Contents

Data

1/23/2019 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) data binding provides a simple and consistent way for applications to present and interact with data. Elements can be bound to data from a variety of data sources in the form of common language runtime (CLR) objects and XML. Windows Presentation Foundation (WPF) also provides a mechanism for the transfer of data through drag-and-drop operations.

In This Section

Data Binding
Drag and Drop

Reference

System.Windows.Data

Binding

DataTemplate

DataTemplateSelector

Related Sections

Controls

Styling and Templating

Data Binding

See also

- Walkthrough: My first WPF desktop application
- Walkthrough: Caching Application Data in a WPF Application

Data Binding (WPF)

5/4/2018 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) data binding provides a simple and consistent way for applications to present and interact with data. Elements can be bound to data from a variety of data sources in the form of common language runtime (CLR) objects and XML.

In This Section

Data Binding Overview
Binding Sources Overview
Data Templating Overview
Binding Declarations Overview
How-to Topics

Reference

System.Windows.Data

Binding

DataTemplate

DataTemplateSelector

Related Sections

Drag and Drop

Data Binding

Walkthrough: Caching Application Data in a WPF Application

Data Binding Overview

1/23/2019 • 37 minutes to read • Edit Online

Windows Presentation Foundation (WPF) data binding provides a simple and consistent way for applications to present and interact with data. Elements can be bound to data from a variety of data sources in the form of common language runtime (CLR) objects and XML. ContentControls such as Button and ItemsControls such as ListBox and ListView have built-in functionality to enable flexible styling of single data items or collections of data items. Sort, filter, and group views can be generated on top of the data.

The data binding functionality in WPF has several advantages over traditional models, including a broad range of properties that inherently support data binding, flexible UI representation of data, and clean separation of business logic from UI.

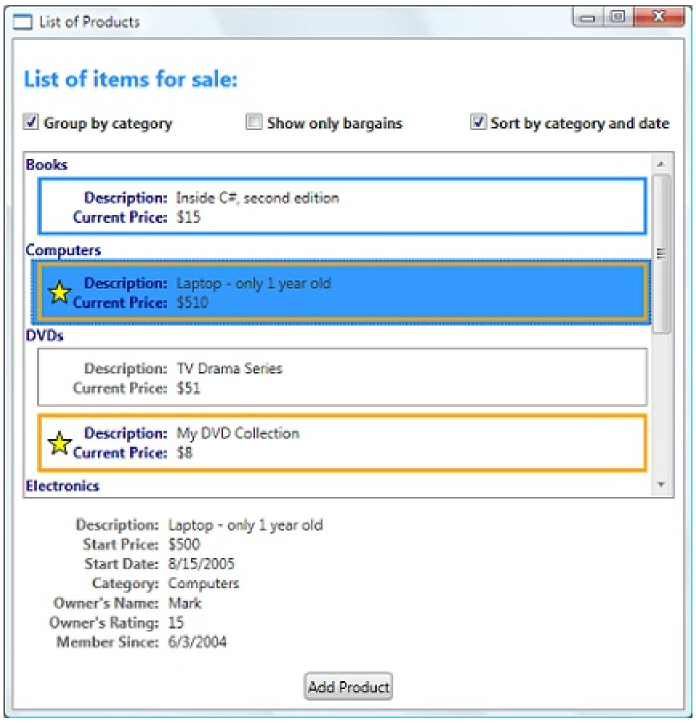
This topic first discusses concepts fundamental to WPF data binding and then goes into the usage of the Binding class and other features of data binding.

What Is Data Binding?

Data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically. Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change.

A typical use of data binding is to place server or local configuration data into forms or other UI controls. In WPF, this concept is expanded to include the binding of a broad range of properties to a variety of data sources. In WPF, dependency properties of elements can be bound to CLR objects (including ADO.NET objects or objects associated with Web Services and Web properties) and XML data.

For an example of data binding, take a look at the following application UI from the Data Binding Demo:



List of items for sale:

[✓] Group by category   [ ] Show only bargains   [✓] Sort by category and date

Books
Description: Inside C#, second edition
Current Price: $15

Computers
Description: Laptop - only 1 year old
Current Price: $510

DVDs
Description: TV Drama Series
Current Price: $51

Description: My DVD Collection
Current Price: $8

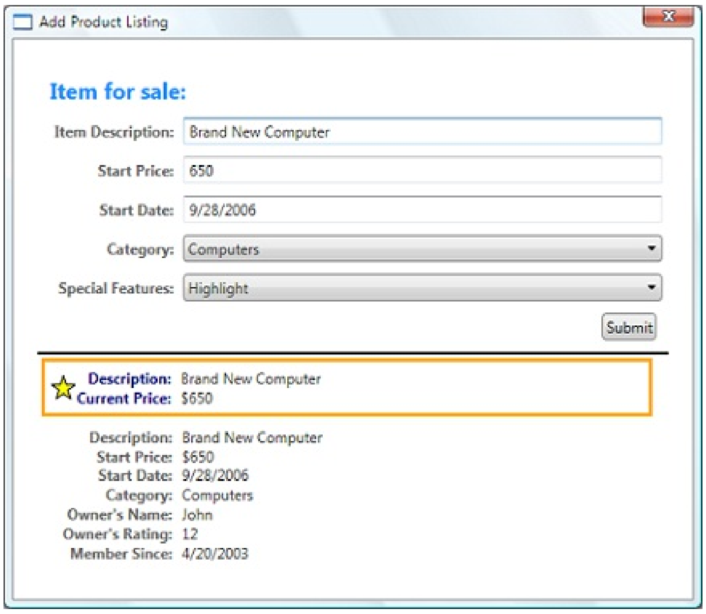
Electronics

Description:	Laptop - only 1 year old
Start Price:	$500
Start Date:	8/15/2005
Category:	Computers
Owner's Name:	Mark
Owner's Rating:	15
Member Since:	6/3/2004

The above is the UI of an application that displays a list of auction items. The application demonstrates the following features of data binding:

- The content of the ListBox is bound to a collection of *AuctionItem* objects. An *AuctionItem* object has properties such as *Description*, *StartPrice*, *StartDate*, *Category*, *SpecialFeatures*, etc.
- The data (*AuctionItem* objects) displayed in the ListBox is templated so that the description and the current price are shown for each item. This is done using a DataTemplate. In addition, the appearance of each item depends on the *SpecialFeatures* value of the *AuctionItem* being displayed. If the *SpecialFeatures* value of the *AuctionItem* is *Color*, the item has a blue border. If the value is *Highlight*, the item has an orange border and a star. The Data Templating section provides information about data templating.
- The user can group, filter, or sort the data using the CheckBoxes provided. In the image above, the "Group by category" and "Sort by category and date" CheckBoxes are selected. You may have noticed that the data is grouped based on the category of the product, and the category name is in alphabetical order. It is difficult to notice from the image but the items are also sorted by the start date within each category. This is done using a *collection view*. The Binding to Collections section discusses collection views.
- When the user selects an item, the ContentControl displays the details of the selected item. This is called the *Master-Detail scenario*. The Master-Detail Scenario section provides information about this type of binding scenario.
- The type of the *StartDate* property is DateTime, which returns a date that includes the time to the millisecond. In this application, a custom converter has been used so that a shorter date string is displayed. The Data Conversion section provides information about converters.

When the user clicks the *Add Product* button, the following form comes up:


Add Product Listing

Item for sale:

Item Description:	Brand New Computer
Start Price:	650
Start Date:	9/28/2006
Category:	Computers
Special Features:	Highlight

Description:	Brand New Computer
Current Price:	$650

Description:	Brand New Computer
Start Price:	$650
Start Date:	9/28/2006
Category:	Computers
Owner's Name:	John
Owner's Rating:	12
Member Since:	4/20/2003

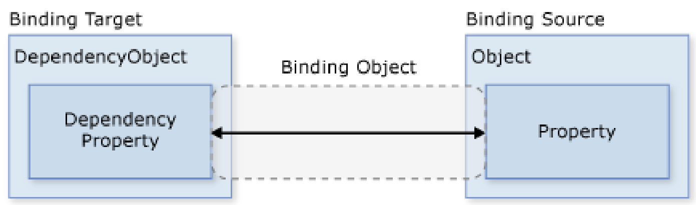
The user can edit the fields in the form, preview the product listing using the short preview and the more detailed preview panes, and then click *submit* to add the new product listing. Any existing grouping, filtering and sorting functionalities will apply to the new entry. In this particular case, the item entered in the above image will be displayed as the second item within the *Computer* category.

Not shown in this image is the validation logic provided in the *Start Date* TextBox. If the user enters an invalid date (invalid formatting or a past date), the user will be notified with a ToolTip and a red exclamation point next to the TextBox. The Data Validation section discusses how to create validation logic.

Before going into the different features of data binding outlined above, we will first discuss in the next section the fundamental concepts that are critical to understanding WPF data binding.

Basic Data Binding Concepts

Regardless of what element you are binding and the nature of your data source, each binding always follows the model illustrated by the following figure:



As illustrated by the above figure, data binding is essentially the bridge between your binding target and your binding source. The figure demonstrates the following fundamental WPF data binding concepts:

- Typically, each binding has these four components: a binding target object, a target property, a binding source, and a path to the value in the binding source to use. For example, if you want to bind the content of a TextBox to the *Name* property of an *Employee* object, your target object is the TextBox, the target property is the Text property, the value to use is *Name*, and the source object is the *Employee* object.
- The target property must be a dependency property. Most UIElement properties are dependency properties and most dependency properties, except read-only ones, support data binding by default. (Only DependencyObject types can define dependency properties and all UIElements derive from DependencyObject.)
- Although not specified in the figure, it should be noted that the binding source object is not restricted to being a custom CLR object. WPF data binding supports data in the form of CLR objects and XML. To provide some examples, your binding source may be a UIElement, any list object, a CLR object that is associated with ADO.NET data or Web Services, or an XmlNode that contains your XML data. For more information, see Binding Sources Overview.

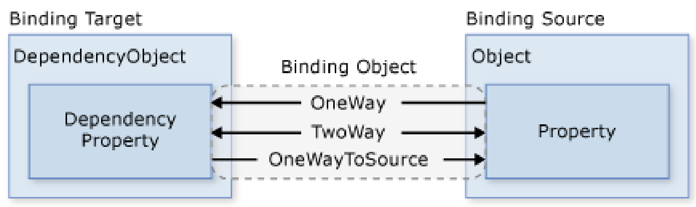
As you read through other software development kit (SDK) topics, it is important to remember that when you are establishing a binding, you are binding a binding target *to* a binding source. For example, if you are displaying some underlying XML data in a ListBox using data binding, you are binding your ListBox to the XML data.

To establish a binding, you use the Binding object. The rest of this topic discusses many of the concepts associated with and some of the properties and usage of the Binding object.

Direction of the Data Flow

As mentioned previously and as indicated by the arrow in the figure above, the data flow of a binding can go from the binding target to the binding source (for example, the source value changes when a user edits the value of a TextBox) and/or from the binding source to the binding target (for example, your TextBox content gets updated with changes in the binding source) if the binding source provides the proper notifications.

You may want your application to enable users to change the data and propagate it back to the source object. Or you may not want to enable users to update the source data. You can control this by setting the Mode property of your Binding object. The following figure illustrates the different types of data flow:



- OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. This type of binding is appropriate if the control being bound is implicitly read-only. For instance, you may bind to a source such as a stock ticker or perhaps your target property has no control interface provided for making changes, such as a data-bound background color of a table. If there is no need to monitor the changes of the target property, using the OneWay binding mode avoids the overhead of the TwoWay binding mode.

