DataBinding: Collections

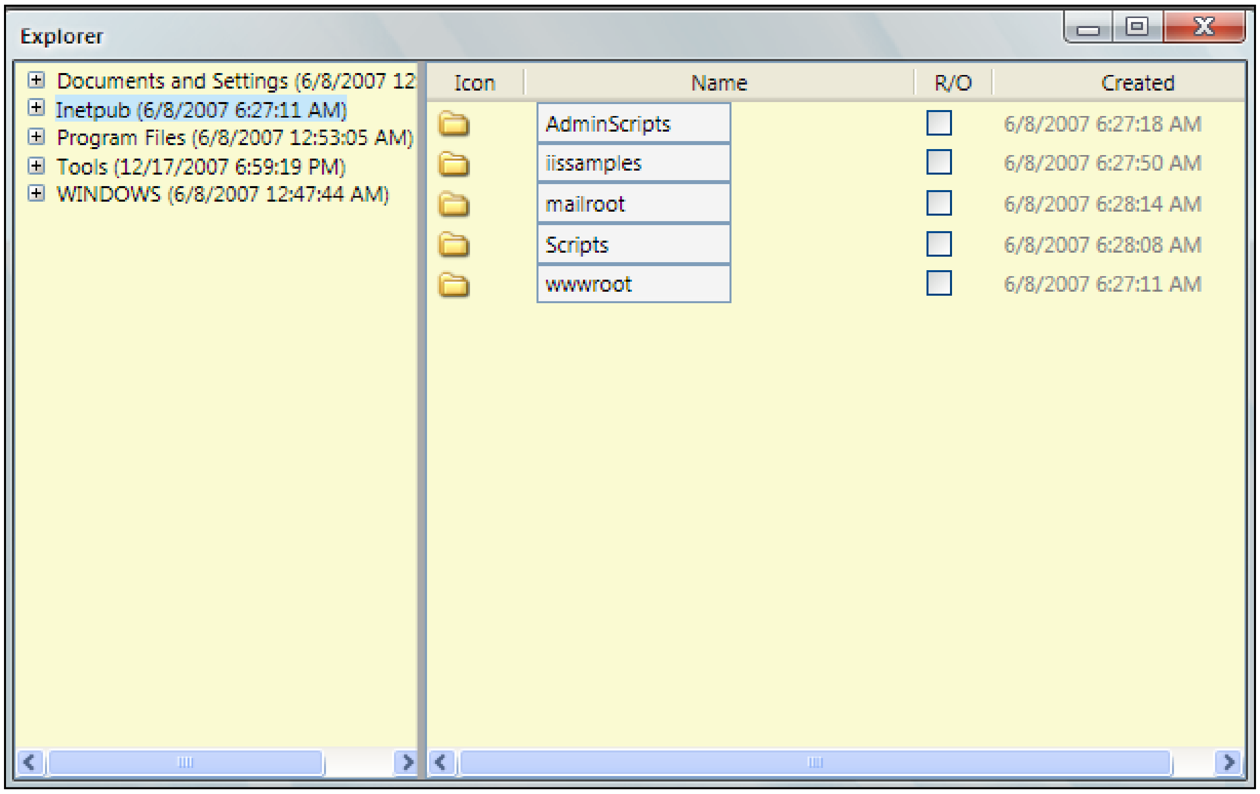
Estimated time for completion: **60 minutes**

Goals:

- Learn how to bind to collections
- Utilize Data Templates to create UI versions of underlying data

Overview:

In this lab, you'll learn how to use data bindings and data templates to create a compelling user interface very quickly. In this case, you'll write a smaller version of Windows Explorer which will look like:



Explorer

⊞ Documents and Settings (6/8/2007 12
⊞ Inetpub (6/8/2007 6:27:11 AM)
⊞ Program Files (6/8/2007 12:53:05 AM)
⊞ Tools (12/17/2007 6:59:19 PM)
⊞ WINDOWS (6/8/2007 12:47:44 AM)

Icon	Name	R/O	Created
📁	AdminScripts	☐	6/8/2007 6:27:18 AM
📁	iissamples	☐	6/8/2007 6:27:50 AM
📁	mailroot	☐	6/8/2007 6:28:14 AM
📁	Scripts	☐	6/8/2007 6:28:08 AM
📁	wwwroot	☐	6/8/2007 6:27:11 AM

The underlying data will be simple data objects and data binding and a couple of Value Converters will be used to connect these up to the UI

Part 1 – Familiarize yourself with the before solution

In this part, you'll familiarize yourself with the "before" solution. It is not very complex but you should be comfortable with it before you move on.

Steps:

1. Open and explore the "Explorer" starter application. It is located in the **before** folder associated with the lab.
2. Examine the different classes.
3. Notice it already has two classes `FileInfo` and `DirectoryInfo` which allow you to get information about files and directories. There is also an interop class used to retrieve icons in the `Win32.cs` file.
4. Compile and run the application. At this point, the interface is blank. Your job is to implement it.
5. The main window contains a grid with two controls.
   a. You can adjust the position of the separator.
   b. On the left is a tree view.
   c. On the right is a list view
6. Open the `Window1.xaml` file. Verify that you see the definition for the `ListView` and the `TreeView`.

