the TreeViewXml control to hide all nodes not associated with the rule type
`HideNodesNamed`	Causes the TreeViewXml control to hide nodes associated with the rule type
`ShowOnlyNodesWithPath`	Causes the TreeViewXml control to hide all nodes whose path does not match one of the paths associated with the rule type
`HideNodesWithPath`	Causes the TreeViewXml control to hide all nodes whose path matches one of the paths associated with the rule type
`XpathQuery`	Causes the TreeViewXml control to display the result of an XPath query for all nodes whose path matches the path associated with the rule type
`XslTemplate`	Causes the TreeViewXml control to display the result of an XSLT style sheet applied to all nodes whose path matches the path associated with the rule type

The first four types simply hide or show nodes. The `ShowOnlyNodesNamed` and `HideNodesNamed` types show and hide nodes, respectively; the difference is that when you add nodes to the `ShowOnlyNodesNamed` list, the TreeViewXml control shows *only* nodes associated with that rule—in other words, if you begin adding element names to this rule type, you must add *all* the names you want to display. In contrast, the `HideNodesNamed` rule simply hides the nodes associated with the rule. The `ShowOnlyNodesWithPath` and `HideNodesWithPath` rules work identically but use node paths rather than the node `LocalName` property.

DisplayRule Properties

Each DisplayRule has three properties: `Name`, `Value`, and `DisplayRuleType`. The `DisplayRuleType` property is one of the `DisplayRuleContentType` enumeration values shown in Table 1.

The `Name` property holds either a simple node `LocalName` or a path from the document root to some element. For example, "employees" is the `LocalName` for all the `<employees>` elements. In contrast, the path from the document root to each `<employees>` element is `departments/department/employees`. This dual naming scheme simplifies element identification; when the element names are unique within the document, you can use the `LocalName`, and when they're not, you can use the path.

Each DisplayRule applies either to all elements whose `LocalName` (such as "employees" or "address") matches the value of its `Name` property—regardless of where those elements appear in the document—or where the path stored in the `Name` property matches the path to a node in the document. For example, a `DisplayRule.Name` value of `departments/department/employees/employee/lastname` applies to all `<lastname>` child elements of each `<employee>` element.

The `Value` property holds either an XPath query or an XslTransform object. Simple show/hide DisplayRule types don't require a value; you need the `Value` property only when you plan to alter the display value of an element or attribute.

Creating DisplayRules

You create display rules by adding them to a DisplayRulesCollection exposed by the custom TreeViewXml class.

For example, using the sample document from Listing 1, you can hide everything but the `<departments>` and `<department>` nodes by creating two DisplayRule objects whose `Name` property contains the strings `"departments"` and `"department"`, respectively, and the `DisplayRuleType` property is `ShowOnlyNodesNamed`.

As another example, the `<department>` nodes contain a `name` attribute. To display, or surface, that `name` attribute in the TreeNode for each `<department>` node, you can create a DisplayRule that identifies `<department>` nodes by their path and then display the result of an XPath query that retrieves the `name` attribute value. In other words, the `Name` property would be `departments/department`, the `Value` property would be the XPath query `@name`, and the `DisplayRuleType` property would be `XpathQuery`.

Simple XPath substitutions work fine for surfacing attributes, but not so well for surfacing or concatenating child elements. Therefore, display rules also accept an instance of the XslTransform class. The rule substitutes the output of the XslTransform applied to the nodes

identified by the DisplayRule. For example, to obtain a string containing each employee's ID followed by that employee's name in lastname/firstname order, you could write this XSLT style sheet:

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet version='1.0'
   xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method='text' encoding='UTF-8'/>

<xsl:template match=""employee"">
   <xsl:value-of select=""@id""/>: <xsl:value-of
      select=""lastname"" />, <xsl:value-of select=""firstname"" />
</xsl:template></xsl:stylesheet>
```

When you transform an employee node using this style sheet, the output is a string that looks like this:

```
e45: Chen, Kelly
```

In other words, the text assigned to the TreeNode for each `<employee>` node is the output of the transformation.

Finding Node Paths

To identify nodes unambiguously based on their path, you must be able to obtain the path, given any node in an XML document. You can do that by walking recursively up the Document Object Model (DOM) tree until you reach the document node, concatenating a string that describes the relative hierarchical position of the node in question.

Listing 2 shows a `getPath` method that returns the path string for a node.

Listing 2 **The getPath method returns the path from the document root to any child node. (TreeViewXml.vb)**

