* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Data Binding

**Contents**

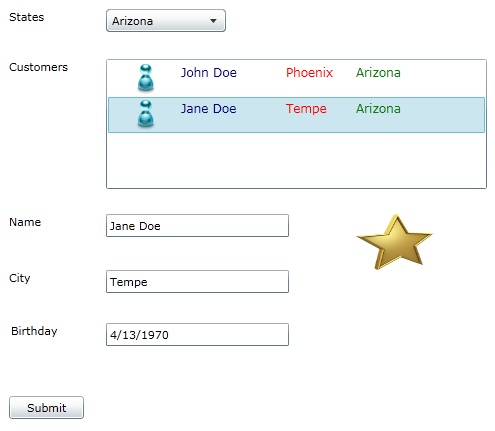
[Lab 4: Silverlight Data Binding 3](#_Toc276332990)

[Exercise 1: Create Data Entity Classes and Implement INotifyPropertyChanged 5](#_Toc276332991)

[Exercise 2: Creating a User Interface and Binding Data to Controls 13](#_Toc276332992)

[Summary 24](#_Toc276332993)

Lab 4: Silverlight Data Binding

* 1. Data binding is a key technology in Silverlight that allows data to be presented to end users and then processed. In this lab exercise you'll learn different data binding techniques that can be used to perform one way and two way bindings and see how data can be accessed directly without having to go through controls in the user interface.You'll also work with new binding properties available in Silverlight 4 and learn how to write a custom value converter. The application that you'll build in the lab exercises is shown next:
  2. 
  3. Figure 1
  4. The GUI application

#### You Will Benefit from this Lab if:

* + You need to bind data to a Silverlight user interface
  + You would like to learn how to visually bind data to controls in Visual Studio 2010

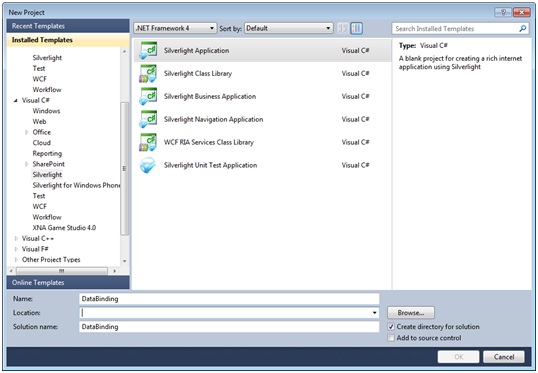
#### You Will Learn:

* + How to use the INotifyPropertyChanged interface
  + The role of DataContext in Silverlight applications
  + How to bind data to controls using the Visual Studio 2010 designer
  + How to access data using two way bindings
  + How to use the StringFormat property
  + How to create and use a value converter

#### Business Requirements for the Silverlight application include:

* + Create a new Silverlight project
  + Create data entity classes that implement INotifyPropertyChanged
  + Create a user interface capable of displaying and filtering customers
  + Bind data entity objects to Silverlight controls
  + Use the StringFormat property to control how data is output in the user interface
  + Create a value converter and use it within the user interface to show "gold" customers
  1. Estimated Time: 60 minutes

Exercise 1: Create Data Entity Classes and Implement INotifyPropertyChanged

* 1. In this exercise you'll create data entity classes with properties that will be bound to Silverlight controls. Two of the classes you'll create will implement INotifyPropertyChanged which is a key interface used for data binding in Silverlight applications.
  2. Create a new **Silverlight Application** project named **DataBinding** in Visual Studio 2010 (the project can be saved anywhere you'd like):
     1. 
     2. Figure 2
     3. New Project Dialog
  3. Right-click on the **ClientBin** folder in the **DataBinding.Web** project and **select Add 🡪 New Folder**. Name the folder **Images.**
  4. Copy the **blue.png** and **GoldStar.png** files from the lab's **Starting Point** folder into the **Images** folder using the **Add 🡪 Existing Item** option in Visual Studio.
  5. Locate the **DataBinding** project in the Solution Explorer and add the following classes into it by right-clicking on the project and selecting **Add 🡪 Class**:

|  |
| --- |
| Class Name |
| Customer |
| State |
| CustomerContainer |

* 1. Add the following properties into the **State** class (add them as standard .NET properties):

|  |  |
| --- | --- |
| Property | Type |
| Name | String |
| Abbreviation | String |