Part 2 – Display the directories in the tree view

In this part, you'll show the tree of directories in the tree view.

Steps:

1. The first step is to define an object for the root directory. Although it would be possible to do this in code, in this case we'll use XAML exclusively.
   a. Open the `Window1.xaml` file.
   b. Locate the `Grid` element and define a `Grid.Resources` sub-element. You'll put all your resources here.
2. In order for the XAML compiler to understand the `DirectoryInfo` and `FileInfo` classes, you need to map an XML namespace to the CLR namespace.

```
xmlns:explorer="clr-namespace:Explorer"
```

3. Define a `DirectoryInfo` object as a resource.
   a. Set its `FullName` to "`c:\`" and give it a resource name like `rootDir`. Remember to use the proper XAML namespace

```
<explorer:DirectoryInfo x:Key="rootDir" FullName="c:\" />
```

4. Now that we have a root directory, turn your attention to the `TreeView`. Bind its `ItemsSource` property to the `SubDirectories` property of the `rootDir` resource.
    a. You will need to use the `{Binding}` markup extension and set its source to the `rootDir` and path to the `SubDirectories` property.

```
<TreeView ItemsSource="{Binding Source={StaticResource rootDir},
                        Path=SubDirectories}" Name="treeView" />
```

5. Compile and run. At this point, you should see a list of `DirectoryInfo` in the tree view. Verify that you have as many `DirectoryInfo` entries in the tree view as directories in "`c:\`".

Part 3 – Change the look of the directories

So far, the tree view knows which collection of directories to display but it does not know how to display each individual directory. In this part, you'll define a Data template to specify the look of directories in the tree view.

Steps:

1. In the resources of the `Grid`, define a new `DataTemplate`.
    a. Set its `DataType` to the type of `DirectoryInfo`. **Hint**: use the `x:Type` markup extension. **Hint #2**: don't forget the xml namespace for the directory type.
2. Inside the data template, define a `TextBlock` and set its `Text` property to "Hello".

```
<DataTemplate DataType="{x:Type explorer:DirectoryInfo}">
    <TextBlock Text="Hello" />
</DataTemplate>
```

3. Compile and run. Verify that the tree view now displays "Hello" for each directory.
4. Now bind the `Text` property of the `TextBlock` to the `Name` of the `DirectoryInfo`. In this instance, only the `Path` of the binding needs to be specified - the `Source` of the binding is implicit.
5. Compile and run. Verify that the tree view now displays the name of the directory.
    a. If don't see the name of the directory, you probably have a typo in your binding. Run it under the debugger and look at the Visual Studio Output window for more information.

6. Next to the `TextBlock`, add another `TextBlock` and bind it to the `CreationTime` of the directory.

```
<DataTemplate DataType=”{x:Type explorer:DirectoryInfo}”>
    <TextBlock Text=”{Binding Path=Name}” />
    <TextBlock Text=”{Binding Path=CreationTime}” />
</DataTemplate>
```

7. Compile and run. You should get an error because data templates accept only one child element. Fix this by adding a horizontal `StackPanel` containing the `TextBlocks.`

```
<DataTemplate DataType=”{x:Type explorer:DirectoryInfo}”>
    <StackPanel Orientation=”Horizontal”>
       <TextBlock Text=”{Binding Path=Name}” />
       <TextBlock Text=”{Binding Path=CreationTime}” />
    </StackPanel>
</DataTemplate>
```

8. So far, the tree view displays only one level because it does not know how to get the children of a directory. To fix this, replace the `DataTemplate` with a `HierarchicalDataTemplate` and bind the `ItemsSource` property to the `SubDirectories` property.

```
<HierarchicalDataTemplate DataType="{x:Type explorer:DirectoryInfo}"
                          ItemsSource="{Binding SubDirectories}">
  <StackPanel Orientation="Horizontal">
    <TextBlock Text="{Binding Name}" />
    <TextBlock Text=" (" />
    <TextBlock Text="{Binding CreationTime}" />
    <TextBlock Text=")" />
  </StackPanel>
</HierarchicalDataTemplate>
```

9. Compile and run. Verify that the tree view now displays the hierarchy of directories.

Part 4 – Binding the ListView

In this part, you'll bind the content of the list view with the directory selected in the tree view. Then, you'll change the look of each file or directory in the list view.

Steps:

1. Locate the `ListView`. It is currently empty. Like all `ItemsControls`, the list view exposes an `ItemsSource` property. Since we want to see the list of children of the currently selected directory, bind this property to the `Children` of the `SelectedItem` of the `TreeView`.

    a. For this, you'll need to use `ElementName` to specify the source.

2. Compile and run. Select a directory in the tree view. Verify that all its children (subdirectories and files) appear in the list view. At this point, no template has been defined for the `FileInfo` type so you should only see the string “FileInfo”.

