Data Binding

Estimated time for completion: 60 minutes

Overview:

In this lab you will learn how to bind data for presentation to the user interface. Data binding helps to minimize the amount of code that you have to write in order to produce a RIA.

Goals:

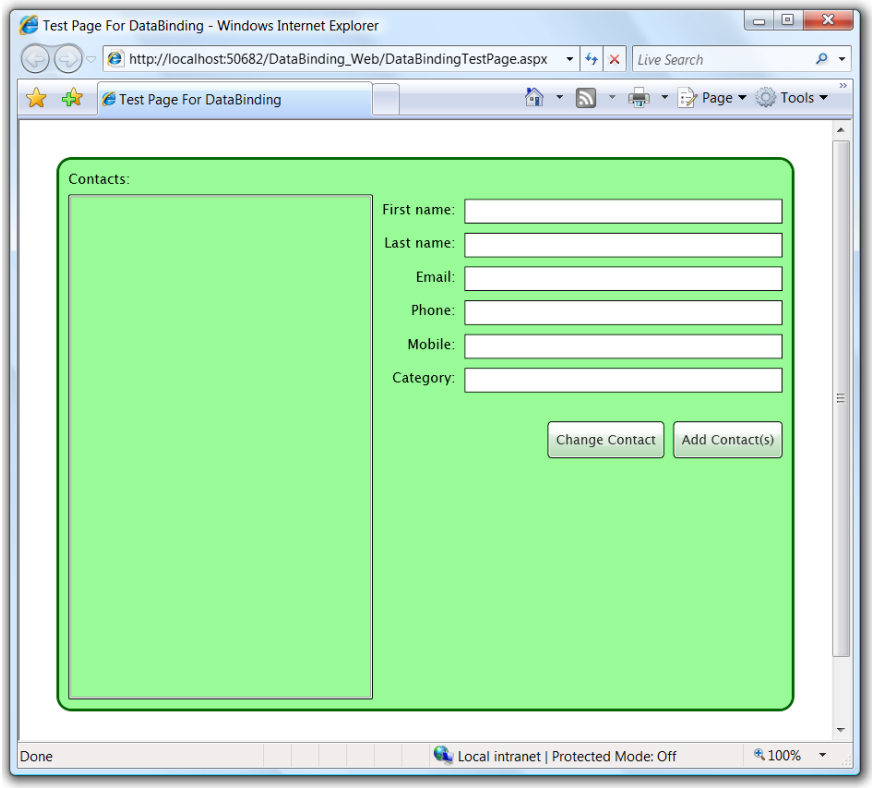
- Bind to a simple CLR object
- Understand change propagation and the `INotifyPropertyChanged` interface
- Examine two-way bindings
- Binding to collections
- Building a master / details view using element binding

Binding to a simple CLR object

Silverlight has interesting data binding capabilities. In the first part of the lab you will learn how to bind the properties of a CLR object to user interface elements.

Simple data binding

1. Open the project .\work\DataBinding\before\DataBinding.sln. Build and run the application. You will see a basic contact display application appear, as shown below:



   In this lab you will provide functionality to bind information from a `Contact` object into this user interface. Close the application now.

2. Open the file Contact.cs. You will see the `Contact` type that will be bound, along with a supporting factory class that can be used to create `Contact` objects.

   The most important facet of the `Contact` class is that it exposes accessible properties, which is a requirement for it to be bound in Silverlight.

> Note that this lab doesn't require the use of a database, as Silverlight applications themselves have no direct access to a database. Instead, data will typically be retrieved using network calls, with the returned objects being bound into the UI.
>
> These network calls are simulated using the ContactGenerator class.

3. Open MainPage.xaml.cs and add an event handler for the btnAddContact's `Click` event. Inside this event handler, add code to create a `Contact` object using the `ContactGenerator.GenerateContact` method.

   Store the returned reference in the **`DataContext`** property of the `MainPage` user control, as Silverlight bindings walk the tree to find the data to which they should be bound.