- TwoWay binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully-interactive UI scenarios. Most properties default to OneWay binding, but some dependency properties (typically properties of user-editable controls such as the Text property of TextBox and the IsChecked property of CheckBox) default to TwoWay binding. A programmatic way to determine whether a dependency property binds one-way or two-way by default is to get the property metadata of the property using GetMetadata and then check the Boolean value of the BindsTwoWayByDefault property.

- OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes. One example scenario is if you only need to re-evaluate the source value from the UI.

- Not illustrated in the figure is OneTime binding, which causes the source property to initialize the target property, but subsequent changes do not propagate. This means that if the data context undergoes a change or the object in the data context changes, then the change is not reflected in the target property. This type of binding is appropriate if you are using data where either a snapshot of the current state is appropriate to use or the data is truly static. This type of binding is also useful if you want to initialize your target property with some value from a source property and the data context is not known in advance. This is essentially a simpler form of OneWay binding that provides better performance in cases where the source value does not change.

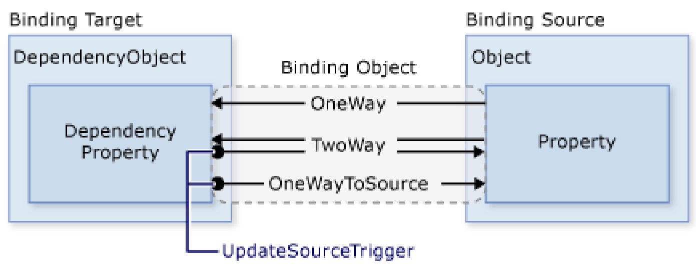
Note that to detect source changes (applicable to OneWay and TwoWay bindings), the source must implement a suitable property change notification mechanism such as INotifyPropertyChanged. See Implement Property Change Notification for an example of an INotifyPropertyChanged implementation.

The Mode property page provides more information about binding modes and an example of how to specify the direction of a binding.

What Triggers Source Updates

Bindings that are TwoWay or OneWayToSource listen for changes in the target property and propagate them back to the source. This is known as updating the source. For example, you may edit the text of a TextBox to change the underlying source value. As described in the last section, the direction of the data flow is determined by the value of the Mode property of the binding.

However, does your source value get updated while you are editing the text or after you finish editing the text and point your mouse away from the TextBox? The UpdateSourceTrigger property of the binding determines what triggers the update of the source. The dots of the right arrows in the following figure illustrate the role of the UpdateSourceTrigger property:



If the UpdateSourceTrigger value is PropertyChanged, then the value pointed to by the right arrow of

TwoWay or the OneWayToSource bindings gets updated as soon as the target property changes. However, if the UpdateSourceTrigger value is LostFocus, then that value only gets updated with the new value when the target property loses focus.

Similar to the Mode property, different dependency properties have different default UpdateSourceTrigger values. The default value for most dependency properties is PropertyChanged, while the Text property has a default value of LostFocus. This means that source updates usually happen whenever the target property changes, which is fine for CheckBoxes and other simple controls. However, for text fields, updating after every keystroke can diminish performance and it denies the user the usual opportunity to backspace and fix typing errors before committing to the new value. That is why the Text property has a default value of LostFocus instead of PropertyChanged.

See the UpdateSourceTrigger property page for information about how to find the default UpdateSourceTrigger value of a dependency property.

The following table provides an example scenario for each UpdateSourceTrigger value using the TextBox as an example:

UPDATESOURCETRIGGER VALUE	WHEN THE SOURCE VALUE GETS UPDATED	EXAMPLE SCENARIO FOR TEXTBOX
LostFocus (default for TextBox.Text)	When the TextBox control loses focus	A TextBox that is associated with validation logic (see Data Validation section)
PropertyChanged	As you type into the TextBox	TextBox controls in a chat room window
Explicit	When the application calls UpdateSource	TextBox controls in an editable form (updates the source values only when the user clicks the submit button)

For an example, see Control When the TextBox Text Updates the Source.

Creating a Binding

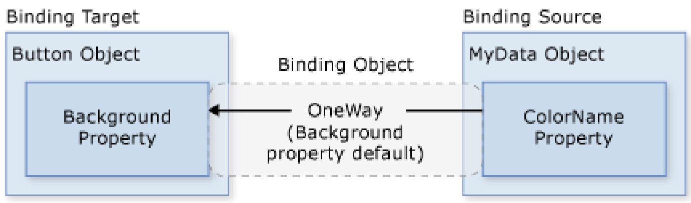
To recapitulate some of the concepts discussed in the previous sections, you establish a binding using the Binding object, and each binding usually has four components: binding target, target property, binding source, and a path to the source value to use. This section discusses how to set up a binding.

Consider the following example, in which the binding source object is a class named *MyData* that is defined in the *SDKSample* namespace. For demonstration purposes, *MyData* class has a string property named *ColorName*, of which the value is set to "Red". Thus, this example generates a button with a red background.

```
<DockPanel
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c="clr-namespace:SDKSample">
  <DockPanel.Resources>
    <c:MyData x:Key="myDataSource"/>
  </DockPanel.Resources>
  <DockPanel.DataContext>
    <Binding Source="{StaticResource myDataSource}"/>
  </DockPanel.DataContext>
  <Button Background="{Binding Path=ColorName}"
          Width="150" Height="30">I am bound to be RED!</Button>
</DockPanel>
```

For more details on the binding declaration syntax and for examples of how to set up a binding in code, see Binding Declarations Overview.

If we apply this example to our basic diagram, the resulting figure looks like the following. This is a OneWay binding because the Background property supports OneWay binding by default.



You may wonder why this works even though the *ColorName* property is of type string while the Background property is of type Brush. This is default type conversion at work and is discussed in the Data Conversion section.

Specifying the Binding Source

Notice that in the previous example, the binding source is specified by setting the DataContext property on the DockPanel element. The Button then inherits the DataContext value from the DockPanel, which is its parent element. To reiterate, the binding source object is one of the four necessary components of a binding. Therefore, without the binding source object being specified, the binding would do nothing.

There are several ways to specify the binding source object. Using the DataContext property on a parent element is useful when you are binding multiple properties to the same source. However, sometimes it may be more appropriate to specify the binding source on individual binding declarations. For the previous example, instead of using the DataContext property, you can specify the binding source by setting the Source property directly on the binding declaration of the button, as in the following example:

```
<DockPanel.Resources>
  <c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<Button Width="150" Height="30"
        Background="{Binding Source={StaticResource myDataSource},
                             Path=ColorName}">I am bound to be RED!</Button>
```

Other than setting the DataContext property on an element directly, inheriting the DataContext value from an ancestor (such as the button in the first example), and explicitly specifying the binding source by setting the Source property on the Binding (such as the button the last example), you can also use the ElementName property or the RelativeSource property to specify the binding source. The ElementName property is useful when you are binding to other elements in your application, such as when you are using a slider to adjust the width of a button. The RelativeSource property is useful when the binding is specified in a ControlTemplate or a Style. For more information, see Specify the Binding Source.

Specifying the Path to the Value

If your binding source is an object, you use the Path property to specify the value to use for your binding. If you are binding to XML data, you use the XPath property to specify the value. In some cases, it may be applicable to use the Path property even when your data is XML. For example, if you want to access the Name property of a returned XmlNode (as a result of an XPath query), you should use the Path property in addition to the XPath property.

For syntax information and examples, see the Path and XPath property pages.

Note that although we have emphasized that the Path to the value to use is one of the four necessary components of a binding, in the scenarios which you want to bind to an entire object, the value to use would be the same as the binding source object. In those cases, it is applicable to not specify a Path. Consider the following example:

```
<ListBox ItemsSource="{Binding}"
         IsSynchronizedWithCurrentItem="true"/>
```

The above example uses the empty binding syntax: {Binding}. In this case, the ListBox inherits the DataContext from a parent DockPanel element (not shown in this example). When the path is not specified, the default is to bind to the entire object. In other words, in this example, the path has been left out because we are binding the ItemsSource property to the entire object. (See the Binding to Collections section for an in-depth discussion.)

Other than binding to a collection, this scenario is also useful when you want to bind to an entire object instead of just a single property of an object. For example, if your source object is of type string and you simply want to bind to the string itself. Another common scenario is when you want to bind an element to an object with several properties.

Note that you may need to apply custom logic so that the data is meaningful to your bound target property. The custom logic may be in the form of a custom converter (if default type conversion does not exist). See Data Conversion for information about converters.

Binding and BindingExpression

Before getting into other features and usages of data binding, it would be useful to introduce the BindingExpression class. As you have seen in previous sections, the Binding class is the high-level class for the declaration of a binding; the Binding class provides many properties that allow you to specify the characteristics of a binding. A related class, BindingExpression, is the underlying object that maintains the connection between the source and the target. A binding contains all the information that can be shared across several binding expressions. A BindingExpression is an instance expression that cannot be shared and contains all the instance information of the Binding.

For example, consider the following, where *myDataObject* is an instance of *MyData* class, *myBinding* is the source Binding object, and *MyData* class is a defined class that contains a string property named *MyDataProperty*. This example binds the text content of *mytext*, an instance of TextBlock, to *MyDataProperty*.

```
// Make a new source.
MyData myDataObject = new MyData(DateTime.Now);
Binding myBinding = new Binding("MyDataProperty");
myBinding.Source = myDataObject;
// Bind the new data source to the myText TextBlock control's Text dependency property.
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

```
' Make a new source.
Dim data1 As New MyData(DateTime.Now)
Dim binding1 As New Binding("MyDataProperty")
binding1.Source = data1
' Bind the new data source to the myText TextBlock control's Text dependency property.
Me.myText.SetBinding(TextBlock.TextProperty, binding1)
```

You can use the same *myBinding* object to create other bindings. For example, you may use *myBinding* object to bind the text content of a check box to *MyDataProperty*. In that scenario, there will be two instances of BindingExpression sharing the *myBinding* object.

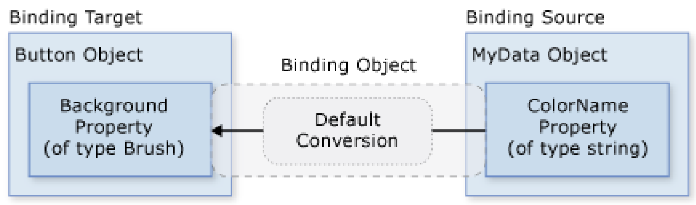
A BindingExpression object can be obtained through the return value of calling GetBindingExpression on a data-bound object. The following topics demonstrate some of the usages of the BindingExpression class:

- Get the Binding Object from a Bound Target Property
- Control When the TextBox Text Updates the Source

Data Conversion

In the previous example, the button is red because its Background property is bound to a string property with the value "Red". This works because a type converter is present on the Brush type to convert the string value to a Brush.

To add this information to the figure in the Creating a Binding section, the diagram looks like the following:



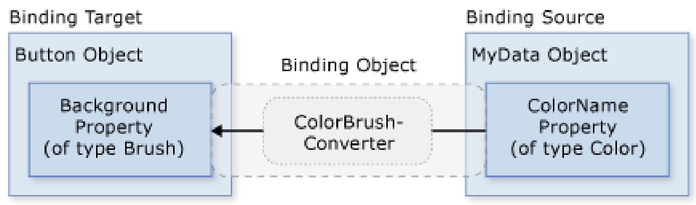
However, what if instead of having a property of type string your binding source object has a *Color* property of type Color? In that case, in order for the binding to work you would need to first turn the *Color* property value into something that the Background property accepts. You would need to create a custom converter by implementing the IValueConverter interface, as in the following example:

