Data Binding in WPF

John Papa

Code download available at: [DataPoints2007\_12.exe](http://download.microsoft.com/download/f/2/7/f279e71e-efb0-4155-873d-5554a0608523/DataPoints2007_12.exe) (161 KB)   
[Browse the Code Online](http://msdn.microsoft.com/en-us/magazine/cc164232.aspx)

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Contents**

[Data Binding Specifics](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S1)   
[Creating a Simple Binding](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S2)   
[Binding Modes](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S3)   
[A Time to Bind](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S4)   
[Binding to XML](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S5)   
[Object Binding and DataTemplates](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S6)   
[Sorting the Data](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S7)   
[Forging Ahead](http://msdn.microsoft.com/en-us/magazine/cc163299.aspx#S8)

By now, many of you know that Windows® Presentation Foundation (WPF) makes it easy to design robust user interfaces. But what you probably didn't know is that it also provides powerful data-binding capabilities. With WPF, you can perform data manipulation using Microsoft® .NET Framework code, XAML, or a combination of both. You can bind to controls, public properties, XML, or objects, making data binding quick, flexible, and easier than ever. So let's take a look at how you can get started binding your controls to the data sources of your choice.

**Data Binding Specifics**

To use WPF data binding, you must always have a target and a source. The target of the binding can be any accessible property or element that is derived from DependencyProperty—an example is a TextBox control's Text property. The source of the binding can be any public property, including properties of other controls, common language runtime (CLR) objects, XAML elements, ADO.NET DataSets, XML Fragments, and so forth. To help you get the binding right, WPF includes two special providers—the XmlDataProvider and the ObjectDataProvider.

Now let's take a look at how the WPF data-binding techniques work and I'll present practical examples that illustrate their use.

**Creating a Simple Binding**

Let's start with a simple example that illustrates how to bind a TextBlock's Text property to a ListBox's selected item. The code displayed in **Figure 1** shows a ListBox that has six ListBoxItems declared. The second TextBlock in the code example has a property called Text (specified in XAML property-element syntax with the XML child element <TextBlock.Text>), which will contain the text for the TextBlock. The Text property declares a binding to the ListBox's selected item with the <Binding> tag. The ElementName attribute of the Binding tag indicates the name of the control that the TextBlock's Text property is bound to. The Path attribute indicates the property of the element (in this case the ListBox) that we will be binding to. The result of this code is that when a color is selected from the ListBox, the name of that color is shown in the TextBlock.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 1 Basic but Verbose Binding to a Control**

<StackPanel>

<TextBlock Width="248" Height="24" Text="Colors:"

TextWrapping="Wrap"/>

<ListBox x:Name="lbColor" Width="248" Height="56">

<ListBoxItem Content="Blue"/>

<ListBoxItem Content="Green"/>

<ListBoxItem Content="Yellow"/>

<ListBoxItem Content="Red"/>

<ListBoxItem Content="Purple"/>

<ListBoxItem Content="Orange"/>

</ListBox>

<TextBlock Width="248" Height="24" Text="You selected color:" />

<TextBlock Width="248" Height="24">

<TextBlock.Text>

<Binding ElementName="lbColor" Path="SelectedItem.Content"/>

</TextBlock.Text>

</TextBlock>

</StackPanel>

The code listed in **Figure 1** can be modified slightly to use a shorthand syntax for data binding. For example, let's replace the TextBlock's <Binding> tag with the following code snippet:

<TextBlock Width="248" Height="24"

Text="{Binding ElementName=lbColor,

Path=SelectedItem.Content}" />

This syntax, called the attribute syntax, condenses the data binding code inside of the Text attribute of the TextBlock. Basically, the Binding tag gets pulled inside of the curly braces along with its attributes.

**Binding Modes**

I can take the previous example further, binding the background color of the TextBlock to the color that is selected in the ListBox. The following code adds a Background property to the TextBlock and uses the attribute binding syntax to bind to the value of the selected item in the ListBox:

<TextBlock Width="248" Height="24"

Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=OneWay}"

x:Name="tbSelectedColor"

Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=OneWay}"/>

When the user selects a color in the ListBox, the name of that color will appear in the TextBlock and the background color of the TextBlock will change to the selected color (see **Figure 2**).