   Your code might look like this:

```
public MainPage()
{
    InitializeComponent();
    btnAddContact.Click += new RoutedEventHandler(btnAddContact_Click);
}

void btnAddContact_Click(object sender, RoutedEventArgs e)
{
    Contact c = ContactGenerator.GenerateContact();
    DataContext = c;
}
```

4. Open MainPage.xaml and set the `Text` property of each element using a `Binding` extension to bind in the corresponding property from the `Contact` object. Your code might look like this:

```
<TextBox ... x:Name="txtFirstName"
                        Text="{Binding FirstName, Mode=TwoWay}" />
<TextBox ... x:Name="txtLastName"
                        Text="{Binding LastName, Mode=TwoWay}"/>
<TextBox ... x:Name="txtEmail"
                        Text="{Binding EmailAddress, Mode=TwoWay}"/>
<TextBox ... x:Name="txtPhone"
                        Text="{Binding PhoneNumber, Mode=TwoWay}" />
<TextBox ... x:Name="txtMobile"
                        Text="{Binding MobileNumber, Mode=TwoWay}" />
<TextBox ... x:Name="txtCategory"
                        Text="{Binding Category, Mode=TwoWay}" />
```

5. Build and run the application. You will now see contact information when you click the *Add Contact(s)* button. If you click the *Add Contact(s)* button multiple times then you'll see that the `TextBox` elements will automatically update their content.

Using a binding converter

Sometimes there is a requirement to change the type of data that comes from the bound element to match a requirement in the UI. In this section of the lab you will write a converter that maps the Contact Category property to a brush.

Implementing a binding converter

1. Add a new class to your project, naming it **CategoryToBrushConverter**.
2. Now implement the **System.Windows.Data.IValueConverter** interface in your class. Inside your implementation of the `Convert` method, return a differently colored `SolidColorBrush` depending on the type of `ContactCategory` that is passed in via the value parameter. Your code might look something like this:

```
using System.Windows.Data;

public class CategoryToBrushConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        ContactCategory category = (ContactCategory) value;
        Color c = Colors.Black;

        switch (category)
        {
            case ContactCategory.Friend: c = Colors.Green; break;
            case ContactCategory.Family: c = Colors.Blue; break;
            case ContactCategory.Colleague: c = Colors.Orange; break;
            case ContactCategory.Customer: c = Colors.Black; break;
            case ContactCategory.MedicalPractitioner:
                  c = Colors.Red; break;
        }

        return new SolidColorBrush(c);
    }

    public object ConvertBack(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

3. You now need to use your converter, so open MainPage.xaml. Since your converter class lies within the project's `DataBinding` namespace, you need to add a new namespace mapping at the top of the file so that you can use the type, as shown below:

```
<UserControl x:Class="DataBinding.MainPage"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:my="clr-namespace:DataBinding"
    ... /
```

4. Now add an instance of the converter to the user control's `Resources` dictionary. Specify a `Key` of **cnvCat**.

```
<UserControl.Resources>
    <my:CategoryToBrushConverter x:Key="cnvCat" />
</UserControl.Resources>
```

5. Now bind the **`Foreground`** property of the `txtCategory` element to the `Category` of the `Contact` object, using your converter in order to retrieve the appropriate brush.

```
<TextBox ... x:Name="txtCategory" Text="{Binding Category}"
  Foreground="{Binding Category, Converter={StaticResource cnvCat}}" />
```

6. Build and run your application. Note that the different categories will be displayed in different colors.

Whilst it can be fun (and straightforward and a great learning experience) to use a converter to select colors for display purposes, it has two distinct problems.

The first is that it limits the designer's capabilities in altering the look and feel of the application. If you intend to allow a color to be used this way, then expose a property from your converter so that it can be set by the designer.

The second is that it fails users who cannot distinguish colors well.

You should therefore use the selection of color within the application with caution.

Understanding change notification

Bindings will automatically pick up changes when the DataContext itself is changed. However, they will not pick up changes to the object that is referenced by the DataContext property unless the object's type implements a specific interface: INotifyPropertyChanged. You will investigate this interface now.

Implementing INotifyPropertyChanged

1. In MainPage.xaml.cs, add a `Click` event handler for the `btnChangeContact` button. In this event handler, extract the `Contact` object from the `DataContext` and change the `FirstName` property to a new value by calling the `ContactGenerator.GetRandomFirstName` method, as shown below:

```
public MainPage()
{
    InitializeComponent();
    btnAddContact.Click += btnAddContact_Click;
    btnChangeContact.Click += btnChangeContact_Click;
}

void btnChangeContact_Click(object sender, RoutedEventArgs e)
{
    Contact c = DataContext as Contact;
    if (c != null)
    {
        c.FirstName = ContactGenerator.GetRandomFirstName(c.FirstName);
    }
}
```

2. Build and run the application. Create a new `Contact` and then click the *Change Contact* button a few times. Note that the changed first name is **not** reflected in the user interface. Close the application.

3. Open up the Contact.cs file and implement the `INotifyPropertyChanged` interface (from the `System.ComponentModel` namespace) on the `Contact` type. To implement the interface, you should raise the `PropertyChanged` event whenever the value of one of the properties is changed.

> For the purpose of this lab, feel free to do this only for the FirstName property to save some time. Note, however, that you should raise the event for ALL properties that are bound and which you want to support notification to the UI if the property is changed by, say, a background operation.
>
> In general, you might well want to consider creating a base class for all entity types that will implement INotifyPropertyChanged.

4. Your code might look like this:

```
public class Contact : INotifyPropertyChanged
{
      string firstName;
      public string FirstName
      {
            get { return firstName; }
            set
            {
                  firstName = value;
                  OnPropertyChanged("FirstName");
            }
      }

      // other properties elided for brevity
      ...

      protected virtual void OnPropertyChanged( string propName )
      {
            if( PropertyChanged != null )
                  PropertyChanged( this,
                        new PropertyChangedEventArgs( propName ) );
      }

      public event PropertyChangedEventHandler PropertyChanged;
}
```

5. Build and run the application. Create a new `Contact` and then click the *Change Contact* button. Note that the UI immediately reflects the changes to the underlying object. This is a considerable advantage when extracting data from the server and using it to update a bound object.

Binding to collections

Many applications allow users to display a list of items so that they can choose something to work with: Silverlight fully supports binding to collections of data, as you are about to see.

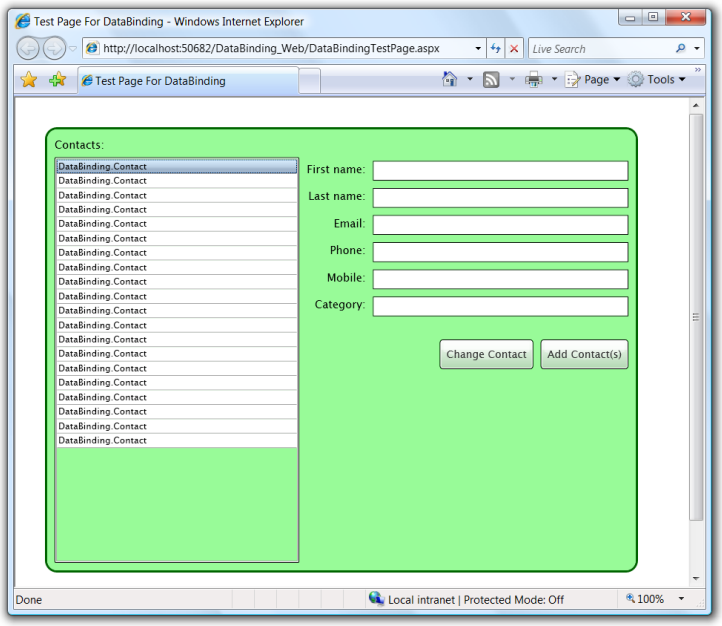
Working with collections

1. Open MainPage.xaml.cs and locate the `btnAddContact_Click` method. Modify this method so that it sets the `DataContext` of the page to the result of a call to the `ContactGenerator.GenerateContactList` method.

   Pass in a reasonably sized value (say 20) as the parameter.

```
void btnAddContact_Click(object sender, RoutedEventArgs e)
{
    List<Contact> contacts = ContactGenerator.GenerateContactList(20);
    DataContext = contacts;
}
```

2. Now flip into MainPage.xaml and set the `ItemsSource` property of the lstContacts `ListBox` (it's the only `ListBox` in the page) to the expression **`"{Binding}"`**. This will bind the `ItemsSource` collection to the list that is referenced by the `DataContext` property.
3. Build and run the application. The following application should appear:



   There are two things to note: the first is that Silverlight is using the type name as the visual element to display in the `ListBox`; the second is that the other controls have lost their content.

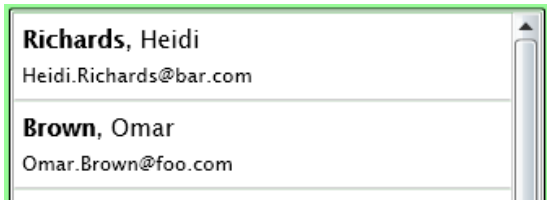
Changing the appearance of items using DataTemplates

1. Let's fix the first problem, and change how Silverlight displays the `Contact` object. Start by adding an override of the `ToString` method to the `Contact` class so that it returns the name as "Lastname, Firstname", as shown below.

```
public override string ToString()
{
    return LastName + ", " + FirstName;
}
```

2. Now build and run the application, and observe how the items are displayed in a more interesting fashion. This highlights the fact that Silverlight calls `ToString()` on an object when it needs a visual representation for it. You should rarely take this approach, as you have hard-coded visual appearance deep into the code of the class. Instead, you should use a `DataTemplate` to map the type to a visual tree.

3. Open MainPage.xaml and locate the lstContacts `ListBox`. Using the complex property setting syntax, add a `DataTemplate` for the `ListBox`'s `ItemTemplate` property. Using your experience of data binding and the layout panels, change the visual appearance for the `ListBoxItem` so that it looks as follows:



If you get stuck, take a look at the code below that shows you one sample template.