```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]
public class ColorBrushConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        return null;
    }
}
```

```
<ValueConversion(GetType(Color), GetType(SolidColorBrush))>
Public Class ColorBrushConverter
    Implements IValueConverter
    Public Function Convert(ByVal value As Object, ByVal targetType As Type, ByVal parameter As Object,
ByVal culture As System.Globalization.CultureInfo) As Object Implements IValueConverter.Convert
        Dim color As Color = CType(value, Color)
        Return New SolidColorBrush(color)
    End Function

    Public Function ConvertBack(ByVal value As Object, ByVal targetType As Type, ByVal parameter As
Object, ByVal culture As System.Globalization.CultureInfo) As Object Implements
IValueConverter.ConvertBack
        Return Nothing
    End Function
End Class
```

The IValueConverter reference page provides more information.

Now the custom converter is used instead of default conversion, and our diagram looks like this:



To reiterate, default conversions may be available because of type converters that are present in the type being bound to. This behavior will depend on which type converters are available in the target. If in doubt, create your own converter.

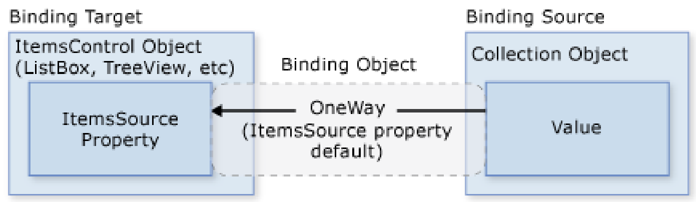
Following are some typical scenarios where it makes sense to implement a data converter:

- Your data should be displayed differently, depending on culture. For instance, you might want to implement a currency converter or a calendar date/time converter based on the values or standards used in a particular culture.
- The data being used is not necessarily intended to change the text value of a property, but is instead intended to change some other value, such as the source for an image, or the color or style of the display text. Converters can be used in this instance by converting the binding of a property that might not seem to be appropriate, such as binding a text field to the Background property of a table cell.
- More than one control or to multiple properties of controls are bound to the same data. In this case, the primary binding might just display the text, whereas other bindings handle specific display issues but still use the same binding as source information.
- So far we have not yet discussed MultiBinding, where a target property has a collection of bindings. In the case of a MultiBinding, you use a custom IMultiValueConverter to produce a final value from the values of the bindings. For example, color may be computed from red, blue, and green values, which can be values from the same or different binding source objects. See the MultiBinding class page for examples and information.

Binding to Collections

A binding source object can be treated either as a single object of which the properties contain data or as a data collection of polymorphic objects that are often grouped together (such as the result of a query to a database). So far we've only discussed binding to single objects, however, binding to a data collection is a common scenario. For example, a common scenario is to use an ItemsControl such as a ListBox, ListView, or TreeView to display a data collection, such as in the application shown in the What Is Data Binding? section.

Fortunately, our basic diagram still applies. If you are binding an ItemsControl to a collection, the diagram looks like this:



As shown in this diagram, to bind an ItemsControl to a collection object, ItemsSource property is the property to use. You can think of ItemsSource property as the content of the ItemsControl. Note that the binding is OneWay because the ItemsSource property supports OneWay binding by default.

How to Implement Collections

You can enumerate over any collection that implements the IEnumerable interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the INotifyCollectionChanged interface. This interface exposes an event that should be raised whenever the underlying collection changes.

WPF provides the ObservableCollection<T> class, which is a built-in implementation of a data collection that exposes the INotifyCollectionChanged interface. Note that to fully support transferring data values from source objects to targets, each object in your collection that supports bindable properties must also implement the INotifyPropertyChanged interface. For more information, see Binding Sources Overview.

Before implementing your own collection, consider using ObservableCollection<T> or one of the existing collection classes, such as List<T>, Collection<T>, and BindingList<T>, among many others. If you have an advanced scenario and want to implement your own collection, consider using IList, which provides a non-generic collection of objects that can be individually accessed by index and thus the best performance.

Collection Views

Once your ItemsControl is bound to a data collection, you may want to sort, filter, or group the data. To do that, you use collection views, which are classes that implement the ICollectionView interface.

What Are Collection Views?

A collection view is a layer on top of a binding source collection that allows you to navigate and display the source collection based on sort, filter, and group queries, without having to change the underlying source collection itself. A collection view also maintains a pointer to the current item in the collection. If the source collection implements the INotifyCollectionChanged interface, the changes raised by the CollectionChanged event are propagated to the views.

Because views do not change the underlying source collections, each source collection can have multiple views associated with it. For example, you may have a collection of *Task* objects. With the use of views, you can display that same data in different ways. For example, on the left side of your page you may want to show tasks sorted by priority, and on the right side, grouped by area.

How to Create a View

One way to create and use a view is to instantiate the view object directly and then use it as the binding source. For example, consider the Data Binding Demo application shown in the What Is Data Binding? section. The application is implemented such that the ListBox binds to a view over the data collection instead of the data collection directly. The following example is extracted from the Data Binding Demo application. The CollectionViewSource class is the Extensible Application Markup Language (XAML) proxy of a class that inherits from CollectionView. In this particular example, the Source of the view is bound to the *AuctionItems* collection (of type ObservableCollection<T>) of the current application object.

```
<Window.Resources>
```

```
<CollectionViewSource
      Source="{Binding Source={x:Static Application.Current}, Path=AuctionItems}"
      x:Key="listingDataView" />
```

```
</Window.Resources>
```

The resource *listingDataView* then serves as the binding source for elements in the application, such as the ListBox:

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
    ItemsSource="{Binding Source={StaticResource listingDataView}}">
```

```
</ListBox>
```

To create another view for the same collection, you can create another CollectionViewSource instance and give it a different `x:Key` name.

The following table shows which view data types are created as the default collection view or by CollectionViewSource based on the source collection type.

SOURCE COLLECTION TYPE	COLLECTION VIEW TYPE	NOTES
IEnumerable	An internal type based on CollectionView	Cannot group items.
IList	ListCollectionView	Fastest.
IBindingList	BindingListCollectionView	

Using a Default View

Specifying a collection view as a binding source is one way to create and use a collection view. WPF also creates a default collection view for every collection used as a binding source. If you bind directly to a collection, WPF binds to its default view. Note that this default view is shared by all bindings to the same collection, so a change made to a default view by one bound control or code (such as sorting or a change to the current item pointer, discussed later) is reflected in all other bindings to the same collection.

To get the default view, you use the GetDefaultView method. For an example, see Get the Default View of a Data Collection.

Collection Views with ADO.NET DataTables

To improve performance, collection views for ADO.NET DataTable or DataView objects delegate sorting and filtering to the DataView. This causes sorting and filtering to be shared across all collection views of the data source. To enable each collection view to sort and filter independently, initialize each collection view with its own DataView object.

Sorting

As mentioned before, views can apply a sort order to a collection. As it exists in the underlying collection, your data may or may not have a relevant, inherent order. The view over the collection allows you to impose an order, or change the default order, based on comparison criteria that you supply. Because it is a client-based view of the data, a common scenario is that the user might want to sort columns of tabular data per the value that the column corresponds to. Using views, this user-driven sort can be applied, again without making any changes to the underlying collection or even having to requery for the collection content. For an example, see Sort a GridView Column When a Header Is Clicked.

The following example shows the sorting logic of the "Sort by category and date" CheckBox of the application UI in the What Is Data Binding? section:

```
private void AddSorting(object sender, RoutedEventArgs args)
{
    // This sorts the items first by Category and within each Category,
    // by StartDate. Notice that because Category is an enumeration,
    // the order of the items is the same as in the enumeration declaration
    listingDataView.SortDescriptions.Add(
        new SortDescription("Category", ListSortDirection.Ascending));
    listingDataView.SortDescriptions.Add(
        new SortDescription("StartDate", ListSortDirection.Ascending));
}
```

```
Private Sub AddSorting(ByVal sender As Object, ByVal args As RoutedEventArgs)
    'This sorts the items first by Category and within each Category, by StartDate
    'Notice that because Category is an enumeration, the order of the items is the same as in the
    'enumeration declaration
    listingDataView.SortDescriptions.Add(New SortDescription("Category", ListSortDirection.Ascending))
    listingDataView.SortDescriptions.Add(New SortDescription("StartDate", ListSortDirection.Ascending))
End Sub
```

Filtering

Views can also apply a filter to a collection. This means that although an item might exist in the collection, this particular view is intended to show only a certain subset of the full collection. You might filter on a condition in the data. For instance, as is done by the application in the What Is Data Binding? section, the "Show only bargains" CheckBox contains logic to filter out items that cost $25 or more. The following code is executed to set *ShowOnlyBargainsFilter* as the Filter event handler when that CheckBox is selected:

```
listingDataView.Filter += new FilterEventHandler(ShowOnlyBargainsFilter);
```

```
AddHandler listingDataView.Filter, AddressOf ShowOnlyBargainsFilter
```

The *ShowOnlyBargainsFilter* event handler has the following implementation:

```
private void ShowOnlyBargainsFilter(object sender, FilterEventArgs e)
{
    AuctionItem product = e.Item as AuctionItem;
    if (product != null)
    {
        // Filter out products with price 25 or above
        if (product.CurrentPrice < 25)
        {
            e.Accepted = true;
        }
        else
        {
            e.Accepted = false;
        }
    }
}
```

```
Private Sub ShowOnlyBargainsFilter(ByVal sender As Object, ByVal e As FilterEventArgs)
    Dim product As AuctionItem = CType(e.Item, AuctionItem)
    If Not (product Is Nothing) Then
        'Filter out products with price 25 or above
        If product.CurrentPrice < 25 Then
            e.Accepted = True
        Else
            e.Accepted = False
        End If
    End If
End Sub
```

If you are using one of the CollectionView classes directly instead of CollectionViewSource, you would use the Filter property to specify a callback. For an example, see Filter Data in a View.

Grouping

Except for the internal class that views an IEnumerable collection, all collection views support the functionality of grouping, which allows the user to partition the collection in the collection view into logical groups. The groups can be explicit, where the user supplies a list of groups, or implicit, where the groups are generated dynamically depending on the data.

The following example shows the logic of the "Group by category" CheckBox:

```
// This groups the items in the view by the property "Category"
PropertyGroupDescription groupDescription = new PropertyGroupDescription();
groupDescription.PropertyName = "Category";
listingDataView.GroupDescriptions.Add(groupDescription);
```

```
'This groups by property "Category"
Dim groupDescription As PropertyGroupDescription = New PropertyGroupDescription
groupDescription.PropertyName = "Category"
listingDataView.GroupDescriptions.Add(groupDescription)
```

For another grouping example, see Group Items in a ListView That Implements a GridView.

Current Item Pointers

Views also support the notion of a current item. You can navigate through the objects in a collection view. As you navigate, you are moving an item pointer that allows you to retrieve the object that exists at that particular location in the collection. For an example, see Navigate Through the Objects in a Data CollectionView.

