DataBinding Basics

Estimated time for completion: 45 minutes

Goals:

- Binding controls together
- Binding `ContentControls` to data
- Create binding-aware classes in .NET

Overview:

This lab is designed to familiarize the student with basic data binding techniques. In many cases data binding obviates the need for code behind and so this lab will do most of its work in XAML with no procedural code necessary.

Part 1 – Binding Controls together

In this part, we will utilize the data binding capabilities to link multiple controls together.

Steps:

1. Create a new WPF project in Visual Studio.
2. Add two columns to the `Grid` layout root.
3. Make the first column size to content.

```
<Grid>
  <Grid.ColumnDefinitions>
     <ColumnDefinition Width="Auto" />
     <ColumnDefinition />
  </Grid.ColumnDefinitions>
</Grid>
```

4. In the first column, place a vertical slider.
    a. Give it a name; the sample code will use “slider1”.
    b. Align it to the bottom
    c. Set the `Height` to be “100”.
    d. Set the range to be “0” – “100”.

```
<Slider VerticalAlignment="Bottom"
        Name="slider1" Orientation="Vertical"
        Minimum="0" Maximum="100" Height="100" />
```

5. In the second column, place a `Rectangle`.
6. Set the fill color to be a brush; any color will do – the sample will use a gradient brush which looks somewhat like the Vista progress bar.
7. Set the `Width` to be "20" and the `Height` to be "100".

```
<Rectangle VerticalAlignment="Bottom"
           HorizontalAlignment="Center"
           Stroke="LightGray" StrokeThickness="2"
           Grid.Column="1" Width="20" Height="100">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
          <GradientStop Color="DarkGreen" Offset="0" />
          <GradientStop Color="White" Offset=".5" />
          <GradientStop Color="LightGreen" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
```

8. Right now, these two elements are disconnected. The goal is to bind the height of the rectangle to the current value of the slider.
9. Change the `Height` value to use a `{Binding}` expression.
10. You will need to set the `ElementName` property to the name of the slider and the `Path` property to the `Slider.Value` property.

```
<Rectangle VerticalAlignment="Bottom"
           HorizontalAlignment="Center"
           Stroke="LightGray" StrokeThickness="2"
           Grid.Column="1" Width="20"
           Height="{Binding ElementName=slider1, Path=Value}">
```

11. Move the slider. You should see the rectangle changing height dynamically with the slider value.

Part 2 – Two way bindings

In this part, we will force two-way bindings to ensure that when either value changes, both values change.

Steps:

1. Add a second row to the grid. The second row should size to the content.

```
  <Grid.RowDefinitions>
     <RowDefinition />
     <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
```

2. In the second row, place a `TextBox`.
3. Have the `TextBox` span both columns.
4. Bind the `Text` property to the `Height` of the `Rectangle`.
5. **Hint:** you will need to name the rectangle to do this.

```
<TextBox MinWidth="200" Grid.ColumnSpan="2" Grid.Row="1"
      Text="{Binding ElementName=rc1, Path=Height}" />
```

6. Change the slider value – notice how the `TextBox` changes with it. This is because of the data binding being applied when the height of the rectangle is changed.
7. Now, type a new value into the `TextBox` and press TAB (or click on the slider to remove focus). Notice that the rectangle changes, but the slider does not. This is because the default binding from the `Slider` to the `Rectangle.Height` is a One-Way binding. Our goal is to change that.
8. Add onto the `Rectangle` binding a `Mode` value indicating two-way binding.

```
<Rectangle Name="rc1" VerticalAlignment="Bottom"
           HorizontalAlignment="Center"
           Stroke="LightGray" StrokeThickness="2"
           Grid.Column="1" Width="20"
           Height="{Binding ElementName=slider1, Path=Value, Mode=TwoWay}">
```

9. Now, type a new value into the `TextBox` and press TAB. Both the slider *and* rectangle should change values now.

Part 3 – Changing when updates occur

In this part, we will change the behavior of the binding to occur immediately when the value changes by manipulating the binding properties.

Steps:

1. Notice how you must press TAB or move focus out of the `TextBox` in order to get the value applied. Our goal in this part is to change that behavior to be automatic.
2. On the `TextBox` binding, change the `UpdateSourceTrigger` property to change when the property changes.

```
  <TextBox MinWidth="200" Grid.ColumnSpan="2" Grid.Row="1"
      Text="{Binding ElementName=rc1, Path=Height,
```

```
            UpdateSourceTrigger=PropertyChanged}" />
```