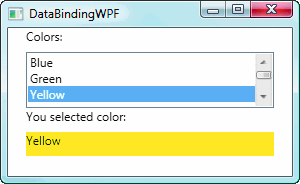


Figure 2**Binding a Source to Two Targets**

Notice the statement in the previous example that sets the Mode attribute to OneWay. The Mode attribute defines the binding mode that will determine how data flows between the source and the target. In addition to OneWay, there are three other binding modes available: OneTime, OneWayToSource, and TwoWay,

When using OneWay binding, as shown in the preceding code snippet, the data flows from the source to the target each time a change is made on the source. And while I explicitly specified this binding mode in my example, OneWay binding is the default binding mode for the TextBlock's Text property and does not need to be specified. Like OneWay binding, OneTime binding sends data from the source to the target; however, it does this only when the application is started or when the DataContext changes and, as a result, does not listen for change notifications in the source. Unlike OneWay and OneTime binding, OneWayToSource binding sends data from the target to the source. Finally, TwoWay binding sends the source data to the target, and if there are changes in the target property's value, those will be sent back to the source.

In the previous example, I used OneWay binding because I wanted the source (the selected ListBoxItem) to be sent to the TextBlock whenever a change is made in the ListBox selection. I do not want changes from the TextBlock to go back to the ListBox. Of course, there is no way for a user to edit a TextBlock. If I want to explore TwoWay binding, I can add a TextBox to this code, bind its text and background color to the ListBox, and set the Mode to TwoWay. When a user selects a color in the ListBox, the color is shown in the TextBox and its background color changes. When that user types in a color (such as Cyan) in the TextBox, the name of the color is updated in the ListBox (target to source) and in turn, since the ListBox was updated, the new value is sent to all elements that are bound to the ListBox's SelectedItem property. This means that the TextBlock will also have its color updated and its text value set to the new color (see **Figure 3**).

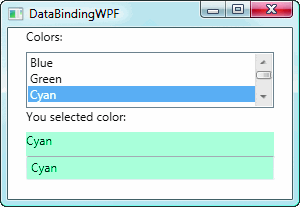


Figure 3**TwoWay Binding in Action**

Here's the code I used to bind the TextBlock (OneWay) and the TextBox (TwoWay) to the ListBox:

<TextBlock Width="248" Height="24"

Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=OneWay}" x:Name="tbSelectedColor"

Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=OneWay}"/>

<TextBox Width="248" Height="24"

Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=TwoWay}" x:Name="txtSelectedColor"

Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,

Mode=OneWay}"/>

If I change the TwoWay mode back to OneWay, the user will be able to edit the color in the TextBox without causing the changed value to be sent back to the ListBox.

Selecting the appropriate binding mode is important. I often employ OneWay when I want to display read-only data to a user. I use TwoWay binding when I want the user to be able to change the data in the control and have that change reflected in the data source (a DataSet, object, XML, or another bound control). I find OneWayToSource to be a good choice when I want to allow a user to change the data source without having the data source bind its data back to the target. I have used OneTime binding when I was tasked with showing, in a read-only control, the state of the data as it was when the screen loaded. Using OneTime binding, a series of read-only controls was bound to the data and, as the user interacted with the form and the data source's values were changed, the bound controls remained unchanged. This provided a way for the users to compare the changes that were made. Additionally, OneTime binding is a good choice when your source doesn't implement INotifyPropertyChanged.

**A Time to Bind**

In the previous example, the TextBox allows TwoWay binding to the selected ListBoxItem in the ListBox. This flow of data from the TextBox back to the ListBox happens when the TextBox loses focus. To change the event that causes the data to be sent back to the source, you can specify a value for UpdateSourceTrigger, which is the binding property that defines when the source should be updated. There are three values that can be set for the UpdateSourceTrigger: Explicit, LostFocus, and PropertyChanged.

When you set the UpdateSourceTrigger to Explicit, the source will not be updated unless the BindingExpression.UpdateSource method is called from code. The LostFocus setting (the default value for the TextBox control) indicates that the source will be updated when the target control loses focus. The PropertyChanged value indicates that the target will update the source every time the target control's bound property changes. This setting is useful when you want to dictate when the binding will occur.

**Binding to XML**

Binding to data sources such as XML and objects is also handy. **Figure 4** shows a sample of an XmlDataProvider that contains an embedded list of colors that will be used as a data source. The XmlDataProvider can be used to bind to an XML document or fragment that is either embedded in the XmlDataProvider tag or is in a file referred to in an external location.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 4 XmlDataProvider**

<StackPanel>

<StackPanel.Resources>

<XmlDataProvider x:Key="MoreColors" XPath="/colors">

<x:XData>

<colors >

<color name="pink"/>

<color name="white"/>

<color name="black"/>

<color name="cyan"/>

<color name="gray"/>

<color name="magenta"/>

</colors>

</x:XData>

</XmlDataProvider>

Embedded XML content must be placed within a <x:XData> tag inside of an XmlDataProvider, as shown in**Figure 4**. The XmlDataProvider must be given an x:Key value so that it can be referred to by data-binding targets. Notice that the XPath attribute is set to "/colors". This attribute defines the level of the XML content that will be used as the data source. This becomes very useful when binding to a large XML structure that may be contained in a file or database and the data you want to bind to is not the root element.