Because WPF binds to a collection only by using a view (either a view you specify, or the collection's default view), all bindings to collections have a current item pointer. When binding to a view, the slash ("/") character in a `Path` value designates the current item of the view. In the following example, the data context is a collection view. The first line binds to the collection. The second line binds to the current item in the collection. The third line binds to the `Description` property of the current item in the collection.

```
<Button Content="{Binding }" />
<Button Content="{Binding Path=/}" />
<Button Content="{Binding Path=/Description}" />
```

The slash and property syntax can also be stacked to traverse a hierarchy of collections. The following example binds to the current item of a collection named `Offices`, which is a property of the current item of the source collection.

```
<Button Content="{Binding /Offices/}" />
```

The current item pointer can be affected by any sorting or filtering that is applied to the collection. Sorting preserves the current item pointer on the last item selected, but the collection view is now restructured around it. (Perhaps the selected item was at the beginning of the list before, but now the selected item might be somewhere in the middle.) Filtering preserves the selected item if that selection remains in view after the filtering. Otherwise, the current item pointer is set to the first item of the filtered collection view.

Master-Detail Binding Scenario

The notion of a current item is useful not only for navigation of items in a collection, but also for the master-detail binding scenario. Consider the application UI in the What Is Data Binding? section again. In that application, the selection within the ListBox determines the content shown in the ContentControl. To put it in another way, when a ListBox item is selected, the ContentControl shows the details of the selected item.

You can implement the master-detail scenario simply by having two or more controls bound to the same view. The following example from the Data Binding Demo shows the markup of the ListBox and the ContentControl you see on the application UI in the What Is Data Binding? section:

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
    ItemsSource="{Binding Source={StaticResource listingDataView}}">
```

```
</ListBox>
```

```
<ContentControl Name="Detail" Grid.Row="3" Grid.ColumnSpan="3"
        Content="{Binding Source={StaticResource listingDataView}}"
        ContentTemplate="{StaticResource detailsProductListingTemplate}"
        Margin="9,0,0,0"/>
```

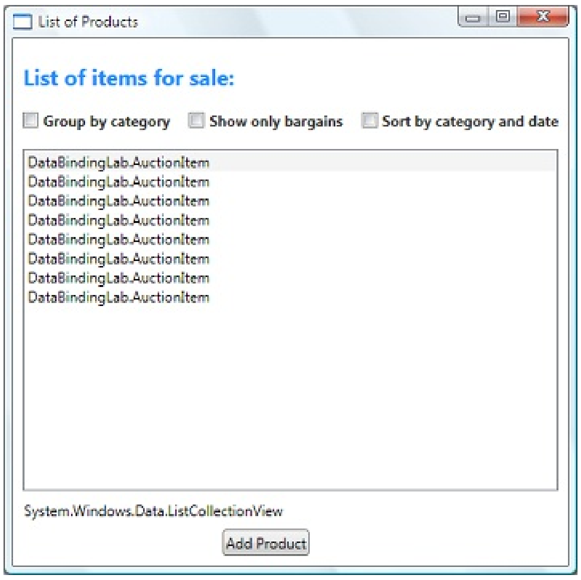
Notice that both of the controls are bound to the same source, the *listingDataView* static resource (see the definition of this resource in the How to Create a View section). This works because when a singleton object (the ContentControl in this case) is bound to a collection view, it automatically binds to the CurrentItem of the view. Note that CollectionViewSource objects automatically synchronize currency and selection. If your list control is not bound to a CollectionViewSource object as in this example, then you would need to set its IsSynchronizedWithCurrentItem property to `true` for this to work.

For other examples, see Bind to a Collection and Display Information Based on Selection and Use the Master-Detail Pattern with Hierarchical Data.

You may have noticed that the above example uses a template. In fact, the data would not be displayed the way we wish without the use of templates (the one explicitly used by the ContentControl and the one implicitly used by the ListBox). We now turn to data templating in the next section.

Data Templating

Without the use of data templates, our application UI in the What Is Data Binding? section would look like the following:



DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem
DataBindingLab.AuctionItem

System.Windows.Data.ListCollectionView

As shown in the example in the previous section, both the ListBox control and the ContentControl are bound to the entire collection object (or more specifically, the view over the collection object) of *AuctionItem*s. Without specific instructions of how to display the data collection, the ListBox is displaying a string representation of each object in the underlying collection and the ContentControl is displaying a string representation of the object it is bound to.

To solve that problem, the application defines DataTemplates. As shown in the example in the previous section, the ContentControl explicitly uses the *detailsProductListingTemplate*DataTemplate. The ListBox control implicitly uses the following DataTemplate when displaying the *AuctionItem* objects in the collection:

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
    <Border BorderThickness="1" BorderBrush="Gray"
            Padding="7" Name="border" Margin="3" Width="500">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
          </Grid.RowDefinitions>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="20"/>
            <ColumnDefinition Width="86"/>
            <ColumnDefinition Width="*"/>
          </Grid.ColumnDefinitions>

            <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
                     Fill="Yellow" Stroke="Black" StrokeThickness="1"
                     StrokeLineJoin="Round" Width="20" Height="20"
                     Stretch="Fill"
                     Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7 7,7"
                     Visibility="Hidden" Name="star"/>

            <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
                       Name="descriptionTitle"
                       Style="{StaticResource smallTitleStyle}">Description:</TextBlock>
            <TextBlock Name="DescriptionDTDataType" Grid.Row="0" Grid.Column="2"
                Text="{Binding Path=Description}"
                Style="{StaticResource textStyleTextBlock}"/>

            <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
                       Name="currentPriceTitle"
                       Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>
            <StackPanel Grid.Row="1" Grid.Column="2" Orientation="Horizontal">
                <TextBlock Text="$" Style="{StaticResource textStyleTextBlock}"/>
                <TextBlock Name="CurrentPriceDTDataType"
                    Text="{Binding Path=CurrentPrice}"
                    Style="{StaticResource textStyleTextBlock}"/>
            </StackPanel>
        </Grid>
    </Border>
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Path=SpecialFeatures}">
            <DataTrigger.Value>
                <src:SpecialFeatures>Color</src:SpecialFeatures>
            </DataTrigger.Value>
          <DataTrigger.Setters>
            <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
            <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
            <Setter Property="BorderThickness" Value="3" TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
          </DataTrigger.Setters>
        </DataTrigger>
        <DataTrigger Binding="{Binding Path=SpecialFeatures}">
            <DataTrigger.Value>
                <src:SpecialFeatures>Highlight</src:SpecialFeatures>
            </DataTrigger.Value>
            <Setter Property="BorderBrush" Value="Orange" TargetName="border" />
            <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
            <Setter Property="Visibility" Value="Visible" TargetName="star" />
            <Setter Property="BorderThickness" Value="3" TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger>
    </DataTemplate.Triggers>
```

```
</DataTemplate>
```

With the use of those two DataTemplates, the resulting UI is the one shown in the What Is Data Binding? section. As you can see from that screenshot, in addition to letting you place data in your controls, DataTemplates allow you to define compelling visuals for your data. For example, DataTriggers are used in the above DataTemplate so that *AuctionItems* with *SpecialFeatures* value of *HighLight* would be displayed with an orange border and a star.

For more information about data templates, see the Data Templating Overview.

Data Validation

Most applications that take user input need to have validation logic to ensure that the user has entered the expected information. The validation checks can be based on type, range, format, or other application-specific requirements. This section discusses how data validation works in the WPF.

Associating Validation Rules with a Binding

The WPF data binding model allows you to associate ValidationRules with your Binding object. For example, the following example binds a TextBox to a property named `StartPrice` and adds a ExceptionValidationRule object to the Binding.ValidationRules property.

```
<TextBox Name="StartPriceEntryForm" Grid.Row="2" Grid.Column="1"
    Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5">
  <TextBox.Text>
    <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

A ValidationRule object checks whether the value of a property is valid. WPF has the following two types of built-in ValidationRule objects:

- A ExceptionValidationRule checks for exceptions thrown during the update of the binding source property. In the previous example, `StartPrice` is of type integer. When the user enters a value that cannot be converted to an integer, an exception is thrown, causing the binding to be marked as invalid. An alternative syntax to setting the ExceptionValidationRule explicitly is to set the ValidatesOnExceptions property to `true` on your Binding or MultiBinding object.

- A DataErrorValidationRule object checks for errors that are raised by objects that implement the IDataErrorInfo interface. For an example of using this validation rule, see DataErrorValidationRule. An alternative syntax to setting the DataErrorValidationRule explicitly is to set the ValidatesOnDataErrors property to `true` on your Binding or MultiBinding object.

You can also create your own validation rule by deriving from the ValidationRule class and implementing the Validate method. The following example shows the rule used by the *Add Product Listing* "Start Date" TextBox from the What Is Data Binding? section:

```
class FutureDateRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        DateTime date;
        try
        {
            date = DateTime.Parse(value.ToString());
        }
        catch (FormatException)
        {
            return new ValidationResult(false, "Value is not a valid date.");
        }
        if (DateTime.Now.Date > date)
        {
            return new ValidationResult(false, "Please enter a date in the future.");
        }
        else
        {
            return ValidationResult.ValidResult;
        }
    }
}
```

```
Public Class FutureDateRule
    Inherits ValidationRule

    Public Overrides Function Validate(ByVal value As Object,
                                       ByVal cultureInfo As System.Globalization.CultureInfo) _
                                   As System.Windows.Controls.ValidationResult

        Dim DateVal As DateTime

        Try
            DateVal = DateTime.Parse(value.ToString)
        Catch ex As FormatException
            Return New ValidationResult(False, "Value is not a valid date.")
        End Try

        If DateTime.Now.Date > DateVal Then
            Return New ValidationResult(False, "Please enter a date in the future.")
        Else
            Return ValidationResult.ValidResult
        End If
    End Function
End Class
```

The *StartDateEntryForm* TextBox uses this *FutureDateRule*, as shown in the following example:

```
<TextBox Name="StartDateEntryForm" Grid.Row="3" Grid.Column="1"
    Validation.ErrorTemplate="{StaticResource validationTemplate}"
    Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5">
    <TextBox.Text>
        <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged"
            Converter="{StaticResource dateConverter}" >
            <Binding.ValidationRules>
                <src:FutureDateRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Note that because the UpdateSourceTrigger value is PropertyChanged, the binding engine updates the source value on every keystroke, which means it also checks every rule in the ValidationRules collection on every keystroke. We discuss this further in the Validation Process section.

Providing Visual Feedback

If the user enters an invalid value, you may want to provide some feedback about the error on the application UI. One way to provide such feedback is to set the Validation.ErrorTemplate attached property to a custom ControlTemplate. As shown in the previous subsection, the *StartDateEntryForm* TextBox uses an ErrorTemplate called *validationTemplate*. The following example shows the definition of *validationTemplate*:

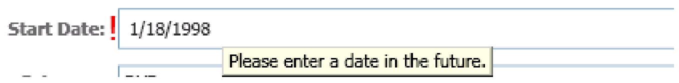
```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

The AdornedElementPlaceholder element specifies where the control being adorned should be placed.

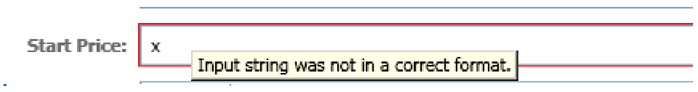
In addition, you may also use a ToolTip to display the error message. Both the *StartDateEntryForm* and the *StartPriceEntryForm*TextBoxes use the style *textStyleTextBox*, which creates a ToolTip that displays the error message. The following example shows the definition of *textStyleTextBox*. The attached property HasError is `true` when one or more of the bindings on the properties of the bound element are in error.