* 1. Open the **Customer** class and import the **System.ComponentModel** namespace.
  2. Implement the **INotifyPropertyChanged** interface on the **Customer** class (if you need assistance with this step please refer to the Completed folder and view the Customer class in the lab solution):

|  |  |
| --- | --- |
| Language | Code |
| C# | After adding the interface to the **Customer** class right-click it and select **Implement Interface 🡪 Implement Interface** from the menu. |
| Visual Basic | After adding the interface to the **Customer** class hit [Enter] to implement the interface. |

* + 1. **Note:** The INotifyPropertyChanged interface is a key part of the data binding infrastructure available in Silverlight. It contains a single event named PropertyChanged that is used to notify objects when a particular property value changes.
  1. Add the following **OnPropertyChanged** method into the **Customer** class to handle raising the **PropertyChanged event:**
     1. C#
     2. protected void OnPropertyChanged(string propName)
     3. {
     4. if (PropertyChanged != null)
     5. {
     6. PropertyChanged(this, new PropertyChangedEventArgs(propName));
     7. }
     8. }
     9. Visual Basic
     10. Protected Overridable Sub OnPropertyChanged(ByVal propName As String)
     11. RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propName))
     12. End Sub
  2. Add a **Name** property and associated field into the **Customer** class that raises the **PropertyChanged** event in its set block as shown next:
     1. C#
     2. string \_Name;
     3. public string Name
     4. {
     5. get { return \_Name; }
     6. set
     7. {
     8. if (\_Name != value)
     9. {
     10. \_Name = value;
     11. OnPropertyChanged("Name");
     12. }
     13. }
     14. }
     15. Visual Basic
     16. Private \_Name As String
     17. Public Property Name() As String
     18. Get
     19. Return \_Name
     20. End Get
     21. Set(ByVal value As String)
     22. If \_Name IsNot value Then
     23. \_Name = value
     24. OnPropertyChanged("Name")
     25. End If
     26. End Set
     27. End Property
     28. **Note:** If you're using Visual Basic ensure that **IsNot** is used for comparing reference types and **<>** is used for comparing value types when creating the properties that follow.
  3. Using the same pattern shown in the previous step, add the following properties and associated fields into the **Customer** class. Ensure that **OnPropertyChanged** is called in each **set** block and that the property name is passed as a parameter to the method:
     1. **Note:** A code snippet file is available in the lab's Starting Point folder that can be used to simplify the process of creating properties that call OnPropertyChanged. Use the Code Snippet Manager to import the appropriate snippet file (C# or VB) if you'd like to use the snippet. Once imported, the shortcut for the snippet is mvvmInpc.

|  |  |
| --- | --- |
| Property | Type |
| City | String |
| State | String |
| ImageUrl | String |
| Birthday | DateTime |
| IsGold | Boolean |

* 1. Open the **CustomerContainer** class and import the **System.ComponentModel**, **System.Linq** and **System.Collections.ObjectModel** namespaces.
  2. Implement **INotifyPropertyChanged** on the **CustomerContainer** class and add an **OnPropertyChanged** method into it to raise the event.
     1. **Note:** Anytime INotifyPropertyChanged must be implemented on multiple classes it's often more efficient to create a base class that implements the interface and provides the OnPropertyChanged method. Classes needing to implement the interface can then derive from the base class which provides better code-use and simplified maintenance. If time permits, create a base class that implements INotifyPropertyChanged and contains the OnPropertyChanged method and then derive Customer and CustomerContainer from it.
  3. Add the following properties into the **CustomerContainer** class. Ensure that each property's **set** block makes a call to **OnPropertyChanged** and passes the appropriate property name as a parameter. Follow the pattern shown earlier with the **Name** property in the **Customer** class.
     1. **Note:** A code snippet file is available in the Starting Point folder that can be used to simplify the process of creating properties that call OnPropertyChanged. Use the Code Snippet Manager to import the appropriate snippet file (C# or VB) if you'd like to use the snippet. Once imported, the shortcut for the snippet is mvvmInpc.