An XmlDataProvider is a resource that can be placed inside of a context-specific resource. As **Figure 4** shows, the XmlDataProvider is defined as a resource within the context of the StackPanel. This means that the XmlDataProvider will be available to all content inside of that StackPanel. Setting the context of a resource helps limit the exposure of a data source to the appropriate areas. This enables you to create well-defined, self-contained regions of both controls and supporting resources within your page, thus improving readability.

The syntax for binding to a resource is slightly different than it is for binding to an element. When binding to a control, you set the ElementName and the Path properties of the Binding. However, when you bind to a resource you set the Source and, since we are binding to an XmlDataProvider, you set the XPath property of the Binding as well. For example, the following code will bind the ListBox's items to the MoreColors resource. The Source property is set to a resource and it is specified as a StaticResource named MoreColors. The XPath property indicates that the items will be bound to the <color> element's name attribute within the XML data source:

<ListBox x:Name="lbColor" Width="248" Height="56"

IsSynchronizedWithCurrentItem="True"

ItemsSource="{Binding Source={StaticResource MoreColors},

XPath=color/@name}">

</ListBox>

I specified StaticResource in this case because the XML will not change. If changes occur in the data source, they will not be sent to the target. The DynamicResource setting indicates the opposite—changes will be sent. This is useful when referencing system themes, languages in globalization, or fonts. A DynamicResource will allow these types of settings to be propagated throughout the UI elements that are bound to them dynamically.

The XmlDataProvider can also point to an external source for the XML content. For my example, I have a file named colors.xml that contains the list of colors I want my ListBox to be bound to. I can simply add a second XmlDataProvider resource to the StackPanel and direct it to the XML file. Notice I set the Source attribute to the name of the XML file and the x:Key to Colors:

<XmlDataProvider x:Key="Colors" Source="Colors.xml" XPath="/colors"/>

Both XmlDataProviders exist as resources within the same StackPanel. I can tell the ListBox to bind itself to this new resource by changing the name that the StaticResource is set to:

<ListBox x:Name="lbColor" Width="248" Height="56"

IsSynchronizedWithCurrentItem="True"

ItemsSource="{Binding Source={StaticResource Colors},

XPath=color/@name}">

</ListBox>

**Object Binding and DataTemplates**

While the XmlDataProvider works great for XML, when you want to bind to an object or a list of objects, you can create an ObjectDataProvider as a resource. The ObjectDataProvider's ObjectType designates the object that will provide the data‑binding source while the MethodName indicates the method that will be invoked to get the data. For example, assuming I have a class called PersonService that has a method called GetPersonList that returns a List<Person>, the ObjectDataProvider would look like this:

<StackPanel.Resources>

<ObjectDataProvider x:Key="persons"

ObjectType="{x:Type svc:PersonService}"

MethodName="GetPersonList"></ObjectDataProvider>

</StackPanel.Resources>

If you want a more complete look, the PersonService and Person classes, as well as all other sample code, are contained in the code that accompanies this column.

There are a handful of other properties that are available on the ObjectDataProvider. The ConstructionParameters property allows you to pass parameters to the constructor of the class being invoked. You can also specify parameters using the MethodParameters property, and you can use the ObjectInstance property to specify an existing instance of an object as the source.

If you want the data to be retrieved asynchronously, you can set the IsAsynchronous property of the ObjectDataProvider to true. Then the user will be able to interact with the screen while waiting for the data to populate in the target control that is bound to the ObjectDataProvider's source.

When adding an ObjectDataProvider, you have to qualify the namespace of the data-source class. In this case, I have to add an xmlns attribute to the <Window> tag so that the svc shortcut is qualified and indicates the proper namespace:

xmlns:svc="clr-namespace:DataBindingWPF"

Now that the data source is defined via the ObjectDataProvider, I want to bind items in a ListBox control to this data. I want to display two lines of text in each ListBoxItem. The first line will display the FullName property of the Person instance in bold and the second line will show the Title and City of the instance. In XAML, this is quite simple using DataTemplates, which allow you to define a data visualization strategy that can be reused.