```
<Style x:Key="textStyleTextBox" TargetType="TextBox">
  <Setter Property="Foreground" Value="#333333" />
  <Setter Property="MaxLength" Value="40" />
  <Setter Property="Width" Value="392" />
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={RelativeSource Self},
                        Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

With the custom ErrorTemplate and the ToolTip, the *StartDateEntryForm* TextBox looks like the following when there is a validation error:


Start Date: ! 1/18/1998
Please enter a date in the future.

If your Binding has associated validation rules but you do not specify an ErrorTemplate on the bound control, a default ErrorTemplate will be used to notify users when there is a validation error. The default ErrorTemplate is a control template that defines a red border in the adorner layer. With the default ErrorTemplate and the ToolTip, the UI of the *StartPriceEntryForm* TextBox looks like the following when there is a validation error:


Start Price: x
Input string was not in a correct format.

For an example of how to provide logic to validate all controls in a dialog box, see the Custom Dialog Boxes section in the Dialog Boxes Overview.

Validation Process

Validation usually occurs when the value of a target is transferred to the binding source property. This occurs on TwoWay and OneWayToSource bindings. To reiterate, what causes a source update depends on the value of the UpdateSourceTrigger property, as described in the What Triggers Source Updates section.

The following describes the *validation* process. Note that if a validation error or other type of error occurs at any time during this process, the process is halted.

1. The binding engine checks if there are any custom ValidationRule objects defined whose ValidationStep is set to RawProposedValue for that Binding, in which case it calls the Validate method on each ValidationRule until one of them runs into an error or until all of them pass.

2. The binding engine then calls the converter, if one exists.

3. If the converter succeeds, the binding engine checks if there are any custom ValidationRule objects defined whose ValidationStep is set to ConvertedProposedValue for that Binding, in which case it calls the Validate method on each ValidationRule that has ValidationStep set to ConvertedProposedValue until one of them runs into an error or until all of them pass.

4. The binding engine sets the source property.

5. The binding engine checks if there are any custom ValidationRule objects defined whose ValidationStep is set to UpdatedValue for that Binding, in which case it calls the Validate method on each ValidationRule that has ValidationStep set to UpdatedValue until one of them runs into an error or until all of them pass. If a DataErrorValidationRule is associated with a binding and its ValidationStep is set to the default, UpdatedValue, the DataErrorValidationRule is checked at this point. This is also the point when bindings that have the ValidatesOnDataErrors set to `true` are checked.

6. The binding engine checks if there are any custom ValidationRule objects defined whose ValidationStep is set to CommittedValue for that Binding, in which case it calls the Validate method on each ValidationRule that has ValidationStep set to CommittedValue until one of them runs into an error or until all of them pass.

If a ValidationRule does not pass at any time throughout this process, the binding engine creates a ValidationError object and adds it to the Errors collection of the bound element. Before the binding engine runs the ValidationRule objects at any given step, it removes any ValidationError that was added to the Errors attached property of the bound element during that step. For example, if a ValidationRule whose ValidationStep is set to UpdatedValue failed, the next time the validation process occurs, the binding engine removes that ValidationError immediately before it calls any ValidationRule that has ValidationStep set to UpdatedValue.

When Errors is not empty, the HasError attached property of the element is set to `true`. Also, if the NotifyOnValidationError property of the Binding is set to `true`, then the binding engine raises the Validation.Error attached event on the element.

Also note that a valid value transfer in either direction (target to source or source to target) clears the Errors attached property.

If the binding either has an ExceptionValidationRule associated with it, or had the ValidatesOnExceptions property is set to `true` and an exception is thrown when the binding engine sets the source, the binding engine checks to see if there is a UpdateSourceExceptionFilter. You have the option to use the UpdateSourceExceptionFilter callback to provide a custom handler for handling exceptions. If an UpdateSourceExceptionFilter is not specified on the Binding, the binding engine creates a ValidationError with the exception and adds it to the Errors collection of the bound element.

Debugging Mechanism

You can set the attached property TraceLevel on a binding-related object to receive information about the status of a specific binding.

See also

- DataErrorValidationRule
- What's New in WPF Version 4.5
- Bind to the Results of a LINQ Query
- Data Binding
- Data Binding Demo
- How-to Topics
- Bind to an ADO.NET Data Source

Binding Sources Overview

1/23/2019 • 6 minutes to read • Edit Online

In data binding, the binding source object refers to the object you obtain data from. This topic discusses the types of objects you can use as the binding source.

Binding Source Types

Windows Presentation Foundation (WPF) data binding supports the following binding source types:

BINDING SOURCE	DESCRIPTION
common language runtime (CLR) objects	You can bind to public properties, sub-properties, as well as indexers, of any common language runtime (CLR) object. The binding engine uses CLR reflection to get the values of the properties. Alternatively, objects that implement ICustomTypeDescriptor or have a registered TypeDescriptionProvider also work with the binding engine. For more information about how to implement a class that can serve as a binding source, see Implementing a Class for the Binding Source later in this topic.
dynamic objects	You can bind to available properties and indexers of an object that implements the IDynamicMetaObjectProvider interface. If you can access the member in code, you can bind to it. For example, if a dynamic object enables you to access a member in code via `someObjet.AProperty`, you can bind to it by setting the binding path to `AProperty`.
ADO.NET objects	You can bind to ADO.NET objects, such as DataTable. The ADO.NET DataView implements the IBindingList interface, which provides change notifications that the binding engine listens for.
XML objects	You can bind to and run `XPath` queries on an XmlNode, XmlDocument, or XmlElement. A convenient way to access XML data that is the binding source in markup is to use an XmlDataProvider object. For more information, see Bind to XML Data Using an XMLDataProvider and XPath Queries. You can also bind to an XElement or XDocument, or bind to the results of queries run on objects of these types by using LINQ to XML. A convenient way to use LINQ to XML to access XML data that is the binding source in markup is to use an ObjectDataProvider object. For more information, see Bind to XDocument, XElement, or LINQ for XML Query Results.
DependencyObject objects	You can bind to dependency properties of any DependencyObject. For an example, see Bind the Properties of Two Controls.

Implementing a Class for the Binding Source

You can create your own binding sources. This section discusses the things you need to know if you are implementing a class to serve as a binding source.

Providing Change Notifications

If you are using either OneWay or TwoWay binding (because you want your UI to update when the binding source properties change dynamically), you must implement a suitable property changed notification mechanism. The recommended mechanism is for the CLR or dynamic class to implement the INotifyPropertyChanged interface. For more information, see Implement Property Change Notification.

If you create a CLR object that does not implement INotifyPropertyChanged, then you must arrange for your own notification system to make sure that the data used in a binding stays current. You can provide change notifications by supporting the `PropertyChanged` pattern for each property that you want change notifications for. To support this pattern, you define a *PropertyName*Changed event for each property, where *PropertyName* is the name of the property. You raise the event every time the property changes.

If your binding source implements one of these notification mechanisms, target updates happen automatically. If for any reason your binding source does not provide the proper property changed notifications, you have the option to use the UpdateTarget method to update the target property explicitly.

Other Characteristics

The following list provides other important points to note:

- If you want to create the object in XAML, the class must have a default constructor. In some .NET languages, such as C#, the default constructor might be created for you.

- The properties you use as binding source properties for a binding must be public properties of your class. Explicitly defined interface properties cannot be accessed for binding purposes, nor can protected, private, internal, or virtual properties that have no base implementation.

- You cannot bind to public fields.

- The type of the property declared in your class is the type that is passed to the binding. However, the type ultimately used by the binding depends on the type of the binding target property, not of the binding source property. If there is a difference in type, you might want to write a converter to handle how your custom property is initially passed to the binding. For more information, see IValueConverter.

Using Entire Objects as a Binding Source

You can use an entire object as a binding source. You can specify a binding source by using the Source or the DataContext property, and then provide a blank binding declaration: `{Binding}`. Scenarios in which this is useful include binding to objects that are of type string, binding to objects with multiple properties you are interested in, or binding to collection objects. For an example of binding to an entire collection object, see Use the Master-Detail Pattern with Hierarchical Data.

Note that you may need to apply custom logic so that the data is meaningful to your bound target property. The custom logic may be in the form of a custom converter (if default type conversion does not exist) or a DataTemplate. For more information about converters, see the Data Conversion section of Data Binding Overview. For more information about data templates, see Data Templating Overview.

Using Collection Objects as a Binding Source

Often, the object you want to use as the binding source is a collection of custom objects. Each object serves as the source for one instance of a repeated binding. For example, you might have a `CustomerOrders` collection that consists of `CustomerOrder` objects, where your application iterates over the collection to determine how many orders exist and the data contained in each.

You can enumerate over any collection that implements the IEnumerable interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the INotifyCollectionChanged interface. This interface exposes an event that must be raised whenever the underlying collection changes.

The ObservableCollection<T> class is a built-in implementation of a data collection that exposes the INotifyCollectionChanged interface. The individual data objects within the collection must satisfy the requirements described in the preceding sections. For an example, see Create and Bind to an ObservableCollection. Before implementing your own collection, consider using ObservableCollection<T> or one of the existing collection classes, such as List<T>, Collection<T>, and BindingList<T>, among many others.

WPF never binds directly to a collection. If you specify a collection as a binding source, WPF actually binds to the collection's default view. For information about default views, see Data Binding Overview.

If you have an advanced scenario and you want to implement your own collection, consider using the IList interface. IList provides a non-generic collection of objects that can be individually accessed by index, which can improve performance.

Permission Requirements in Data Binding

When data binding, you must consider the trust level of the application. The following table summarizes what property types can be bound to in an application that is executing in either full trust or partial trust:

PROPERTY TYPE (ALL ACCESS MODIFIERS)	DYNAMIC OBJECT PROPERTY	DYNAMIC OBJECT PROPERTY	CLR PROPERTY	CLR PROPERTY	DEPENDENCY PROPERTY	DEPENDENCY PROPERTY
Trust level	**Full trust**	**Partial trust**	**Full trust**	**Partial trust**	**Full trust**	**Partial trust**
Public class	Yes	Yes	Yes	Yes	Yes	Yes
Non-public class	Yes	No	Yes	No	Yes	Yes

This table describes the following important points about permission requirements in data binding:

- For CLR properties, data binding works as long as the binding engine is able to access the binding source property using reflection. Otherwise, the binding engine issues a warning that the property cannot be found and uses the fallback value or the default value, if it is available.
- You can bind to properties on dynamic objects that are defined at compile time or run time.
- You can always bind to dependency properties.

The permission requirement for XML binding is similar. In a partial-trust sandbox, XmlDataProvider fails when it does not have permissions to access the specified data.

Objects with an anonymous type are internal. You can bind to properties of anonymous types only when running in full trust. For more information about anonymous types, see Anonymous Types (C# Programming Guide) or Anonymous Types (Visual Basic) (Visual Basic).

For more information about partial-trust security, see WPF Partial Trust Security.