```
<ListBox ... ItemsSource="{Binding}" >
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="5">
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding LastName}"
                                                FontWeight="Bold" />
                    <TextBlock Text=", " />
                    <TextBlock Text="{Binding FirstName}" />
                </StackPanel>
                <TextBlock FontSize="12" Margin="0,2,0,0"
                                        Text="{Binding EmailAddress}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

4. Build and run the application. Your list of contacts should now look pretty.

Building master / detail applications

Master / detail applications are relatively easy to create, as you can simply bind the DataContext of the "detail" panel to the item of interest from the "master". You'll do this now.

1. Currently, the "details" controls are not displaying any information. To fix this, add an element binding that sets the `DataContext` of the `gridDetails` panel (which is the container for all of the details controls) to the currently selected item in the list.

   Your code might look like this:

```
<Grid x:Name="gridDetails"
  DataContext="{Binding ElementName=lstContacts, Path=SelectedItem}"
  ... >
```

2. Build and run the application. Now, when you select an item from the `ListBox` the details controls will show their correct information.

When resolving a Binding, the element traverses the tree upwards until it finds the first non-null DataContext property on a parent. Placing all of the "details" controls in a container makes it very easy to create master / details views, as only a single property needs to be changed.

On a final note, you should use the ObservableCollection<T> on your binding types if you want to receive change notifications whenever an item is added or removed from the list. The sweet spot of binding is therefore to use an ObservableCollection<T>, where T implements INotifyPropertyChanged.

[Optional] Examination of validation

Silverlight has some capabilities for validating user input. In this section of the exercise you will modify the Contact class to support validation for the FirstName property.

Simple validation

1. The first thing that you need to do is update the **`Contact`** class to support validation. In this case, you will only work with one property, the `FirstName` property.

   Modify the **`FirstName`** property so that it throws an exception if it is passed a null or empty string. Also decorate it with a `System.ComponentModel.DataAnnotations.Display` attribute, setting its Description to indicate what the property is used for.

   Your code might look like this:

```
[Display(Description="The first name of the contact")]
public string FirstName
{
  get { return firstName; }
  set
  {
    if (string.IsNullOrEmpty(value))
      throw new Exception("You must specify a first name");
    firstName = value;
    OnPropertyChanged("FirstName");
  }
}
```

2. Now open the MainPage.xaml file and modify the `txtFirstName's Text` binding so that it validates the control when an exception is thrown, as shown:

```
<TextBox Grid.Row="0" Margin="5" Grid.Column="1" x:Name="txtFirstName"
 Text="{Binding FirstName, Mode=TwoWay, ValidatesOnExceptions=True}" />
```

3. Build and run the application.
4. Click on the Add Contact(s) button to create a new list of contacts, and then select one from the list.
5. Clear the content of the FirstName `TextBox`.
6. Tab to the next control. The setter will be called and the exception will be thrown. A dialog box might appear in the IDE, informing you about the exception. Dismiss it and click the Continue button (or Debug -> Continue, or press F5) to continue execution.

7. The `TextBox` will display its invalid state, as shown below:

8. Now update the binding so that it raises the `BindingValidationError` event, by setting **`NotifyOnValidationError`** to `True`.
9. In MainPage.xaml.cs, add an event handler for the MainPage user control's BindingValidationError event. In the handler, display a MessageBox that shows the error if the error is being added.

   Your code might look like this:

```
void MainPage_BindingValidationError(object sender,
                                     ValidationErrorEventArgs e)
{
  if( e.Action == ValidationErrorEventAction.Added )
    MessageBox.Show( e.Error.ErrorContent.ToString(), "Data Binding",
    MessageBoxButton.OK );
}
```

10. Build and run the application. Click the AddContact(s) button, select a contact from the list and then delete the first name. Tab out of the control. The `MessageBox` will appear.
11. Close the application.

Binding validation is a complex topic. This initial taster shows you some of the capabilities of the binding mechanism in Silverlight, but there is much to consider. For example, these validation techniques tend to be focus driven, but you cannot guarantee that the user will always either (a) set focus into all the elements in the UI and (b) that the object can be validated this way.

The good news is that controls such as the DataGrid and DataForm are deeply integrated into this validation mechanism, making them very useful for many applications.

You should therefore ensure that real validation is performed in response to a more positive action from the user, such as clicking a Submit button.

Solutions

.\work\DataBinding\after\DataBinding.sln