|  |  |
| --- | --- |
| Property | Type |
| States | ObservableCollection of State |
| Customers | ObservableCollection of Customer |
| FilteredCustomers | ObservableCollection of Customer |
| CurrentCustomer | Customer |
| CurrentState | State |

* 1. Add the following method into the **CustomerContainer** class to filter **Customer** objects based upon a **State**:
     1. C#

private void FilterCustomersByState()

* + 1. {
    2. if (CurrentState != null)
    3. {
    4. if (CurrentState.Name != "View All")
    5. {
    6. var customers = Customers.Where(c => c.State== CurrentState.Name);
    7. FilteredCustomers = new ObservableCollection<Customer>(customers);
    8. }
    9. else
    10. {
    11. FilteredCustomers = Customers;
    12. }
    13. }
    14. }
    15. Visual Basic
    16. Private Sub FilterCustomersByState()
    17. If CurrentState IsNot Nothing Then
    18. If CurrentState.Name <> "View All" Then
    19. Dim customers=Me.Customers.Where(Function(c) c.State=CurrentState.Name)
    20. FilteredCustomers = New ObservableCollection(Of Customer)(customers)
    21. Else
    22. FilteredCustomers = Customers
    23. End If
    24. End If
    25. End Sub
  1. Within the **set** block of the **CurrentState** property add a call to **FilterCustomersByState()**. It should look like the following code once completed:
     1. C#
     2. set
     3. {
     4. if (\_CurrentState != value)
     5. {
     6. \_CurrentState = value;
     7. OnPropertyChanged("CurrentState");
     8. FilterCustomersByState();
     9. }
     10. }
     11. Visual Basic
     12. Set(ByVal value As State)
     13. If \_CurrentState IsNot value Then
     14. \_CurrentState = value
     15. OnPropertyChanged("CurrentState")
     16. FilterCustomersByState()
     17. End If
     18. End Set
  2. Add the following constant into the **CustomerContainer** class:
     1. C#
     2. const string IMAGE = "Images/blue.png";
     3. Visual Basic
     4. Const IMAGE As String = "Images/blue.png"
  3. Add an empty constructor into **CustomerContainer** and add the following code into it to create **State** objects:
     1. C#
     2. States = new ObservableCollection<State>
     3. {
     4. new State{Name="Arizona",Abbreviation="AZ"},
     5. new State{Name="California",Abbreviation="CA"},
     6. new State{Name="Nevada",Abbreviation="NV"},  
         new State{Name="View All"}
     7. };
     8. Visual Basic
     9. States = New ObservableCollection (Of State)() From {
     10. New State With {.Name="Arizona", .Abbreviation="AZ"},
     11. New State With {.Name="California", .Abbreviation="CA"},
     12. New State With {.Name="Nevada", .Abbreviation="NV"},  
          New State With {.Name="View All"}}
  4. Add the following code into the constructor to create instances of the **Customer** class and assign them to the **Customers** property:
     1. **Note:** This code can be cut-and-paste from the CustomerContainer class in the lab solution available in the Completed folder.
     2. C#
     3. Customers = new ObservableCollection<Customer>
     4. {
     5. new Customer{Name="John Doe",City="Phoenix", State="Arizona",IsGold=true,
     6. Birthday=new DateTime(1950,5,10),ImageUrl=IMAGE},
     7. new Customer{Name="Jane Doe",City="Tempe", State="Arizona",
     8. Birthday=new DateTime(1970,4,13),ImageUrl=IMAGE},
     9. new Customer{Name="Johnny Doe",City="San Diego",State="California",
     10. Birthday=new DateTime(1980,8,26),ImageUrl=IMAGE},
     11. new Customer{Name="James Doe",City="Las Vegas",State="Nevada",IsGold=true,
     12. Birthday=new DateTime(1956,8,30),ImageUrl=IMAGE},
     13. new Customer{Name="Gina Doe",City="Anaheim",State="California",
     14. Birthday=new DateTime(1984,2,28),ImageUrl=IMAGE}