See also

- ObjectDataProvider

- XmlDataProvider
- Specify the Binding Source
- Data Binding Overview
- How-to Topics
- WPF Data Binding with LINQ to XML Overview
- Data Binding

Data Templating Overview

1/23/2019 • 12 minutes to read • Edit Online

The WPF data templating model provides you with great flexibility to define the presentation of your data. WPF controls have built-in functionality to support the customization of data presentation. This topic first demonstrates how to define a DataTemplate and then introduces other data templating features, such as the selection of templates based on custom logic and the support for the display of hierarchical data.

Prerequisites

This topic focuses on data templating features and is not an introduction of data binding concepts. For information about basic data binding concepts, see the Data Binding Overview.

DataTemplate is about the presentation of data and is one of the many features provided by the WPF styling and templating model. For an introduction of the WPF styling and templating model, such as how to use a Style to set properties on controls, see the Styling and Templating topic.

In addition, it is important to understand `Resources`, which are essentially what enable objects such as Style and DataTemplate to be reusable. For more information on resources, see XAML Resources.

Data Templating Basics

To demonstrate why DataTemplate is important, let's walk through a data binding example. In this example, we have a ListBox that is bound to a list of `Task` objects. Each `Task` object has a `TaskName` (string), a `Description` (string), a `Priority` (int), and a property of type `TaskType`, which is an `Enum` with values `Home` and `Work`.

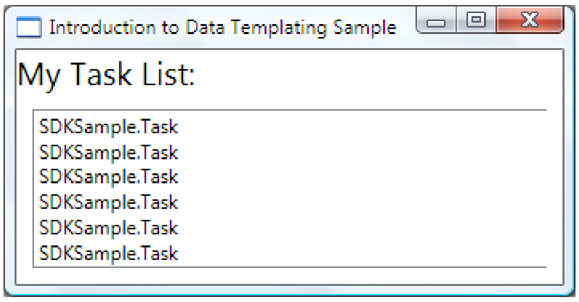
```
<Window x:Class="SDKSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:SDKSample"
  Title="Introduction to Data Templating Sample">
  <Window.Resources>
    <local:Tasks x:Key="myTodoList"/>
```

```
</Window.Resources>
  <StackPanel>
    <TextBlock Name="blah" FontSize="20" Text="My Task List:"/>
    <ListBox Width="400" Margin="10"
             ItemsSource="{Binding Source={StaticResource myTodoList}}"/>
```

```
  </StackPanel>
</Window>
```

Without a DataTemplate

Without a DataTemplate, our ListBox currently looks like this:



My Task List:

```
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
```

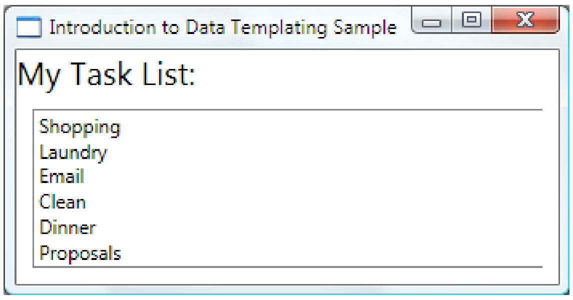
What's happening is that without any specific instructions, the ListBox by default calls `ToString` when trying to display the objects in the collection. Therefore, if the `Task` object overrides the `ToString` method, then the ListBox displays the string representation of each source object in the underlying collection.

For example, if the `Task` class overrides the `ToString` method this way, where `name` is the field for the `TaskName` property:

```
public override string ToString()
{
    return name.ToString();
}
```

```
Public Overrides Function ToString() As String
    Return _name.ToString()
End Function
```

Then the ListBox looks like the following:



My Task List:

```
Shopping
Laundry
Email
Clean
Dinner
Proposals
```

However, that is limiting and inflexible. Also, if you are binding to XML data, you wouldn't be able to override `ToString`.

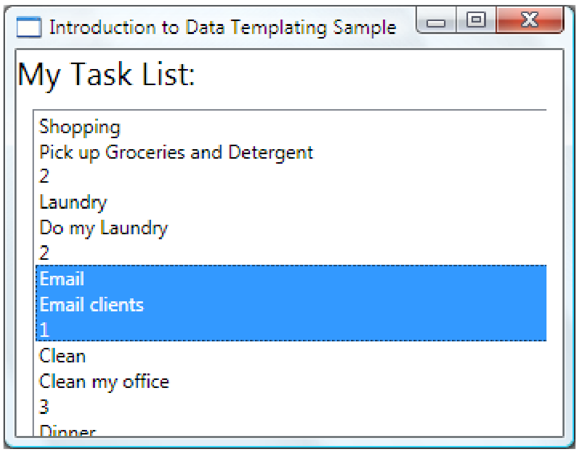
Defining a Simple DataTemplate

The solution is to define a DataTemplate. One way to do that is to set the ItemTemplate property of the ListBox to a DataTemplate. What you specify in your DataTemplate becomes the visual structure of your data object. The following DataTemplate is fairly simple. We are giving instructions that each item appears as three TextBlock elements within a StackPanel. Each TextBlock element is bound to a property of the `Task` class.

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}">
   <ListBox.ItemTemplate>
     <DataTemplate>
       <StackPanel>
         <TextBlock Text="{Binding Path=TaskName}" />
         <TextBlock Text="{Binding Path=Description}"/>
         <TextBlock Text="{Binding Path=Priority}"/>
       </StackPanel>
     </DataTemplate>
   </ListBox.ItemTemplate>
 </ListBox>
```

The underlying data for the examples in this topic is a collection of CLR objects. If you are binding to XML data, the fundamental concepts are the same, but there is a slight syntactic difference. For example, instead of having `Path=TaskName`, you would set XPath to `@TaskName` (if `TaskName` is an attribute of your XML node).

Now our ListBox looks like the following:



My Task List:

Shopping
Pick up Groceries and Detergent
2
Laundry
Do my Laundry
2
Email
Email clients
1
Clean
Clean my office
3
Dinner

Creating the DataTemplate as a Resource

In the above example, we defined the DataTemplate inline. It is more common to define it in the resources section so it can be a reusable object, as in the following example:

```
<Window.Resources>
```

```
<DataTemplate x:Key="myTaskTemplate">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}"/>
    <TextBlock Text="{Binding Path=Priority}"/>
  </StackPanel>
</DataTemplate>
```

```
</Window.Resources>
```

Now you can use `myTaskTemplate` as a resource, as in the following example:

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}"
         ItemTemplate="{StaticResource myTaskTemplate}"/>
```

Because `myTaskTemplate` is a resource, you can now use it on other controls that have a property that takes a

DataTemplate type. As shown above, for ItemsControl objects, such as the ListBox, it is the ItemTemplate property. For ContentControl objects, it is the ContentTemplate property.

The DataType Property

The DataTemplate class has a DataType property that is very similar to the TargetType property of the Style class. Therefore, instead of specifying an `x:Key` for the DataTemplate in the above example, you can do the following:

```
<DataTemplate DataType="{x:Type local:Task}">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}"/>
    <TextBlock Text="{Binding Path=Priority}"/>
  </StackPanel>
</DataTemplate>
```

This DataTemplate gets applied automatically to all `Task` objects. Note that in this case the `x:Key` is set implicitly. Therefore, if you assign this DataTemplate an `x:Key` value, you are overriding the implicit `x:Key` and the DataTemplate would not be applied automatically.

If you are binding a ContentControl to a collection of `Task` objects, the ContentControl does not use the above DataTemplate automatically. This is because the binding on a ContentControl needs more information to distinguish whether you want to bind to an entire collection or the individual objects. If your ContentControl is tracking the selection of an ItemsControl type, you can set the Path property of the ContentControl binding to "`/`" to indicate that you are interested in the current item. For an example, see Bind to a Collection and Display Information Based on Selection. Otherwise, you need to specify the DataTemplate explicitly by setting the ContentTemplate property.

The DataType property is particularly useful when you have a CompositeCollection of different types of data objects. For an example, see Implement a CompositeCollection.

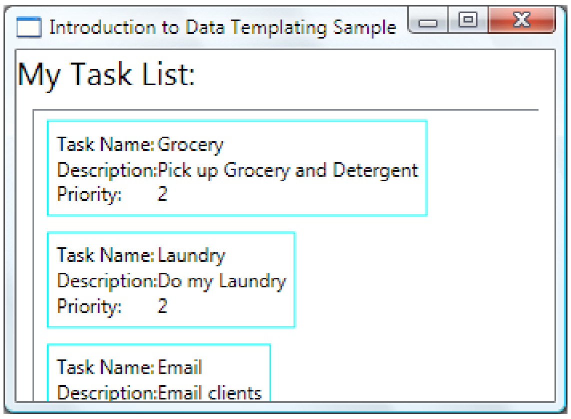
Adding More to the DataTemplate

Currently the data appears with the necessary information, but there's definitely room for improvement. Let's improve on the presentation by adding a Border, a Grid, and some TextBlock elements that describe the data that is being displayed.

```
<DataTemplate x:Key="myTaskTemplate">
  <Border Name="border" BorderBrush="Aqua" BorderThickness="1"
          Padding="5" Margin="5">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Text="Task Name:"/>
      <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=TaskName}" />
      <TextBlock Grid.Row="1" Grid.Column="0" Text="Description:"/>
      <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Description}"/>
      <TextBlock Grid.Row="2" Grid.Column="0" Text="Priority:"/>
      <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=Priority}"/>
    </Grid>
  </Border>
```

```
</DataTemplate>
```

The following screenshot shows the ListBox with this modified DataTemplate:


Introduction to Data Templating Sample

My Task List:

Task Name: Grocery
Description:Pick up Grocery and Detergent
Priority: 2

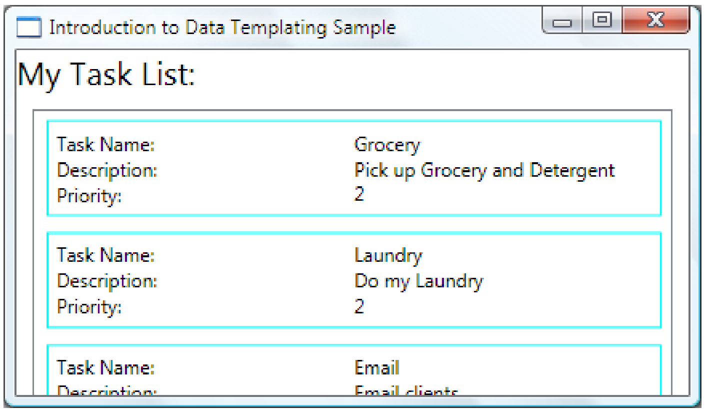
Task Name: Laundry
Description:Do my Laundry
Priority: 2

Task Name: Email
Description:Email clients

We can set HorizontalContentAlignment to Stretch on the ListBox to make sure the width of the items takes up the entire space:

```
<ListBox Width="400" Margin="10"
     ItemsSource="{Binding Source={StaticResource myTodoList}}"
     ItemTemplate="{StaticResource myTaskTemplate}"
     HorizontalContentAlignment="Stretch"/>
```

With the HorizontalContentAlignment property set to Stretch, the ListBox now looks like this:


Introduction to Data Templating Sample

My Task List:

Task Name: Grocery
Description: Pick up Grocery and Detergent
Priority: 2

Task Name: Laundry
Description: Do my Laundry
Priority: 2

Task Name: Email
Description: Email clients

Use DataTriggers to Apply Property Values

The current presentation does not tell us whether a `Task` is a home task or an office task. Remember that the `Task` object has a `TaskType` property of type `TaskType`, which is an enumeration with values `Home` and `Work`.