3. Now change the `TextBox` – the slider and rectangle should update immediately.

Solution

```
<Grid>
  <Grid.ColumnDefinitions>
     <ColumnDefinition />
     <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
     <RowDefinition />
     <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>


  <Slider VerticalAlignment="Bottom" Name="slider1"
          Orientation="Vertical" Minimum="0" Maximum="100" Height="100" />
  <Rectangle Name="rc1" VerticalAlignment="Bottom"
             HorizontalAlignment="Center"
             Stroke="LightGray" StrokeThickness="2"
             Grid.Column="1" Width="20"
             Height="{Binding ElementName=slider1, Path=Value, Mode=TwoWay}">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
          <GradientStop Color="DarkGreen" Offset="0" />
          <GradientStop Color="White" Offset=".5" />
          <GradientStop Color="LightGreen" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>

  <TextBox MinWidth="200" Grid.ColumnSpan="2" Grid.Row="1"
           Text="{Binding ElementName=rc1, Path=Height,
                  UpdateSourceTrigger=PropertyChanged}" />
</Grid>
```

Part 4 – Using a Binding Source

In this part, we will bind to other types of data beyond controls. This requires a slightly different binding syntax – we no longer specify ElementName, but instead provide a specific Source.

Note: to do the next two parts you will need admin rights on your machine. If you do not have admin rights, then skip to part 6.

Steps:

1. Create a new WPF project in Visual Studio.
2. Change the root panel from a `Grid` to a `StackPanel`.
3. Create a resources section in the `StackPanel`. This is where we will define some global data for our XAML to utilize.

```
<StackPanel>
    <StackPanel.Resources>
    </StackPanel.Resources>
</StackPanel>
```

4. Add into the resources a new `EventLog` object – this is contained in the `System.Diagnostics` namespace and is part of `System.dll`.
5. Hint: you will need to bind the CLR namespace to a XAML namespace to create this object.
6. Set the `Log` property to "Application" and the `MachineName` property to ".".
    a. **Note:** Visual Studio may give a warning indicating that the `Log` property cannot be set but this is a warning that can be ignored.

```
<d:EventLog xmlns:d="clr-namespace:System.Diagnostics;assembly=System"
    x:Key="evtLog" Log="Application" MachineName="." />
```

7. Add four `TextBlock` objects to the `StackPanel` and use data binding to connect them up to the first event entry in the `EventLog.Entries` collection. In this case, you will need to specific a `Source` for the binding expression.
8. Bind the text blocks to the following properties of the event entry:
9. `TimeGenerated` property
10. `Source` property
11. `EntryType` property
12. `Message` property

```
<TextBlock FontSize="16pt"
     Text="{Binding Source={StaticResource evtLog},
            Path=Entries[0].TimeGenerated}" />
<TextBlock FontSize="16pt"
     Text="{Binding Source={StaticResource evtLog},
            Path=Entries[0].Source}" />
<TextBlock FontSize="16pt"
     Text="{Binding Source={StaticResource evtLog},
```

```
            Path=Entries[0].EntryType}" />
<TextBlock FontSize="16pt"
     Text="{Binding Source={StaticResource evtLog},
     Path=Entries[0].Message}" />
```

13. The text should be displaying a single entry from the event log – notice how you can index into the `Entries` collection. Try experimenting with different properties to see what you can bind to.
14. Finally, change one of the `TextBlocks` into a `TextBox` and note the error. See if you can fix it by altering the binding.
15. **Hint:** think about the default binding mode for the `TextBox` vs. the `TextBlock`.

Part 5 – Using a DataContext source

In the previous part, we bound several text blocks to a specific data source. In this part we will condense the code slightly by providing a DataContext for the text blocks.

Steps:

1. Move the entire resources block from the `StackPanel` to a higher level (typically the `Window` object).

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Window.Resources>
      <d:EventLog xmlns:d="clr-namespace:System.Diagnostics;assembly=System"
          x:Key="evtLog" Log="Application" MachineName="." />
    </Window.Resources>
    <StackPanel>
        …
    </StackPanel>
</Window>
```

2. Notice the code continues to work – this is because resources are inherited along the XAML element tree and all we've done is move them up a level.
3. Set the `DataContext` property for the `StackPanel` to be bound to the event log. The binding expression will be identical to the one used on a `TextBlock` excluding the Path.

```
<StackPanel DataContext="{Binding Source={StaticResource evtLog}}">
```

4. Remove the Source off each of the TextBlocks – notice how they continue to have the same value. This is due to the DataContext – it is providing a *default* binding source for anything at that level or below. You can, of course, override it and specify a source – it is only used when no source is provided.

```
<TextBlock FontSize="16pt" Text="{Binding Path=Entries[0].TimeGenerated}" />
<TextBlock FontSize="16pt" Text="{Binding Path=Entries[0].Source}" />
<TextBlock FontSize="16pt" Text="{Binding Path=Entries[0].EntryType}" />
<TextBlock FontSize="16pt" Text="{Binding Path=Entries[0].Message}" />
```

5. Also note that you can omit Path= from the definition, the binding markup extension has a constructor that takes a path – this allows a simplified syntax in the case where the default data source is used.