Figure 5 shows the completed XAML, which has a DataTemplate defined to display the Person information in the layout I designated. I set the DataType property of the DataTemplate to indicate that the DataTemplate will be referring to the Person class type. I do not specify the actual binding in the DataTemplate, as I will do this in the ListBox control. By omitting the Binding Source, the binding will be made to the current DataContext in scope.

In **Figure 5**, I set the ListBox's ItemsSource property to bind to the persons resource so that I could bind the data to the ListBox but not format it. The data is correctly displayed by setting the ItemTemplate property to the personLayout resource, which is the DataTemplate's key name. The ultimate result is a screen that looks like **Figure 6**.

**http://i.msdn.microsoft.com/Global/Images/clear.gif  Figure 5 Object Binding**

<Window x:Class="DataBindingWPF.ObjectBinding"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:svc="clr-namespace:DataBindingWPF"

Title="DataBindingWPF" Height="300" Width="300">

<StackPanel>

<StackPanel.Resources>

<ObjectDataProvider x:Key="persons"

ObjectType="{x:Type svc:PersonService}"

MethodName="GetPersonList" ></ObjectDataProvider>

<DataTemplate x:Key="personLayout" DataType="Person">

<StackPanel Orientation="Vertical">

<TextBlock Text="{Binding Path=FullName}"

FontWeight="Bold" Foreground="Blue">

</TextBlock>

<StackPanel Orientation="Horizontal">

<TextBlock Text="{Binding Path=Title}"></TextBlock>

<TextBlock Text=", "></TextBlock>

<TextBlock Text="{Binding Path=City}"></TextBlock>

</StackPanel>

</StackPanel>

</DataTemplate>

</StackPanel.Resources>

<TextBlock></TextBlock>

<ListBox x:Name="lbPersons"

ItemsSource="{Binding Source={StaticResource persons}}"

ItemTemplate="{DynamicResource personLayout}"

IsSynchronizedWithCurrentItem="True"/>

</StackPanel>

</Window>

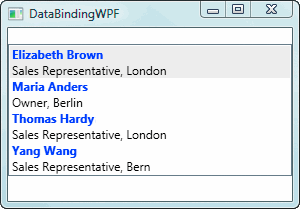


Figure 6**Using a DataTemplate**

**Sorting the Data**

If you want to sort your data in a specific way, you can bind to a CollectionViewSource instead of directly to the ObjectDataProvider. The CollectionViewSource then becomes the data source and serves as an intermediary that intercepts the data from the ObjectDataProvider; provides sorting, grouping, and filtering functionality; and then propagates it to the target.

The CollectionViewSource shown next has its Source attribute set to the resource name of the ObjectDataProvider (persons). I then defined a sort order for the data by indicating the properties to sort by and their direction:

<CollectionViewSource x:Key="personView"

Source="{Binding Source={StaticResource persons}}">

<CollectionViewSource.SortDescriptions>

<ComponentModel:SortDescription

PropertyName="City"

Direction="Ascending" />

<ComponentModel:SortDescription

PropertyName="FullName"

Direction="Descending" />

</CollectionViewSource.SortDescriptions>

</CollectionViewSource>

The DataContext is used to bind all controls within a container control to a data source. This is very useful when you have several controls that all use the same binding source. The code could get repetitive if you indicated the binding source for every control. Instead, you can set the DataContext for the container of the controls to the binding source and simply omit the Source attribute from the contained controls. For example, here is a series of TextBlocks bound explicitly to the same binding source:

<StackPanel>

<TextBlock Text="{Binding Source={StaticResource personView},

Path=FullName}"></TextBlock>

<TextBlock Text="{Binding Source={StaticResource personView},

Path=Title}"></TextBlock>

<TextBlock Text="{Binding Source={StaticResource personView},

Path=City}"></TextBlock>

</StackPanel>

Here are the same three TextBoxes bound to the DataContext, which refers back to the controls' StackPanel container:

<StackPanel DataContext="{Binding Source={StaticResource personView}}" >

<TextBlock Text="{Binding Path=FullName}"></TextBlock>

<TextBlock Text="{Binding Path=Title}"></TextBlock>

<TextBlock Text="{Binding Path=City}"></TextBlock>

</StackPanel>

If the container does not define a DataContext, then it will continue to look at the next outer nested container until it finds the current DataContext.