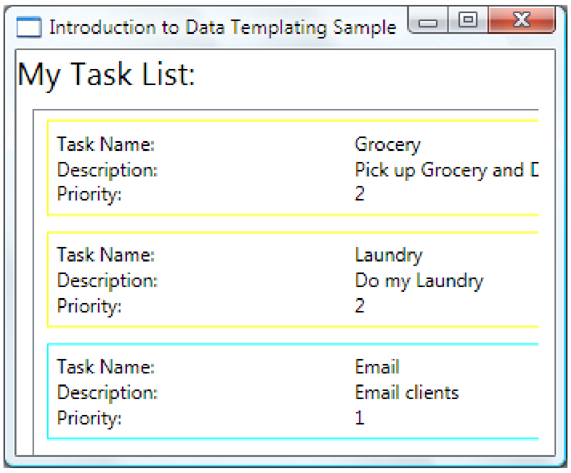
In the following example, the DataTrigger sets the BorderBrush of the element named `border` to `Yellow` if the `TaskType` property is `TaskType.Home`.

```
<DataTemplate x:Key="myTaskTemplate">
```

```
<DataTemplate.Triggers>
  <DataTrigger Binding="{Binding Path=TaskType}">
    <DataTrigger.Value>
      <local:TaskType>Home</local:TaskType>
    </DataTrigger.Value>
    <Setter TargetName="border" Property="BorderBrush" Value="Yellow"/>
  </DataTrigger>
</DataTemplate.Triggers>
```

```
</DataTemplate>
```

Our application now looks like the following. Home tasks appear with a yellow border and office tasks appear with an aqua border:



My Task List:

Task Name:	Grocery
Description:	Pick up Grocery and D
Priority:	2
Task Name:	Laundry
Description:	Do my Laundry
Priority:	2
Task Name:	Email
Description:	Email clients
Priority:	1

In this example the DataTrigger uses a Setter to set a property value. The trigger classes also have the EnterActions and ExitActions properties that allow you to start a set of actions such as animations. In addition, there is also a MultiDataTrigger class that allows you to apply changes based on multiple data-bound property values.

An alternative way to achieve the same effect is to bind the BorderBrush property to the `TaskType` property and use a value converter to return the color based on the `TaskType` value. Creating the above effect using a converter is slightly more efficient in terms of performance. Additionally, creating your own converter gives you more flexibility because you are supplying your own logic. Ultimately, which technique you choose depends on your scenario and your preference. For information about how to write a converter, see IValueConverter.

What Belongs in a DataTemplate?

In the previous example, we placed the trigger within the DataTemplate using the DataTemplate.Triggers property. The Setter of the trigger sets the value of a property of an element (the Border element) that is within the DataTemplate. However, if the properties that your `Setters` are concerned with are not properties of elements that are within the current DataTemplate, it may be more suitable to set the properties using a Style that is for the ListBoxItem class (if the control you are binding is a ListBox). For example, if you want your Trigger to animate the Opacity value of the item when a mouse points to an item, you define triggers within a ListBoxItem style. For an example, see the Introduction to Styling and Templating Sample.

In general, keep in mind that the DataTemplate is being applied to each of the generated ListBoxItem (for more information about how and where it is actually applied, see the ItemTemplate page.). Your DataTemplate is concerned with only the presentation and appearance of the data objects. In most cases, all other aspects of presentation, such as what an item looks like when it is selected or how the ListBox lays out the items, do not belong in the definition of a DataTemplate. For an example, see the Styling and Templating an ItemsControl section.

Choosing a DataTemplate Based on Properties of the Data Object

In The DataType Property section, we discussed that you can define different data templates for different data objects. That is especially useful when you have a CompositeCollection of different types or collections with items of different types. In the Use DataTriggers to Apply Property Values section, we have shown that if you have a collection of the same type of data objects you can create a DataTemplate and then use triggers to apply changes based on the property values of each data object. However, triggers allow you to apply property values or start animations but they don't give you the flexibility to reconstruct the structure of your data objects. Some scenarios may require you to create a different DataTemplate for data objects that are of the same type but have different properties.

For example, when a `Task` object has a `Priority` value of `1`, you may want to give it a completely different look to serve as an alert for yourself. In that case, you create a DataTemplate for the display of the high-priority `Task` objects. Let's add the following DataTemplate to the resources section:

```
<DataTemplate x:Key="importantTaskTemplate">
  <DataTemplate.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="20"/>
    </Style>
  </DataTemplate.Resources>
  <Border Name="border" BorderBrush="Red" BorderThickness="1"
          Padding="5" Margin="5">
    <DockPanel HorizontalAlignment="Center">
      <TextBlock Text="{Binding Path=Description}" />
      <TextBlock>!</TextBlock>
    </DockPanel>
  </Border>
</DataTemplate>
```

Notice this example uses the DataTemplate.Resources property. Resources defined in that section are shared by the elements within the DataTemplate.

To supply logic to choose which DataTemplate to use based on the `Priority` value of the data object, create a subclass of DataTemplateSelector and override the SelectTemplate method. In the following example, the SelectTemplate method provides logic to return the appropriate template based on the value of the `Priority` property. The template to return is found in the resources of the enveloping Window element.

```
using System.Windows;
using System.Windows.Controls;

namespace SDKSample
{
    public class TaskListDataTemplateSelector : DataTemplateSelector
    {
        public override DataTemplate
            SelectTemplate(object item, DependencyObject container)
        {
            FrameworkElement element = container as FrameworkElement;

            if (element != null && item != null && item is Task)
            {
                Task taskitem = item as Task;

                if (taskitem.Priority == 1)
                    return
                        element.FindResource("importantTaskTemplate") as DataTemplate;
                else
                    return
                        element.FindResource("myTaskTemplate") as DataTemplate;
            }

            return null;
        }
    }
}
```

```

Namespace SDKSample
    Public Class TaskListDataTemplateSelector
        Inherits DataTemplateSelector
        Public Overrides Function SelectTemplate(ByVal item As Object, ByVal container As DependencyObject) As
DataTemplate

            Dim element As FrameworkElement
            element = TryCast(container, FrameworkElement)

            If element IsNot Nothing AndAlso item IsNot Nothing AndAlso TypeOf item Is Task Then

                Dim taskitem As Task = TryCast(item, Task)

                If taskitem.Priority = 1 Then
                    Return TryCast(element.FindResource("importantTaskTemplate"), DataTemplate)
                Else
                    Return TryCast(element.FindResource("myTaskTemplate"), DataTemplate)
                End If
            End If

            Return Nothing
        End Function
    End Class
End Namespace
```

We can then declare the `TaskListDataTemplateSelector` as a resource:

```
<Window.Resources>
```

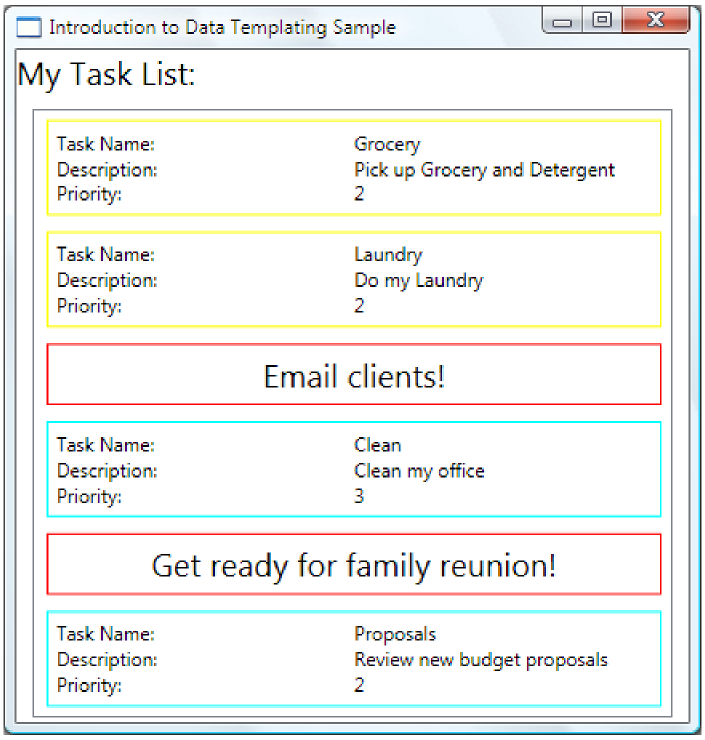
```
<local:TaskListDataTemplateSelector x:Key="myDataTemplateSelector"/>
```

```
</Window.Resources>
```

To use the template selector resource, assign it to the ItemTemplateSelector property of the ListBox. The ListBox calls the SelectTemplate method of the `TaskListDataTemplateSelector` for each of the items in the underlying collection. The call passes the data object as the item parameter. The DataTemplate that is returned by the method is then applied to that data object.

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}"
         ItemTemplateSelector="{StaticResource myDataTemplateSelector}"
         HorizontalContentAlignment="Stretch"/>
```

With the template selector in place, the ListBox now appears as follows:



My Task List:

Task Name:	Grocery
Description:	Pick up Grocery and Detergent
Priority:	2
Task Name:	Laundry
Description:	Do my Laundry
Priority:	2

Email clients!

Task Name:	Clean
Description:	Clean my office
Priority:	3

Get ready for family reunion!

Task Name:	Proposals
Description:	Review new budget proposals
Priority:	2

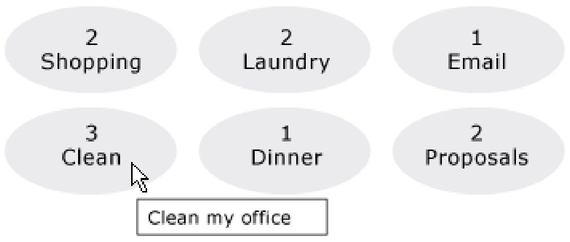
This concludes our discussion of this example. For the complete sample, see Introduction to Data Templating Sample.

Styling and Templating an ItemsControl

Even though the ItemsControl is not the only control type that you can use a DataTemplate with, it is a very common scenario to bind an ItemsControl to a collection. In the What Belongs in a DataTemplate section we discussed that the definition of your DataTemplate should only be concerned with the presentation of data. In order to know when it is not suitable to use a DataTemplate it is important to understand the different style and template properties provided by the ItemsControl. The following example is designed to illustrate the function of each of these properties. The ItemsControl in this example is bound to the same `Tasks` collection as in the previous example. For demonstration purposes, the styles and templates in this example are all declared inline.