```
Private Function getPath(ByVal aNode As XmlNode, _
   ByVal sPath As String) As String
   If sPath Is Nothing Or sPath Is String.Empty Then
      If N.NodeType = XmlNodeType.Attribute Then
         sPath = "@" & N.LocalName
      Else
         sPath = N.LocalName
      End If
   Else
         sPath = N.LocalName & "/" & sPath
   End If
   If Not N.ParentNode Is Nothing Then
```

```
        ' call this function recursively until you reach the
        ' document node
        If N.ParentNode.NodeType <> XmlNodeType.Document Then
           sPath = getPath(N.ParentNode, sPath)
        End If
     End If
     Return sPath
  End Function
```

The method begins by checking the length of the `sPath` String argument. If it's empty, the method sets it to the `LocalName` property of the XmlNode argument.

Next, the method checks to see if the `Parent` property of the `XmlNode` is `Nothing`. If not, and the parent node is not the root node (the `XmlNodeType.Document` node), it calls itself recursively, passing the current value of `sPath` and the parent node as arguments. In effect, this builds the path string by moving "upward" through the document. The process stops when the current node's parent *is* the root node.

Using the concatenated path string, you can check each node against a set of display rules to see how to treat that node. Display rules consist of a key-value pair where the key is a path string and the value is either an XPath string or an XslTransform object. The DisplayRulesCollection class shown in Listing 3 implements the collection.

Listing 3 **The DisplayRulesCollection class contains a collection of key-value pairs used to determine node display values. (DisplayRulesCollection.vb)**

```
Option Strict On
Imports System.Collections.Specialized
Imports System.Xml.Xsl
Public Class DisplayRulesCollection
   Inherits System.Collections.CollectionBase

   Public Sub Add(ByVal aDisplayRule As DisplayRule)
      Me.List.Add(aDisplayRule)
   End Sub

   Public Sub Add(ByVal nodeName As String, _
      ByVal aType As DisplayRule.DisplayRuleContentType)
      Me.Add(New DisplayRule(nodeName, aType))
   End Sub

   Public Sub Add(ByVal nodePath As String, _
      ByVal anXPathQuery As String, _
      ByVal aType As DisplayRule.DisplayRuleContentType)
      Me.Add(New DisplayRule(nodePath, anXPathQuery, _
         DisplayRule.DisplayRuleContentType.XpathQuery))
   End Sub

   Public Sub Add(ByVal nodePath As String, _
      ByVal anXslTransform As XslTransform, _
```

```
      ByVal aType As DisplayRule.DisplayRuleContentType)
      Me.Add(New DisplayRule(nodePath, anXslTransform, _
         DisplayRule.DisplayRuleContentType.XslTemplate))
   End Sub

   Public Sub Add(ByVal nodeNames As String(), _
      ByVal aType As DisplayRule.DisplayRuleContentType)
      Dim nodeName As String
      For Each nodeName In nodeNames
         Me.Add(New DisplayRule(nodeName, aType))
      Next
   End Sub

   Public Sub Remove(ByVal key As String)
      Dim dr As DisplayRule
      Dim o As Object
      For Each o In Me.List
         dr = DirectCast(o, DisplayRule)
         If dr.Name = key Then
            Me.List.Remove(o)
         End If
      Next
   End Sub
   Public Sub Remove(ByVal key As String, _
      ByVal aType As DisplayRule.DisplayRuleContentType)
      Dim dr As DisplayRule
      Dim o As Object
      For Each o In Me.List
         dr = DirectCast(o, DisplayRule)
         If dr.Name = key AndAlso _
            dr.DisplayRuleType = aType Then
            Me.List.Remove(o)
         End If
      Next
   End Sub

   Public ReadOnly Property GetRulesOfType( _
      ByVal aType As DisplayRule.DisplayRuleContentType) _
      As DisplayRuleSet
      Get
         Dim dr As DisplayRule
         Dim o As Object
         Dim drList As New DisplayRuleSet()
         For Each o In Me.List
            dr = DirectCast(o, DisplayRule)
            If dr.DisplayRuleType = aType Then
               drList.Add(dr)
            End If
         Next
         Return drList
      End Get
   End Property
End Class
```

The class is by no means a *complete* typed collection implementation—it contains only the functionality needed for the sample project that accompanies this solution. Note the overridden `Add` methods, which simplify adding the various types of DisplayRules you can add to the class.

The DisplayRulesCollection class makes it easy to retrieve any rules that apply to a specific DisplayRuleContentType. The `GetRulesOfType` method returns all the rules that match a specific type as a DisplayRuleSet object. Because the DisplayRuleSet class inherits from DictionaryBase, you can then use the `Exists` method to test whether a particular node's `LocalName` or path matches an existing rule.

So far, you've seen how to fill a TreeView with XML using a recursive method, how to identify specific nodes so you can alter the display characteristics for those nodes, and how you can use XPath and XSLT style sheets to manipulate the text content displayed for any particular node. All you need to do now is wrap all that up into a class that provides all the functionality.

The TreeViewXml Control

The TreeViewXml class in the sample project inherits from TreeView, and adds the capabilities you need. To display an XML document in a TreeViewXml instance, you call its overloaded `LoadXml` method, passing the name of an XML file, an XML string, or a populated XmlDocument object. In all cases, the control's response is the same—it clears any existing nodes, and then populates itself from the specified data source.

At the simplest level, you can simply pass a filename or string containing XML to the control. When you do that, the control displays all elements and all attributes.

The sample project form named Form2 displays a TreeViewXml control and three buttons.

NOTE You'll need to switch the startup form to Form2 in the project properties dialog box to run Form2.

At startup, the form retrieves a string containing the contents of an embedded XML resource file named `employees.xml` and stores it in a String variable named `employeesXml` using the following code:

```
Dim sr As New System.IO.StreamReader( _
    [Assembly].GetEntryAssembly. _
    GetManifestResourceStream( _
    "TreeViewXMLTest.employees.xml"))
employeesXml = sr.ReadToEnd
sr.Close()
```

If you weren't aware that you could embed files as resources in your .NET assemblies, see Anthony Glenwright's excellent article, "How to Embed Resource Files in .NET Assemblies," which you can find at `www.devx.com/dotnet/Article/10831/0/page/1`.

Using the TreeViewXml Control

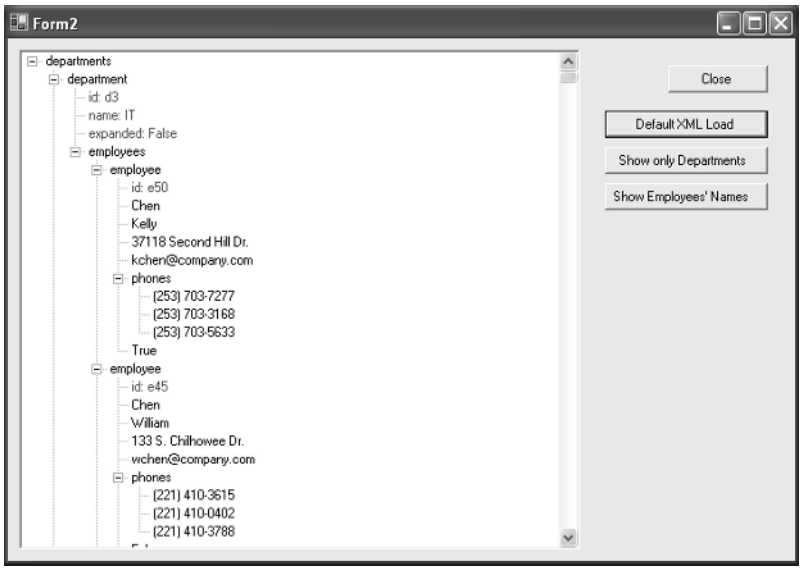
The form creates the TreeViewXml control at startup and uses the variable `tvx1` to refer to the control throughout the class. The Default XML Load button sets a few properties and displays the `employees.xml` string by calling the TreeViewXml control's `LoadXml` method:

```
Private Sub btnDefaultLoad_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles btnDefaultLoad.Click

   tvx1.ShowAttributesAsChildren = True
   tvx1.AttributeColor = Color.Red
   tvx1.DisplayRules.Clear()