1. };  
   FilteredCustomers = Customers;
   * 1. Visual Basic
     2. Customers = New ObservableCollection(Of Customer)
     3. From {
     4. New Customer With {.Name = "John Doe", .City = "Phoenix", \_
     5. .State = "Arizona", .IsGold = True, \_
     6. .Birthday = New Date(1950, 5, 10), .ImageUrl = IMAGE},
     7. New Customer With {.Name = "Jane Doe", .City = "Tempe", \_
     8. .State = "Arizona", .Birthday = New Date(1970, 4, 13), \_
     9. .ImageUrl = IMAGE},
     10. New Customer With {.Name = "Johnny Doe", .City = "San Diego", \_
     11. .State = "California", .Birthday = New Date(1980, 8, 26), \_
     12. .ImageUrl = IMAGE},
     13. New Customer With {.Name = "James Doe", .City = "Las Vegas", \_
     14. .State = "Nevada", .IsGold = True, \_

.Birthday = New Date(1956, 8, 30), .ImageUrl = IMAGE},

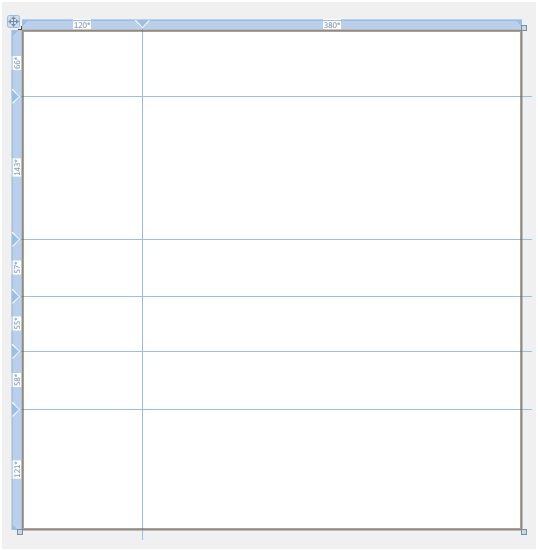
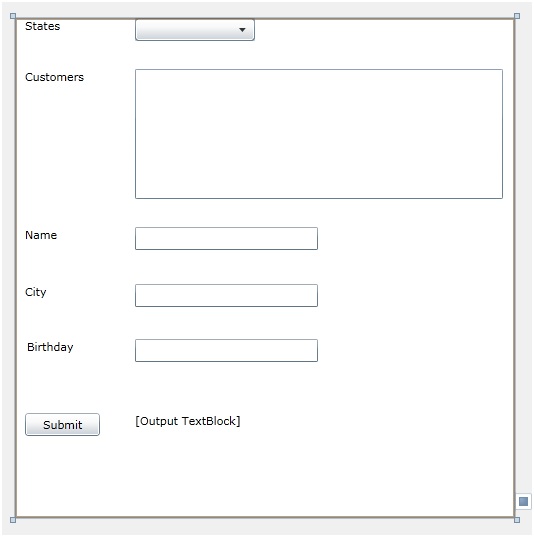
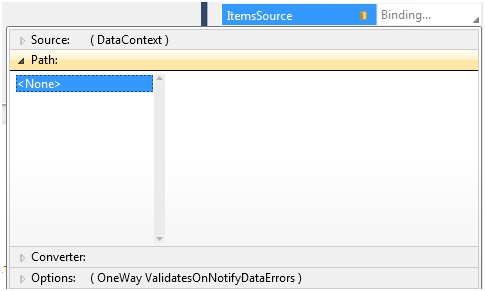
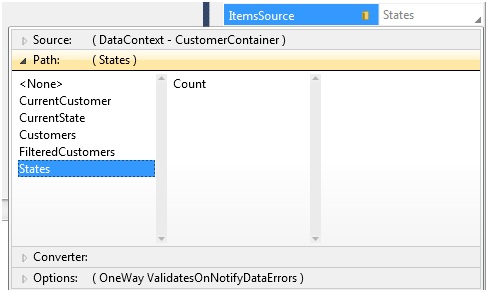
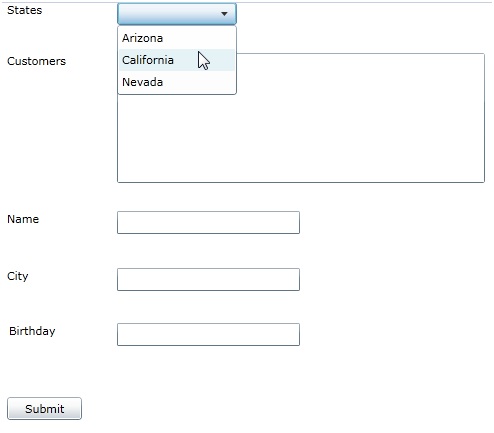
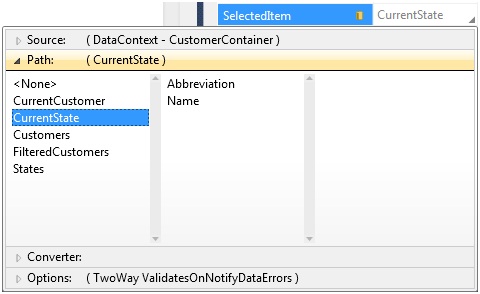
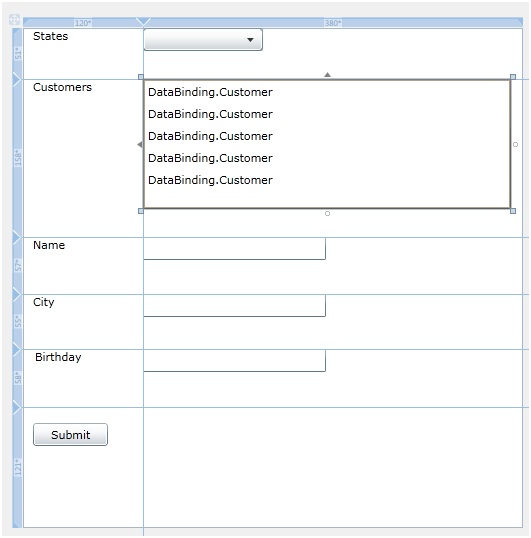
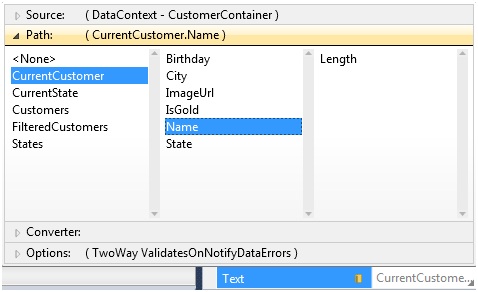
New Customer With {.Name = "Gina Doe", .City = "Anaheim", \_

.State = "California", .Birthday = New Date(1984, 2, 28), \_  
 .ImageUrl = IMAGE}}