**Forging Ahead**

WPF data binding offers a great deal of flexibility and control over the type of data that can be bound to, and how it can be controlled and displayed. With so much power and so many choices, I'm sure you'll want to get your hands dirty right away.

Send your questions and comments for John to [mmdata@microsoft.com](mailto:mmdata@microsoft.com).

# WPF Tutorial - Concept Binding

By **[Abhishek Sur](http://www.codeproject.com/script/Membership/View.aspx?mid=4293807)**, 31 Dec 2010

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | http://s.codeproject.com/script/Ratings/Images/stars-fill-lg.png  http://s.codeproject.com/script/Ratings/Images/stars-empty-lg.png | 4.78 (19 votes) | |  |

Top of Form

* [**Download source - 265.27 KB**](http://www.codeproject.com/KB/WPF/wpf6/WPFSampleApp.zip)

## Table of Contents

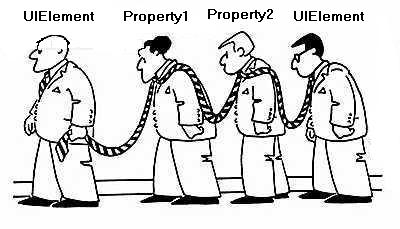
* [Introduction](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0000)
* [Binding in **WPF**](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0001)
  + [DataBinding / Object Binding](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0002)
    - [Why ObservableCollection](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0003)
  + [XML Binding](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0004)
* [Importance of DataContext](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0005)
* [Binding Members](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0006)
* [Binding in Code-behind](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0007)
* [Command Binding](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0008)
* [MultiBinding](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0009)
* [Conclusion](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding#heading0010)

## Introduction

Before this article, I have discussed about the architecture of **WPF**, Markup extensions, dependency properties, logical trees and Visual trees, layout, transformation, etc. Today, I will discuss what we call the most important part of any**WPF** application, the binding. **WPF** comes with superior DataBinding capabilities which enables the user to bind objects so that whenever the other object changes, the main object reflects its changes. The main motive ofDataBinding is to ensure that the UI is always synchronized with the internal object structure automatically.

Before going further, let's jot down the things that we have already discussed. If you are new to this article, you can start from my other articles in the list below:

* [**WPF** Tutorial : Beginning](http://www.codeproject.com/KB/WPF/wpf1.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf1.aspx)]
* [**WPF** Tutorial : Layout-Panels-Containers & Layout Transformation](http://www.codeproject.com/KB/WPF/wpf2.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf2.aspx)]
* [**WPF** Tutorial : Fun with Border & Brush](http://www.codeproject.com/KB/WPF/wpf3.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf3.aspx)]
* [**WPF** Tutorial - TypeConverter & Markup Extension](http://www.codeproject.com/KB/WPF/wpf4.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf4.aspx)]
* [**WPF** Tutorial - Dependency Property](http://www.codeproject.com/KB/WPF/wpf5.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf5.aspx)]
* [**WPF** Tutoriall - Concept Binding](http://www.codeproject.com/KB/WPF/wpf6.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf6.aspx/)]
* [**WPF** Tutorial - Styles, Triggers & Animation](http://www.codeproject.com/KB/WPF/wpf7.aspx) [[^](http://www.codeproject.com/KB/WPF/wpf7.aspx)]



DataBinding was present before the introduction of **WPF**. In ASP.NET, we bind data elements to render proper data from the control. We generally pass in a DataTable and bind the Templates to get data from individual DataRows. On the other hand, in case of traditional windows forms application, we can also bind a property with a data element. The Bindings can be added to properties of objects to ensure whenever the property changes the value, the data is internally reflected to the data. So in one word, DataBinding is nothing new to the system. The main objective ofDataBinding is to show data to the application and hence reduce the amount of work the application developer needs to write to just make the application properly display data. In this article, I will discuss how you could use theDatabinding in **WPF** application and also create a sample application to demonstrate the feature in depth.

## Binding in WPF

**WPF** puts the concept of Binding further and introduced new features, so that we could use the Binding feature extensively. Binding establishes the connection between the application and the business layers. If you want your application to follow strict design pattern rules, DataBinding concept will help you to achieve that. We will look into greater detail with how to do that in a while.

In **WPF**, we can bind two Properties, one Property and one DependencyProperty, two DependencyProperties etc.**WPF** also supports Command Binding. Let's discuss how to implement them in detail.

Binding can be classified into few Types.