```
<TextBlock FontSize="16pt" Text="{Binding Entries[0].TimeGenerated}" />
<TextBlock FontSize="16pt" Text="{Binding Entries[0].Source}" />
<TextBlock FontSize="16pt" Text="{Binding Entries[0].EntryType}" />
<TextBlock FontSize="16pt" Text="{Binding Entries[0].Message}" />
```

6. Finally, move the EventLog instance object directly into the DataContext property – instead of using a binding expression, have XAML create and assign the property directly through the property element syntax.

```
<StackPanel.DataContext>
    <d:EventLog xmlns:d="clr-namespace:System.Diagnostics;assembly=System"
        Log="Application" MachineName="." />
</StackPanel.DataContext>
```

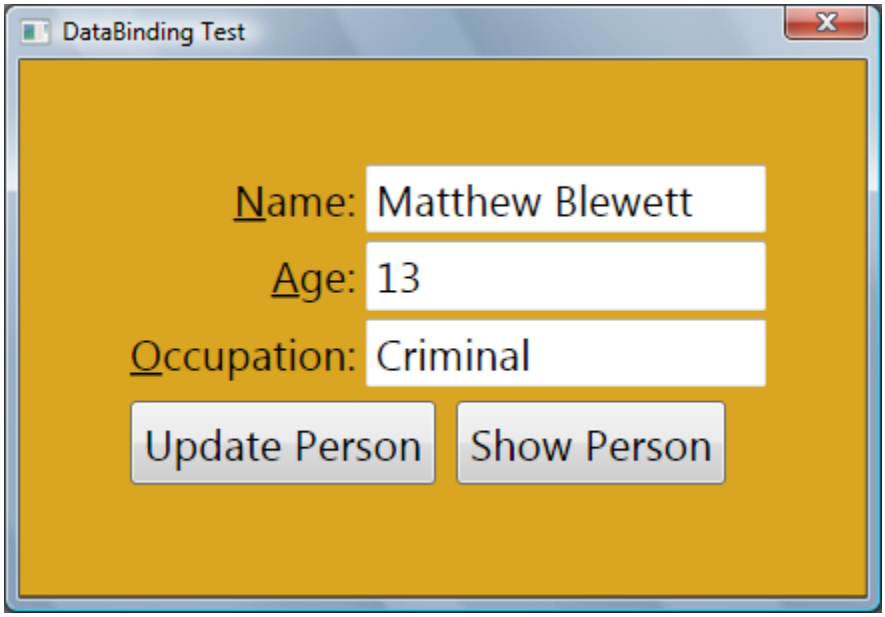
Solution

```
<StackPanel>
   <StackPanel.DataContext>
      <d:EventLog Log="Application" MachineName="."
         xmlns:d="clr-namespace:System.Diagnostics;assembly=System" />
   </StackPanel.DataContext>

   <TextBlock FontSize="16pt" Text="{Binding Entries[0].TimeGenerated}" />
   <TextBlock FontSize="16pt" Text="{Binding Entries[0].Source}" />
   <TextBlock FontSize="16pt" Text="{Binding Entries[0].EntryType}" />
   <TextBlock FontSize="16pt" Text="{Binding Entries[0].Message}" />
</StackPanel>
```

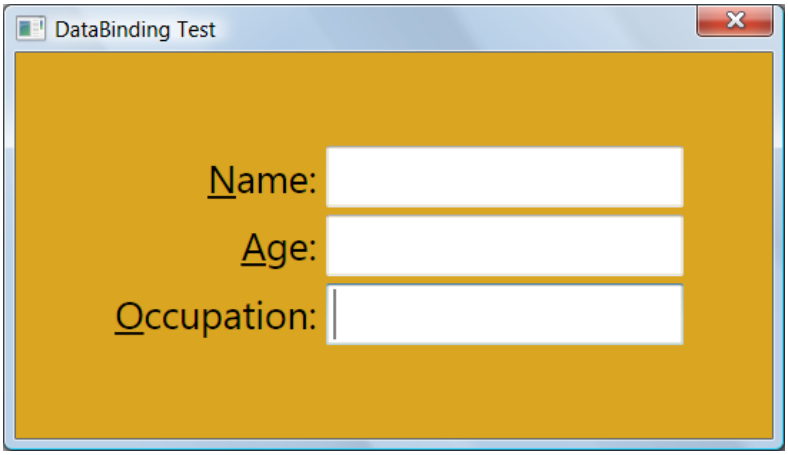
Part 6 – Binding to .NET objects

In this part, we will data bind to custom objects created in procedural code and learn how to design components to be data binding aware without restricting the object usage to WPF. When you are finished, the application window will look like:



Steps:

1. Open the **DataBinding.sln** starter project in the **before** folder for this lab.
2. Examine the existing code – there is a `Person.cs` file which contains a `Person` object and the standard `Window1.xaml` and `Window1.xaml.cs`.
3. As a first step, open the `Window1.xaml` file and add two columns and three rows to the `Grid`. The goal is to display the three properties of the person in the grid.
4. In each of the rows add a `Label` and `TextBox`. The `Label` should be in the first column and the `TextBox` in the second. The labels should be named:
    a. Name
    b. Age
    c. Occupation
5. The goal is to look something like:



6. Switch to the code behind file and add a new field which is of type `Person`. Go ahead and assign it to a new `Person` object. The default constructor fills the object with random data.
7. In the constructor of the `Window`, assign the `DataContext` of the window to the newly created `Person` object – you should do this prior to calling `InitializeComponent` so that it is available when the controls are created.