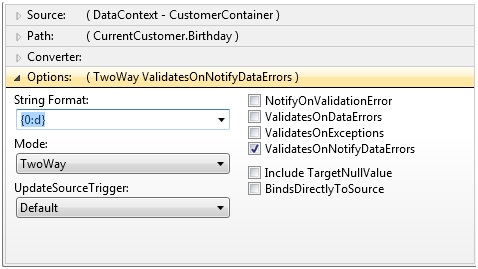
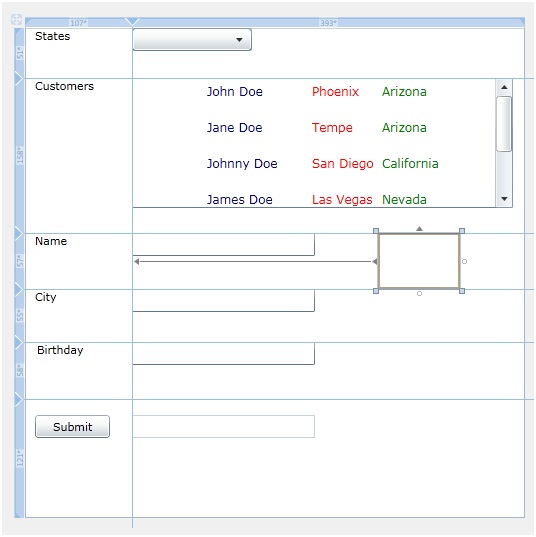
FilteredCustomers = Customers

* + 1. **Note:** Data is being added directly into the CustomerContainer class since the focus of this lab is on data binding. In a real-world application data would be retrieved from a Web Service or RESTful service. Additional labs in this series are available that cover retrieving data from distributed sources using WCF and WCF RIA Services.
  1. Build the project and ensure that no compilation errors occur before continuing.