### DataBinding / Object Binding

The most important and primary binding is Databinding. **WPF** introduces objects like ObjectDataProvider andXMLDataProvider to be declared into XAML to enhance the capability of object binding. DataBinding can be achieved by several ways. As shown by Adnan in his [blog](http://coredotnet.blogspot.com/2006/05/wpf-data-binding-tutorial.html) [[^](http://coredotnet.blogspot.com/2006/05/wpf-data-binding-tutorial.html)], we can make use of Binding capabilities by employing either XAML, XAML and C#, and C# itself. So **WPF** is flexible enough to handle any situation.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<TextBox x:Name="txtName" />

<TextBlock Text="{Binding ElementName=txtName, Path=Text.Length}" />

In the above situation, I have shown the most basic usage of Binding. The Text property of TextBlock is bound with the TextBox txtName so that whenever you enter something on the TextBox during runtime, the TextBlock will show the length of the string.

As a Markup Extension binding is actually a Class with properties, here we specified the value of the propertyElementName and Path. The ElementName ensures the object that the property belongs to. Path determines the property path which the object needs to look into.

You can use ObjectDataProvider to handle data in your XAML easily. ObjectDataProvider can be added as Resource and later on can be referenced using StaticResource. Let's see the code below:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<StackPanel Orientation="Vertical">

<StackPanel.Resources>

<ObjectDataProvider ObjectType="{x:Type m:StringData}"

x:Key="objStrings" MethodName="GetStrings"/>

</StackPanel.Resources>

<ListBox Name="lstStrings" Width="200" Height="300"

ItemsSource="{Binding Source={StaticResource objStrings}}" />

Just as shown above, the ObjectType will get a Type, which is the internal class structure for which the methodGetStrings will be called for. From the ListBox , I have referenced the Object using StaticResource. Now in the code, you can declare a class:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

public class StringData

{

ObservableCollection<String> lst= new ObservableCollection<String>();

public StringData()

{

lst.Add("Abhishek");

lst.Add("Abhijit");

lst.Add("Kunal");

lst.Add("Sheo");

}

public ObservableCollection<String> GetStrings()

{

return lst;

}

}

So you can see the list has been populated with the strings.

#### Why ObservableCollection , the INotifyPropertyChanged, INotifyCollectionChanged?

Now as you can see, I have used ObvervableCollection. This is important. ObservableCollection sends automatic notification when a new item is inserted. Thus notifies the ListBox to update the list. So if you place a button,which inserts some data in the ObservableCollection , the Binding will automatically be notified by the collection and hence update the collection automatically. You don't need to manually insert the same in the ListBox.

**WPF** Binding generally needs to be notified when it is modified. The interfaces INotifyPropertyChanged andINotifyCollectionChanged are needed to update the UIElement which is bound with the data. So if you are crating a property which needed to update the UI when the value of it is modified, the minimum requirement is to implement the same from INotifyPropertyChanged, and for collection (like ItemsSource), it needs to implementINotifyCollectionChanged. ObservableCollection itself implements INotifyCollectionChanged, so it has support to update the control whenever new item is inserted to the list or any old item is removed from the string.

I have already discussed the two in detail in an article : [Change Notification for Objects and Collection](http://www.abhisheksur.com/2010/05/object-notifiers-using.html) [[^](http://www.abhisheksur.com/2010/05/object-notifiers-using.html)].   
  
On the contrary, Sacha has a good point of getting rid of INotifyPropertyChanged interface using [Aspect](http://www.codeproject.com/KB/miscctrl/Aspects.aspx)**[Example](http://www.codeproject.com/KB/miscctrl/Aspects.aspx)**[s (INotifyPropertyChanged via aspects).](http://www.codeproject.com/KB/miscctrl/Aspects.aspx)

### XML Binding

Similar to Object binding, XAML also supports XML binding. You can bind the data coming from XMLDataProvidereasily using built in properties like XPath in Binding class definition. Let's look into the code:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<TextBlock Text="{Binding XPath=@description}"/>

<TextBlock Text="{Binding XPath=text()}"/>

So, if you are in the node XYZ, the InnerText can be fetched using text() property. The @ sign is used for Attributes. So using XPath, you can easily handle your XML.

If you want to read more about XML binding, check: [XML Binding in **WPF**](http://www.abhisheksur.com/2010/07/xml-binding-in-wpf-with-sample-rss.html) [[^](http://www.abhisheksur.com/2010/07/xml-binding-in-wpf-with-sample-rss.html)].