   ' show the tree
   tvx1.LoadXml(employeesXml)
End Sub
```

The `ShowAttributesAsChildren` property takes a Boolean value and controls whether the TreeViewXml instance displays attributes as child nodes below each element. By default, the value is `False`. The `AttributeColor` property accepts or returns a `Color` value that determines the color in which the control displays attributes. The default attribute color is `Color.Blue`. The code clears the `DisplayRules` collection to ensure that the XML will display without applying any rules that may have been added when you clicked the other buttons on the form.

Figure 2 shows Form2 after clicking the Default XML Load button.

FIGURE 2:
The default XML string load



Using the DisplayRules collection, a few simple commands can alter the display of the XML content radically. For example, the Show Only Departments button causes the control to display only the root <departments> node and the child nodes for each department. It uses one DisplayRule to limit the tree display to only those two elements, and a second to apply an XPath query result to each <department> element, causing the tree to display the value of that element's name attribute (see Listing 4).

Listing 4 Using DisplayRules to control the tree display

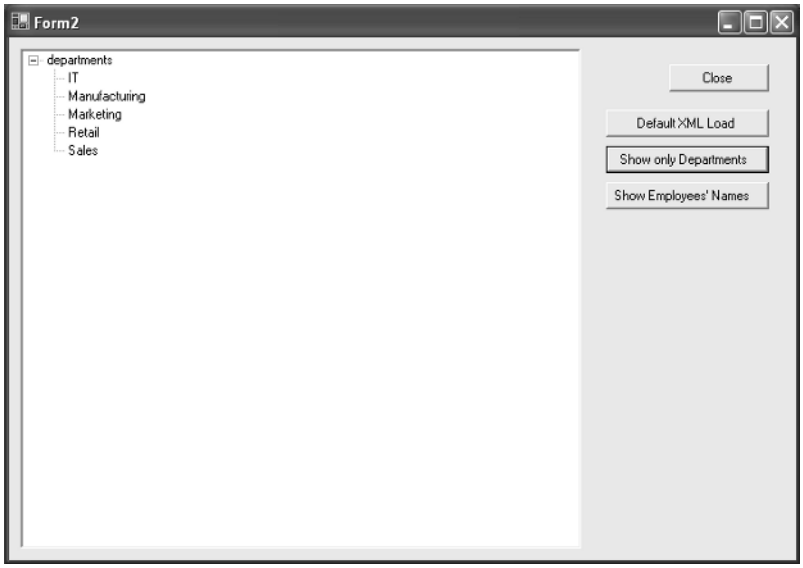
```
Private Sub btnDepartments_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles btnDepartments.Click

   tvx1.ShowAttributesAsChildren = False
   tvx1.DisplayRules.Clear()

   ' display only departments and department elements
   tvx1.DisplayRules.Add( _
      New String() {"departments", "department"}, _
      DisplayRule.DisplayRuleContentType.ShowOnlyNodesNamed)

   ' Display the value of the name attribute for
   ' each department
   tvx1.DisplayRules.Add("departments/department", _
      "./@name", _
      DisplayRule.DisplayRuleContentType.XpathQuery)

   ' load the tree
   tvx1.LoadXml(employeesXml)
End Sub
```

Figure 3 shows the results.

You can alter the display even further by using an XSLT style sheet to modify the display for specific nodes. Clicking the Show Employees' Names button displays all the departments and all the employees within each department, showing each employee's id attribute value and the text values of its child <lastname> and <firstname> concatenated into a single string, as discussed earlier in the section "What Capabilities Do You Need." The button code obtains the style sheet by reading another embedded resource file named ConcatEmpNames.xsl. Again, the code for the button Click event shown in Listing 5 simply creates the DisplayRules and calls the LoadXml method.

FIGURE 3:

Displaying only departments



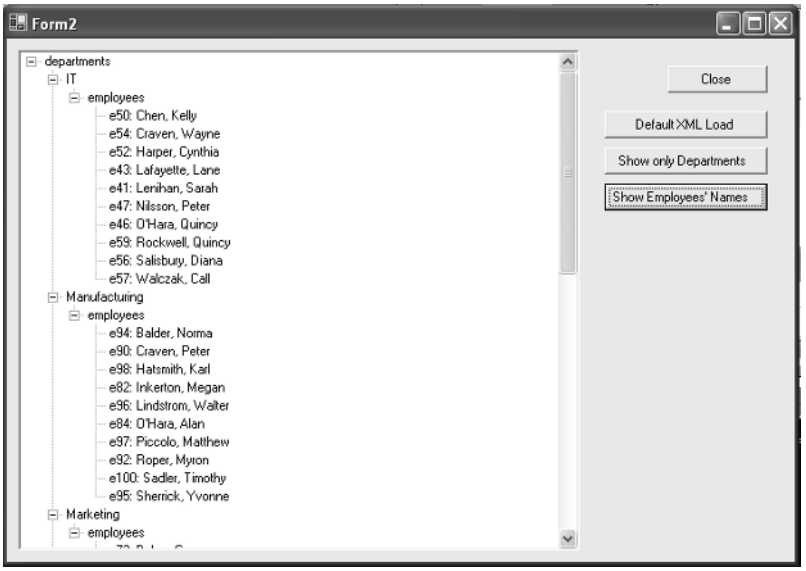
Listing 5 Using an XSLT style sheet to display customized node content

```
Private Sub btnEmpNames_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles btnEmpNames.Click

   tvx1.ShowAttributesAsChildren = False
   tvx1.DisplayRules.Clear()

   ' display only the following elements
   tvx1.DisplayRules.Add(New String() {"departments", _
      "department", "employees", "employee"}, _
      DisplayRule.DisplayRuleContentType. _
      ShowOnlyNodesNamed)