Exercise 2: Creating a User Interface and Binding Data to Controls

* 1. In this exercise you'll build a user interface using standard Silverlight controls and bind an instance of the CustomerContainer class to the DataContext. You'll then bind object properties to controls within the user interface visually using Visual Studio 2010 and the Properties window.
  2. Open **MainPage.xaml** and change the **DesignHeight** and **DesignWidth** attributes on the **UserControl** element to **500** in the XAML
  3. Add **Height** and **Width** attributes to the **UserControl** element and give them a value of **500** to fix the size of the user interface.
  4. Add 6 rows and 2 columns using the designer as shown next:
     1. 
     2. Figure 3
     3. Grid Rows And Columns
     4. **Note:** Click on the Grid control in the Visual Studio designer and then add rows and columns by clicking within the blue regions to the left and top of the interface.
  5. Create the following customer information screen by dragging the appropriate controls from the ToolBox onto the Visual Studio designer:
     1. **Note:** You'll need 6 TextBlock controls, a ComboBox control, a ListBox control, 3 TextBox controls and a Button. Place the controls in the appropriate rows and columns of the Grid.
     2. 
     3. Figure 4
     4. Customer information screen
  6. Give the **TextBlock** with the text **[Output TextBlock]** in the designer a name of **OutputTextBlock** and remove the value from the **Text** property.
  7. Within the **MainPage.xaml.cs** constructor create a new instance of the **CustomerContainer** class and assign it to the LayoutRoot's **DataContext** property(the grid has a name of **LayoutRoot**):
     1. C#
     2. LayoutRoot.DataContext = new CustomerContainer();
     3. Visual Basic
     4. LayoutRoot.DataContext = New CustomerContainer()
  8. Switch back to **MainPage.xaml**, highlight the **ComboBox** control and view its properties in the **Properties** window
  9. Click the **ItemsSource** property and try to visually bind it to the **CustomerContainer** object's **States** property. Notice that none of the custom properties appear in the data binding window (see Figure 5). This is due to the **DataContext** being assigned at runtime rather than at design-time. Design-time data is important when you'd like to see data in the designer while building your application.
     1. 
     2. Figure 5
     3. Data Binding Window
  10. Remove the line of code you added into the **MainPage.xaml.cs** constructor. The next steps will demonstrate how to bind objects declaratively to provide a better design-time experience.
  11. Add the following XML namespace prefix definition on the **UserControl** element in the XAML (use the XAML code editor):
      1. XAML
      2. xmlns:data="clr-namespace:DataBinding"
  12. Add the following code immediately below the **UserControl** element (immediately above the existing **Grid):**
      1. XAML
      2. <UserControl.Resources>
      3. <data:CustomerContainer x:Key="CustomerContainerObject" />
      4. </UserControl.Resources>
      5. **Note**:This will create a new instance of the CustomerContainer object at runtime and assign it to the CustomerContainerObject key.
  13. Locate the **Grid** control named **LayoutRoot** in the XAML and add the following **DataContext** attribute to it:
      1. XAML
      2. DataContext="{Binding Source={StaticResource CustomerContainerObject}}"
      3. **Note:** This code binds the CustomerContainerObject key (which represents an instance of the CustomerContainer object) to the DataContext declaratively. This type of binding will execute in design-mode and at runtime.
  14. Build the solution so that the following data binding steps work properly.
  15. Bind the **ComboBox** control's **ItemsSource** property to the **CustomerContainer** object's **States** property using the **Properties** window as shown next:
      1. **Note:** The CustomerContainer object assigned to the DataContext is automatically detected as the Source.
      2. 
      3. Figure 6
      4. Data Binding Window
  16. Change the **ComboBox** control's **DisplayMemberPath** property to a value of **Name** so that the **ComboBox** shows the **Name** property of the **State** class.
  17. Run the solution and notice that once the Silverlight interface loads the **ComboBox** displays a list of states:
      1. 
      2. Figure 7
      3. The Silverlight User Interface
  18. Bind the **ComboBox** control's **SelectedItem** to the **CurrentState** property:
      1. 
      2. Figure 8
      3. ComboBox Binding
  19. Bind the **ListBox** control's **ItemsSource** to **FilteredCustomers** and its **SelectedItem** property to **CurrentCustomer** using the same technique shown in the previous step.
  20. Once the bindings are in place for the **ListBox** you'll see the text "**DataBinding.Customer"** appear multiple times in the designer as shown in the following figure since it doesn't know what Customer object property to bind to at this point:
      1. 
      2. Figure 9
      3. The ListBox at Design-Time
  21. To fix the issue, add the following **ItemTemplate** within the **ListBox** control using the XAML editor (add this XAML between the **ListBox** control's begin and end tags):
      1. XAML
      2. <ListBox.ItemTemplate>  
          <DataTemplate>  
          <Grid Width="350">  
          <Grid.ColumnDefinitions>  
          <ColumnDefinition Width=".20\*" />  
          <ColumnDefinition Width=".30\*" />  
          <ColumnDefinition Width=".20\*" />  
          <ColumnDefinition Width=".30\*" />  
          </Grid.ColumnDefinitions>  
           <Grid.RowDefinitions>  
          <RowDefinition Height="30" />  
          </Grid.RowDefinitions>  
          <Image Grid.Column="0" Source="{Binding Path=ImageUrl}" />  
          <TextBlock Grid.Column="1" Text="{Binding Path=Name}"
      3. FontSize="12" Foreground="Navy" />  
                     <TextBlock Grid.Column="2" Text="{Binding Path=City}"
      4. FontSize="12" Foreground="Red" />  
                     <TextBlock Grid.Column="3" Text="{Binding Path=State}"
      5. FontSize="12" Foreground="Green" />                         
          </Grid>  
          </DataTemplate>  
         </ListBox.ItemTemplate>
  22. Bind the **Name TextBox** control's **Text** property to **CurrentCustomer.Name:**
      1. 
      2. Figure 10
      3. Bind the NameTextBox
  23. Bind the **City** and **Birthday TextBox** controls to the associated properties on **CurrentCustomer** as shown in the previous step.
  24. Double-click the **Button** control to create an event handler and add the following code within the event handler to write out a message to the **OutputTextBlock** control:
      1. C#
      2. var customers = LayoutRoot.DataContext as CustomerContainer;
      3. var name = customers.CurrentCustomer.Name;
      4. OutputTextBlock.Text = name + " updated!";
      5. Visual Basic

Dim customers = CType(LayoutRoot.DataContext, CustomerContainer)