## Importance of DataContext

You might wonder why I have taken context of DataContext while I am talking about **WPF** Bindings. DataContext is actually a Dependency property. It points to Raw Data such that the object that we pass as DataContext will inherit to all its child controls. I mean to say if you define the DataContext for a Grid, then all the elements that are inside theGrid will get the same DataContext.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<Grid DataContext="{StaticResource dtItem}">

<TextBox Text="{Binding MyProperty}" />

</Grid>

Here as I defined DataContext for the Grid, the TextBox inside the grid can refer to the property MyProperty as the dtItem object will be automatically inherited to all its child elements. While using Binding, DataContext is the most important part which you must use.

## Binding Members

As you all know about Markup Extensions, Binding is actually a Markup Extension. It is a class Binding with few properties. Let's discuss about the Members that are there in Binding:

1. **Source**: The source property holds the DataSource. By default, it references the DataContext of the control. If you place Source property for the Binding, it will take that in lieu of original DataContext element.
2. **ElementName**: In case of Binding with another Element, ElementName takes the name of the Elementdefined within the XAML for reference of the object. ElementName acts as a replacement to Source. If path is not specified for the Binding, it will use ToString to get the data from the Object passed as Source.
3. **Path**: Path defines the actual property path to get the String Data. If the end product is not a string, it will also invoke ToString to get the data.
4. **Mode**: It defines how the Data will be flown. OneWay means object will be updated only when source is updated, on the contrary OneWayToSource is the reverse. TwoWay defines the data to be flown in both ways.
5. **UpdateSourceTrigger**: This is another important part of any Binding. It defines when the source will be updated. The value of UpdateSourceTrigger can be :
   * **PropertyChanged**: It is the default value. As a result, whenever anything is updated in the control, the other bound element will reflect the same.
   * **LostFocus**: It means whenever the property loses its focus, the property gets updated.
   * **Explicit**: If you choose this option, you need to explicitly set when to update the Source. You need to useUpdateSource of BindingExpression to update the control.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

BindingExpression bexp = mytextbox.GetBindingExpression(TextBox.TextProperty);

bexp.UpdateSource();

By this, the source gets updated.

1. **Converter**: Converter gives you an interface to put an object which will be invoked whenever the Binding objects get updated. Any object that implements IValueConverter can be used in place of Converter. You can read more about it from [Converter in Binding](http://www.abhisheksur.com/2010/03/how-to-use-ivalueconverter-in-binding.html) [[^](http://www.abhisheksur.com/2010/03/how-to-use-ivalueconverter-in-binding.html)].
2. **ConverterParameter**: It is used in addition to Converter to send parameters to Converter.
3. **FallbackValue**: Defines the value which will be placed whenever the Binding cannot return any value. By default, it is blank.
4. **StringFormat**: A formatting string that indicates the Format to which the data will follow.
5. **ValidatesOnDataErrors**: When specified, the DataErrors will be validated. You can use IDataErrorInfo to run your custom Validation block when Data object is updated. You can read more about IDataErrorInfofrom : [Validate your application using IDataErrorInfo](http://www.abhisheksur.com/2010/06/validate-your-application-using.html) [[^](http://www.abhisheksur.com/2010/06/validate-your-application-using.html)].

## Binding in Code-behind

Similar to what you might do with XAML, you can also define binding in the codeBehind. To do this, you need to use:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

Binding myBinding = new Binding("DataObject");

myBinding.Source = myDataObject;

myTextBlock.SetBinding(TextBlock.TextProperty, myBinding);

You can also specify the Binding properties in this way.

## Command Binding

**WPF** supports CommandBinding. Each command object like Button exposes a property called Command which takes an object that implements ICommand interface and will execute the method Execute whenever object command gets fired.

Say, you want your command to be executed whenever the window Inputs gets invoked:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<Window.InputBindings>

<KeyBinding Command="{Binding CreateNewStudent}" Key="N" Modifiers="Ctrl" />

<MouseBinding Command="{Binding CreateNewStudent}"

MouseAction="LeftDoubleClick" />

</Window.InputBindings>

In the above code, the CreateNewStudent is a property that exposes the object which inherits ICommand interface and the Execute method will be invoked whenever the Key Ctrl + N or LeftDoubleClick of the window is invoked.

**Note**: In VS 2008, the InputBindings only take Static Command objects. [There is a bug report for this](http://connect.microsoft.com/VisualStudio/feedback/details/431001/the-keybinding-command-property-should-be-a-dependencyproperty) [[^](http://connect.microsoft.com/VisualStudio/feedback/details/431001/the-keybinding-command-property-should-be-a-dependencyproperty)] , and it will be fixed in later releases.