   ' display department names
   tvx1.DisplayRules.Add("departments/department", _
      "./@name", _
      DisplayRule.DisplayRuleContentType.XpathQuery)

   ' load the XSLT stylesheet
   Dim template As New XslTransform()
   template.Load(New XmlTextReader( _
      New System.IO.StringReader(concatEmpNameXsl)))
```

```
    ' add a DisplayRule for employee nodes
    ' using the XSLT stylesheet and a
    ' DisplayRuleContentType of XslTemplate
    tvx1.DisplayRules.Add( _
       "departments/department/employees/employee", _
       template, _
       DisplayRule.DisplayRuleContentType.XslTemplate)
    tvx1.LoadXml(employeesXml)
End Sub
```

Figure 4 shows the results.

FIGURE 4: Using XSLT to display custom employee node strings



How TreeViewXml Works

When you call the TreeViewXml class's `LoadXml` method, the control creates one DisplayRuleSet object for each DisplayRuleContentType by calling the private `InitializeDisplayRuleSets` method, which in turn calls the `DisplayRulesCollection.GetRulesOfType` method:

```
Private Sub InitializeDisplayRuleSets()
    ' for each type in the
    ' DisplayRule.DisplayRuleContentType enumeration
    ' get a DisplayRuleCollection containing only
    ' rules of that type

    mShowOnlyNodesNamed = Me.DisplayRules.GetRulesOfType _
```

```
      (DisplayRule.DisplayRuleContentType. _
      ShowOnlyNodesNamed)

   Me.mShowOnlyNodesWithPath = _
      Me.DisplayRules.GetRulesOfType _
      (DisplayRule.DisplayRuleContentType. _
      ShowOnlyNodesWithPath)

   ' ... additional calls to GetRulesOfType here

   End Sub
```

Next, the control loads the XML passed to the `LoadXml` method (except for the overloaded version that accepts an XmlDocument object), calls the `BeginUpdate` method, clears any existing nodes from the TreeView control, and then attempts to create a set of TreeNodes using the XML by passing the document element to the `FillTree` method. The `FillTree` method recursively fills the tree in a manner similar to the `AddNode` method shown in the section "Iterating through Nodes"; however, it adds checks to determine whether it should display each node based on the various hide/show DisplayRuleSets, using code such as the following:

```
' the ShowOnlyNodesNamed list contains a list of
' element names that you want to show. Check to see
' if this node is in that list.
If Me.mShowOnlyNodesNamed.Count > 0 Then
   showNode = Me.mShowOnlyNodesNamed.Exists(N.LocalName)
End If

' the HideNodesNamed list contains a list of element
' names that you want to hide. Check to see if this node
' is in that list.
If showNode Then
   If Me.mHideNodesNamed.Count > 0 Then
      showNode = Not mHideNodesNamed.Exists(N.LocalName)
   End If
End If
' ... additional similar checks here
```

Whenever the `FillTree` method determines that it *should* display a node, it calls the `setNodeText` method, which checks to see if the node is associated with an XPathQuery or XslTransform DisplayRule. If so, it executes the query or performs the transformation and applies the result to the TreeNode's `Text` property; otherwise, it either uses the XML node's `LocalName` for the `TreeNode.Text` property (when the XML node has child nodes), or simply assigns the XML node's `InnerText` property as the text.

So, the overall logic flow is:

1. For each node...
2. Determine whether to show or hide the node.
3. For displayed nodes, apply XPathQuery or XslTransform rules, if any, or display the node name (non-leaf nodes) or the node text content (leaf nodes).

One final note: The `FillTree` method returns a single TreeNode containing a hierarchical set of TreeNode objects. The calling code then uses the inherited `TreeView.Nodes.AddRange` method to attach all the nodes to the TreeView control in a single operation. In contrast, if you add nodes to the TreeView directly in the `FillTree` method, it takes many times as long to populate the control.

TreeViewXml Extensions

The sample version simplifies the process for displaying customized XML in a TreeView control, but there are many additional ways to extend the TreeViewXml control so you can provide even more customization. For example, you might want to create a new DisplayRuleContentType that accepts a Delegate for the `Value` property, so that you could run custom code to create a display string for specific nodes.

You could add `Color` and `Font` properties to the DisplayRule class and additional constructors so that you could specify the `ForeColor`, `BackColor`, and `Font` for specific nodes. Note that, as delivered, the DisplayRule class constructors are specified as `Friend`, so you'll also have to create additional overloaded `Add` methods for the DisplayRulesCollection class to instantiate new DisplayRule types or add properties.