* + 1. Dim name = customers.CurrentCustomer.Name
    2. OutputTextBlock.Text = name + " updated!"
    3. **Note:** This code accesses the LayoutRoot object's DataContext and casts it to a CustomerConatiner type so that you can access the CurrentCustomer object's Name property value.
  1. Run the application and notice that all customers show in the **ListBox**.
  2. Select a state from the **ComboBox** to filter the customers. Click on a customer within the **ListBox** and note that the appropriate values show in the **TextBox** controls.
  3. The **Birthday TextBox** currently shows the date and time. To only show the date select the **TextBox** back in the Visual Studio designer and go to the **Text** property's data binding window. Click on the **Options** section and select the **{0:d}** format from the **String Format** drop-down as shown next:
     1. 
     2. Figure 11
     3. String Format
     4. **Note:** The StringFormat property provides a way to apply a format code to data as it is bound to a control. Standard .NET format codes used to format dates, times, decimals, currencies, and more can be used.
  4. Take a moment to examine the data binding syntax for the **Birthday TextBox** in the XAML editor and notice that a **StringFormat** property has been applied to the binding and that the **Mode** of the **TextBox** (and the other TextBox controls) is set to **TwoWay**. **TwoWay** bindings automatically push data from controls back to the bound property providing a powerful way to interact with data without having to know control names.
     1. XAML
     2. Text="{Binding Path=CurrentCustomer.Birthday, Mode=TwoWay,   
         StringFormat=\{0:d\}}"
  5. Run the application again and notice that the **BirthdayTextBox** only shows the date as a result of applying the **d** format code.
  6. To finish the application you need to show a gold star image if a customer's **IsGold** property is true. To accomplish this task drag an **Image** control onto the design surface and position it as shown by the highlighted control below:
     1. 
     2. Figure 12

Positioning the Image Control

* 1. Assign the **Image** control's **Source** property a value of **Images/GoldStar.png**.
  2. The **Image** control should only show if a customer's **IsGold** property is true. To hide the Image control when **IsGold** is false you'll need to create a value converter to convert a Boolean value to a Visibility value.
  3. Add a new folder named **Converters** into the **DataBinding** project.
  4. Add a new class named **BoolToVisibilityConverter** into the **Converters** folder and import the **System.Windows.Data** namespace.
  5. Implement the **IValueConverter** interface on the **BoolToVisibilityConverter** class and add the following code within the **Convert()** method:
     1. C#
     2. return ((bool)value == true) ? Visibility.Visible : Visibility.Collapsed;
     3. Visual Basic
     4. Return If(CBool(value) = True, Visibility.Visible, Visibility.Collapsed)
  6. Go to **MainPage.xaml** and add the following XML namespace prefix on the **UserControl** element:
     1. XAML
     2. xmlns:converters="clr-namespace:DataBinding.Converters"
  7. Add the following within the **UserControl.Resources** element to register the converter and make it available to use within the XAML:
     1. XAML
     2. <converters:BoolToVisibilityConverter x:Key="BoolToVisibilityConverter" />
  8. Add the following **Visibility** attribute on the **Image** control within the XAML to associate the converter with the **IsGold** property (ensure the attribute value doesn't wrap):
     1. XAML
     2. Visibility="{Binding CurrentCustomer.IsGold,Converter={StaticResource   
         BoolToVisibilityConverter},FallbackValue=Collapsed}"
     3. **Note:** When the screen first loads CurrentCustomer will be null so the converter will never be called. To account for this the FallbackValue property is used to define that the default Visibility is Collapsed which will hide the image. FallbackValue is used whenever a binding can't be resolved.
  9. Run the application and select the first customer in the **ListBox**. Notice that the gold star shows since the customer's **IsGold** property has a value of true. Select other customers and notice that the gold star disappears for some of them.

Summary

* 1. In this exercise you created data entity objects that implemented INotifyPropertyChanged and used Silverlight data binding features to bind object properties to controls. You also used the StringFormat property to format data and a value converter to hide a control based upon a property value. Specific requirements satisfied in this lab include:
  + Create a new Silverlight project
  + Create data entity classes that implement INotifyPropertyChanged
  + Create a user interface capable of displaying and filtering customers
  + Bind data entity objects to Silverlight controls
  + Use the StringFormat property to control how data is output in the user interface
  + Create a value converter and use it within the user interface to show "gold" customers