3. The next step is to define a visual look for items in the `ListView`. `ListViews` are a little different from other Items controls because they have columns.

    a. Using an XML sub-element, define the `View` property of the `ListView` to a `GridView`. The `GridView` contains a list of `GridViewColumns`, each containing a header property and a cell template which is a `DataTemplate` used to render that column for each row.

```
<ListView Grid.Column="1"
    ItemsSource="{Binding ElementName=treeView, Path=SelectedItem.Children}">
  <ListView.View>
    <GridView>
      <GridViewColumn Header=”title” Width=”200”>
         <GridViewColumn.CellTemplate>
           <DataTemplate>
                 ...
           </DataTemplate>
         </GridViewColumn.CellTemplate>
      </GridViewColumn>
    </GridView>
  </ListView.View>
</ListView>
```

    b. Add four `GridViewColumns` for: “Icon” "Name", "Is Read Only" and "Creation Time".

    c. For each column, set the `Header` property appropriately and define a `GridViewColumn.CellTemplate` as the child of the column to define what it should look like.

    d. For the icon, use an `Image`. Bind the `Source` property to the file icon. Also, set the `Width` and `Height` of the image to 16.

    e. For the name, use a `TextBox` so that we can change the name. Bind the `Text` property.

    f. For the creation time, use a `TextBlock` and bind its `Text` property.

    g. For the read only property, use a check box and bind the `IsChecked` property.

    h. Set the `Width` of each `GridViewColumn` to an appropriate value so that they are visible. The lab sample will use 200, 40 and 150.

```
<ListView Grid.Column="1"
```

```
    ItemsSource="{Binding ElementName=treeView, Path=SelectedItem.Children}">
  <ListView.View>
    <GridView>

      <GridViewColumn Header="Icon" Width="50">
        <GridViewColumn.CellTemplate>
          <DataTemplate>
            <Image Source="{Binding Icon}" Width="16" Height="16" />
          </DataTemplate>
        </GridViewColumn.CellTemplate>
      </GridViewColumn>

      <GridViewColumn Header="Name" Width="200">
        <GridViewColumn.CellTemplate>
          <DataTemplate>
            <TextBox Text="{Binding Name}" />
          </DataTemplate>
        </GridViewColumn.CellTemplate>
      </GridViewColumn>

      <GridViewColumn Header="R/O" Width="40">
        <GridViewColumn.CellTemplate>
          <DataTemplate>
            <CheckBox IsChecked="{Binding IsReadOnly}" />
          </DataTemplate>
        </GridViewColumn.CellTemplate>
      </GridViewColumn>

      <GridViewColumn Header="Created" Width="150">
        <GridViewColumn.CellTemplate>
          <DataTemplate>
            <TextBlock Text="{Binding CreationTime}" />
          </DataTemplate>
        </GridViewColumn.CellTemplate>
      </GridViewColumn>

    </GridView>
  </ListView.View>
</ListView>
```

4. Compile and run. Verify that files are now visible in the `ListView`. If not, there is probably a typo in your binding - look at the Visual Studio Output window for guidance.
5. Select a file and change the name using the text box. Verify that the underlying file has been renamed.

Part 5 – Using a Type Converter

In this part, you'll write two converters to enhance the user experience. The first converter will allow you to disable editing for read only files. The second one will allow you to display files with different colors depending on the creation time of the file.

Steps:

1. Right now, the `TextBox` is always enabled. However, we'd like to disable it for read-only files. Locate the `TextBox` in the data template and bind the `IsEnabled` property to the `IsReadOnly` property.
2. Compile and run. Verify that we have the exact opposite of what we want: read only files allow for editing while writable files do not.
3. To fix this, create a new class called `NotConverter`.
    a. Implement the `IValueConverter` interface. `IValueConverter` has two methods: `Convert` and `ConvertBack`. You won't need to implement `ConvertBack` because we'll use a one way binding.
    b. In the `Convert` method, convert the value parameter to a Boolean and return the opposite.
4. Now that we have a converter, define an instance as a resource of the grid and give it a resource key like “notConverter”.
5. Go back to the `IsEnabled` property.
    a. In its binding, set the converter to the static resource you just defined. At this point, your binding should look like this:

```
IsEnabled="{Binding IsReadOnly, Converter={StaticResource notConverter}}"
```

6. Compile and run. Verify that only writable files are editable.
7. The next step is to display files with different colors depending on their creation time.
    a. For example, you can display file created today in blue and older files in gray. This is just an example; you may try something more sophisticated.
8. To accomplish this, create a new converter which converts a `Date` to a `Brush`.
9. Define an instance of this converter as a resource.
10. Bind the `Foreground` of the `TextBlock` that displays the creation time using this converter. Allow for the colors to be set in the resource rather than be hard coded.
11. Run the application to ensure it runs properly.

As a last step, try expanding the TreeView on a large directory such as your **C:\Windows** folder. Notice how slow it is? Can you think of an easy way to fix this?

Hint: there is a simple property you can add to the binding that will make it run on a different thread – refer to the DataBinding 1 slides if you need some help.

Hint #2: think carefully about *which* binding to apply this to – see the solution if you need some help.

Solutions

The **after** folder has a complete solution for this lab.