```
public partial class Window1 : Window
{
   Person _person = new Person();

   public Window1()
   {
      DataContext = _person;
      InitializeComponent();
   }
   ...
}
```

8. Next, use Data Binding to associate the `TextBox` elements' `Text` property to the three properties exposed by the `Person` object – you won't need a data source because the Data Context has been setup, so just specify the path for each property. An example is:

```
<TextBox Grid.Column="1"
         HorizontalAlignment="Left" VerticalAlignment="Center" MinWidth="200"
         Text="{Binding Name}"/>
```

9. Run the application. It should display the current person in the three text fields.
10. The next step is to check the two-way binding. Add a new row to the grid and in the row place a button titled "Show Person".
11. Attach a `Click` event handler to the button and in the handler display the `Person` object in a `MessageBox`.

```
void ShowPersonClicked(object sender, RoutedEventArgs e)
{
   MessageBox.Show(_person.ToString());
}
```

12. Run the application again and make a change in one of the `TextBox` fields. Click the button – the `MessageBox` should show the current field data. This means our data binding is working the way we expect – when a change is made to the UI, the underlying data is being changed.
13. Next, add another button titled "Update Person" to the fourth row in the window – you can use a `StackPanel` or your favorite layout panel to position them.

```
<StackPanel Grid.ColumnSpan="2" Grid.Row="3" Orientation="Horizontal">
   <Button Margin="5" Padding="5">
      Update Person
   </Button>
   <Button Margin="5" Padding="5" Click="ShowPersonClicked">
      Show Person
   </Button>
</StackPanel>
```

14. Attach a `Click` event handler to the new button and in the handler, generate new person data using the `Person.GenerateRandomPerson()` method. Go ahead and display the `Person` object in a `MessageBox` after it has been changed.

```
void PersonChangeClicked(object sender, RoutedEventArgs e)
{
   _person.GenerateRandomPerson();
   ShowPersonClicked(null, null);
}
```

15. Run the application and click the "Update Person" button. The `MessageBox` should display a new person – but the `TextBox` fields are not changing!
16. Change a `TextBox` field and click the "Show Person" button – you should see a mismatch of data now – some fields are correct (specifically, the one you just changed), but some are no longer synchronized.

    The problem is that WPF doesn't know the underlying object has changed – it has no knowledge of you changing the fields in the person object.

17. In order to fix this issue, we need to implement the `INotifyPropertyChanged` interface. This will allow WPF to *see* changes being made to the Person object.
18. Open the `Person.cs` file.
19. Implement the `System.ComponentModel.INotifyPropertyChanged` interface on the Person class. It consists of a single event named `PropertyChanged`.

```
public class Person : INotifyPropertyChanged
{
   #region INotifyPropertyChanged Members
   public event PropertyChangedEventHandler PropertyChanged;
   #endregion
   ...
}
```

20. Next, change all the auto-property definitions for the public properties and replace them with regular getter/setter implementations backed with fields.
21. In each of the setters for the properties of the person and raise the `PropertyChanged` event as the last step in the setter. The event expects a sender and a `PropertyChangedEventArgs` which specifies the property name that has changed. An example for the name field would be:

```
public string Name
{
   get { return _name; }
   set
   {
      _name = value;
      if (PropertyChanged != null)
         PropertyChanged(this, new PropertyChangedEventArgs("Name"));
   }
}
```

22. For reuse, you should create a private **OnPropertyChanged** method which does this work for you – passing in the property name. Check the slides for an example of this method.
23. Once you have updated each property, run the application again and click the "Update Person" button. You should now see the data changing immediately in the `TextBox` fields. This is happening because WPF attaches a handler to the `PropertyChanged` event when we data bind to this object. It is then notified when any data changes and makes the appropriate changes to anything bound to the field.

Solution

There is a full implementation of this final part in the **after** folder associated with the lab – remember there are separate projects for VS2008 and VS2010.