You can use CommandParameter to pass parameters to the methods that make up the ICommand interface.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<Button Content="CreateNew" Command="{Binding CreateNewStudent}" />

Similar to InputBindings, you can use the Command with a Button. To execute, you need to create an object that implements ICommand like below:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

public class CommandBase : ICommand

{

private Func<object, bool> \_canExecute;

private Action<object> \_executeAction;

private bool canExecuteCache;

public CommandBase(Action<object>executeAction, Func<object, bool> canExecute)

{

this.\_executeAction = executeAction;

this.\_canExecute = canExecute;

}

#region ICommand Members

public bool CanExecute(object parameter)

{

bool tempCanExecute = \_canExecute(parameter);

canExecuteCache = tempCanExecute;

return canExecuteCache;

}

private event EventHandler \_canExecuteChanged;

public event EventHandler CanExecuteChanged

{

add { this.\_canExecuteChanged += value; }

remove { this.\_canExecuteChanged -= value; }

}

protected virtual void OnCanExecuteChanged()

{

if (this.\_canExecuteChanged != null)

this.\_canExecuteChanged(this, EventArgs.Empty);

}

public void Execute(object parameter)

{

\_executeAction(parameter);

}

#endregion

}

I have used a CommandBase class to make the objects look less clumsy. The actual object class looks like:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

private CommandBase createNewstudent;

public CommandBase CreateNewStudent

{

get

{

this.createNewstudent = this.createNewstudent ??

new CommandBase(param => this.CreateStudent(), param => this.CanCreateStudent);

return this.createNewstudent;

}

}

private object CreateStudent()

{

this.CurrentStudent = new StudentItem();

return this.CurrentStudent;

}

public bool CanCreateStudent

{

get { return true; }

}

Thus, you can see the createNewCommand passes CreateStudent lamda expression which is called whenever the object gets updated. The CanCreateStudent is a property that will also be called and based on true or false, **WPF**will allow the command to execute.



The PropertyBinding and CommandBinding give a total package to separate the presentation logic from the Presentation Layer. This gives the architecture to put all the logic separated. Microsoft created the whole Expression blend using MVVM pattern which separates the View from the ViewModel and hence gives a chance to handle Unit Testing easily even for presentation layer. We will discuss more about the topic later on the series.

## MultiBinding

Similar to single Binding, **WPF** also introduces the concept of MultiBinding. In case of MultiBinding, the data bound depends on more than one source. You can specify more than one binding expression and on each of them the actual output is dependent on.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

<TextBlock DockPanel.Dock="Top" >

<TextBlock.Text>

<MultiBinding Converter="{StaticResource mbindingconv}">

<Binding ElementName="lst" Path="Items.Count" />

<Binding ElementName="txtName" Path="Text" />

<Binding ElementName="txtAge" Path="Text" />

</MultiBinding>

</TextBlock.Text>

</TextBlock>

Here, the value for TextBlock is dependent on 3 elements, the first one is the ListBox count, then txtName andtxtAge. I have used Converter to ensure we find all the individual elements in the **IMultiValueConverter** block and handle each value separately. The **IMultiValueConverter** just similar to IValueConverter can take the value and return the object that is bound to the Text property.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/140621/WPF-Tutorial-Concept-Binding)

public class MyMultiBindingConverter : **IMultiValueConverter**

{

#region **IMultiValueConverter** Members

public object Convert(object[] values, Type targetType,

object parameter, System.Globalization.CultureInfo culture)

{

string returnval = "Total no of Data {0}, NewData : ";

if (values.Count() <= 0) return string.Empty;

returnval = string.Format(returnval, values[0]);

for (int i = 1; i < values.Count(); i++)

returnval += "- " + values[i];

return returnval;

}

public object[] ConvertBack(object value, Type[]

targetTypes, object parameter, System.Globalization.CultureInfo culture)

{

throw new NotImplementedException();

}

#endregion

}

For simplicity, I have just concat each of the values that are passed and return back the output.

In the sample application, I have produced the most simple Binding to ensure everything comes from the Model. You can find the sample application from the link at the top of this article.

Bottom of Form

<http://www.certified-easy.com/aa.php?isbn=ISBN:8177227335&name=.NET_Interview_Questions>

WPF Binding

<http://msdn.microsoft.com/en-us/magazine/cc163299.aspx>

<http://www.codeproject.com/KB/WPF/GuidedTourWPF_2.asp>

<http://www.codeproject.com/Articles/18270/A-Guided-Tour-of-WPF-Part-3-Data-binding>