```xml
<ItemsControl Margin="10"
              ItemsSource="{Binding Source={StaticResource myTodoList}}">
  <!--The ItemsControl has no default visual appearance.
      Use the Template property to specify a ControlTemplate to define
      the appearance of an ItemsControl. The ItemsPresenter uses the specified
      ItemsPanelTemplate (see below) to layout the items. If an
      ItemsPanelTemplate is not specified, the default is used. (For ItemsControl,
      the default is an ItemsPanelTemplate that specifies a StackPanel.-->
  <ItemsControl.Template>
    <ControlTemplate TargetType="ItemsControl">
      <Border BorderBrush="Aqua" BorderThickness="1" CornerRadius="15">
        <ItemsPresenter/>
      </Border>
    </ControlTemplate>
  </ItemsControl.Template>
  <!--Use the ItemsPanel property to specify an ItemsPanelTemplate
      that defines the panel that is used to hold the generated items.
      In other words, use this property if you want to affect
      how the items are laid out.-->
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <WrapPanel />
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
  <!--Use the ItemTemplate to set a DataTemplate to define
      the visualization of the data objects. This DataTemplate
      specifies that each data object appears with the Proriity
      and TaskName on top of a silver ellipse.-->
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <DataTemplate.Resources>
        <Style TargetType="TextBlock">
          <Setter Property="FontSize" Value="18"/>
          <Setter Property="HorizontalAlignment" Value="Center"/>
        </Style>
      </DataTemplate.Resources>
      <Grid>
        <Ellipse Fill="Silver"/>
        <StackPanel>
          <TextBlock Margin="3,3,3,0"
                     Text="{Binding Path=Priority}"/>
          <TextBlock Margin="3,0,3,7"
                     Text="{Binding Path=TaskName}"/>
        </StackPanel>
      </Grid>
    </DataTemplate>
  </ItemsControl.ItemTemplate>
  <!--Use the ItemContainerStyle property to specify the appearance
      of the element that contains the data. This ItemContainerStyle
      gives each item container a margin and a width. There is also
      a trigger that sets a tooltip that shows the description of
      the data object when the mouse hovers over the item container.-->
  <ItemsControl.ItemContainerStyle>
    <Style>
      <Setter Property="Control.Width" Value="100"/>
      <Setter Property="Control.Margin" Value="5"/>
      <Style.Triggers>
        <Trigger Property="Control.IsMouseOver" Value="True">
          <Setter Property="Control.ToolTip"
                  Value="{Binding RelativeSource={x:Static RelativeSource.Self},
                          Path=Content.Description}"/>
        </Trigger>
      </Style.Triggers>
    </Style>
  </ItemsControl.ItemContainerStyle>
</ItemsControl>
```

The following is a screenshot of the example when it is rendered:



Note that instead of using the ItemTemplate, you can use the ItemTemplateSelector. Refer to the previous section for an example. Similarly, instead of using the ItemContainerStyle, you have the option to use the ItemContainerStyleSelector.

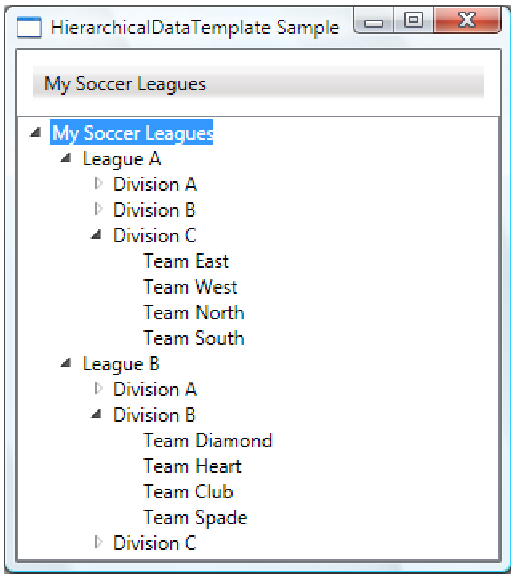
Two other style-related properties of the ItemsControl that are not shown here are GroupStyle and GroupStyleSelector.

Support for Hierarchical Data

So far we have only looked at how to bind to and display a single collection. Sometimes you have a collection that contains other collections. The HierarchicalDataTemplate class is designed to be used with HeaderedItemsControl types to display such data. In the following example, `ListLeagueList` is a list of `League` objects. Each `League` object has a `Name` and a collection of `Division` objects. Each `Division` has a `Name` and a collection of `Team` objects, and each `Team` object has a `Name`.

```
<Window x:Class="SDKSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="HierarchicalDataTemplate Sample"
  xmlns:src="clr-namespace:SDKSample">
  <DockPanel>
    <DockPanel.Resources>
      <src:ListLeagueList x:Key="MyList"/>

      <HierarchicalDataTemplate DataType    = "{x:Type src:League}"
                                ItemsSource = "{Binding Path=Divisions}">
        <TextBlock Text="{Binding Path=Name}"/>
      </HierarchicalDataTemplate>

      <HierarchicalDataTemplate DataType    = "{x:Type src:Division}"
                                ItemsSource = "{Binding Path=Teams}">
        <TextBlock Text="{Binding Path=Name}"/>
      </HierarchicalDataTemplate>

      <DataTemplate DataType="{x:Type src:Team}">
        <TextBlock Text="{Binding Path=Name}"/>
      </DataTemplate>
    </DockPanel.Resources>

    <Menu Name="menu1" DockPanel.Dock="Top" Margin="10,10,10,10">
        <MenuItem Header="My Soccer Leagues"
                  ItemsSource="{Binding Source={StaticResource MyList}}" />
    </Menu>

    <TreeView>
      <TreeViewItem ItemsSource="{Binding Source={StaticResource MyList}}" Header="My Soccer Leagues" />
    </TreeView>

  </DockPanel>
</Window>
```

The example shows that with the use of HierarchicalDataTemplate, you can easily display list data that contains other lists. The following is a screenshot of the example.



My Soccer Leagues

- My Soccer Leagues
  - League A
    - Division A
    - Division B
    - Division C
      - Team East
      - Team West
      - Team North
      - Team South
  - League B
    - Division A
    - Division B
      - Team Diamond
      - Team Heart
      - Team Club
      - Team Spade
    - Division C

See also

- Data Binding
- Find DataTemplate-Generated Elements
- Styling and Templating
- Data Binding Overview
- GridView Column Header Styles and Templates Overview

Binding Declarations Overview

1/23/2019 • 6 minutes to read • Edit Online

This topic discusses the different ways you can declare a binding.

Prerequisites

Before reading this topic, it is important that you are familiar with the concept and usage of markup extensions. For more information about markup extensions, see Markup Extensions and WPF XAML.

This topic does not cover data binding concepts. For a discussion of data binding concepts, see Data Binding Overview.

Declaring a Binding in XAML

This section discusses how to declare a binding in XAML.

Markup Extension Usage

Binding is a markup extension. When you use the binding extension to declare a binding, the declaration consists of a series of clauses following the `Binding` keyword and separated by commas (,). The clauses in the binding declaration can be in any order and there are many possible combinations. The clauses are *Name*=*Value* pairs where *Name* is the name of the Binding property and *Value* is the value you are setting for the property.

When creating binding declaration strings in markup, they must be attached to the specific dependency property of a target object. The following example shows how to bind the TextBox.Text property using the binding extension, specifying the Source and Path properties.

```
<TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}"/>
```

You can specify most of the properties of the Binding class this way. For more information about the binding extension as well as for a list of Binding properties that cannot be set using the binding extension, see the Binding Markup Extension overview.

Object Element Syntax

Object element syntax is an alternative to creating the binding declaration. In most cases, there is no particular advantage to using either the markup extension or the object element syntax. However, in cases which the markup extension does not support your scenario, such as when your property value is of a non-string type for which no type conversion exists, you need to use the object element syntax.

The following is an example of both the object element syntax and the markup extension usage:

```
<TextBlock Name="myconvertedtext"
  Foreground="{Binding Path=TheDate,
                       Converter={StaticResource MyConverterReference}}">
  <TextBlock.Text>
    <Binding Path="TheDate"
             Converter="{StaticResource MyConverterReference}"/>
  </TextBlock.Text>
</TextBlock>
```

The example binds the Foreground property by declaring a binding using the extension syntax. The binding declaration for the Text property uses the object element syntax.

For more information about the different terms, see XAML Syntax In Detail.

MultiBinding and PriorityBinding

MultiBinding and PriorityBinding do not support the XAML extension syntax. Therefore, you must use the object element syntax if you are declaring a MultiBinding or a PriorityBinding in XAML.

Creating a Binding in Code

Another way to specify a binding is to set properties directly on a Binding object in code. The following example shows how to create a Binding object and specify the properties in code. In this example, `TheConverter` is an object that implements the IValueConverter interface.

```
private void OnPageLoaded(object sender, EventArgs e)
{
    // Make a new source, to grab a new timestamp
    MyData myChangedData = new MyData();

    // Create a new binding
    // TheDate is a property of type DateTime on MyData class
    Binding myNewBindDef = new Binding("TheDate");

    myNewBindDef.Mode = BindingMode.OneWay;
    myNewBindDef.Source = myChangedData;
    myNewBindDef.Converter = TheConverter;
    myNewBindDef.ConverterCulture = new CultureInfo("en-US");

      // myDatetext is a TextBlock object that is the binding target object
    BindingOperations.SetBinding(myDateText, TextBlock.TextProperty, myNewBindDef);
    BindingOperations.SetBinding(myDateText, TextBlock.ForegroundProperty, myNewBindDef);

    lbChooseCulture.SelectedIndex = 0;
}
```

```
 Private Sub OnPageLoaded(ByVal sender As Object, ByVal e As EventArgs)
     ' Make a new source, to grab a new timestamp
     Dim myChangedData As New MyData()

     ' Create a new binding
 ' TheDate is a property of type DateTime on MyData class
     Dim myNewBindDef As New Binding("TheDate")

     myNewBindDef.Mode = BindingMode.OneWay
     myNewBindDef.Source = myChangedData
     myNewBindDef.Converter = TheConverter
     myNewBindDef.ConverterCulture = New CultureInfo("en-US")

' myDatetext is a TextBlock object that is the binding target object
     BindingOperations.SetBinding(myDateText, TextBlock.TextProperty, myNewBindDef)
     BindingOperations.SetBinding(myDateText, TextBlock.ForegroundProperty, myNewBindDef)

     lbChooseCulture.SelectedIndex = 0
 End Sub
```

If the object you are binding is a FrameworkElement or a FrameworkContentElement you can call the `SetBinding` method on your object directly instead of using BindingOperations.SetBinding. For an example, see Create a Binding in Code.

Binding Path Syntax

Use the Path property to specify the source value you want to bind to:

- In the simplest case, the Path property value is the name of the property of the source object to use for the binding, such as `Path=PropertyName`.
- Subproperties of a property can be specified by a similar syntax as in C#. For instance, the clause `Path=ShoppingCart.Order` sets the binding to the subproperty `Order` of the object or property `ShoppingCart`.
- To bind to an attached property, place parentheses around the attached property. For example, to bind to the attached property DockPanel.Dock, the syntax is `Path=(DockPanel.Dock)`.
- Indexers of a property can be specified within square brackets following the property name where the indexer is applied. For instance, the clause `Path=ShoppingCart[0]` sets the binding to the index that corresponds to how your property's internal indexing handles the literal string "0". Nested indexers are also supported.
- Indexers and subproperties can be mixed in a `Path` clause; for example, `Path=ShoppingCart.ShippingInfo[MailingAddress,Street].`
- Inside indexers you can have multiple indexer parameters separated by commas (,). The type of each parameter can be specified with parentheses. For example, you can have `Path="[(sys:Int32)42,(sys:Int32)24]"`, where `sys` is mapped to the `System` namespace.
- When the source is a collection view, the current item can be specified with a slash (/). For example, the clause `Path=/` sets the binding to the current item in the view. When the source is a collection, this syntax specifies the current item of the default collection view.
- Property names and slashes can be combined to traverse properties that are collections. For example, `Path=/Offices/ManagerName` specifies the current item of the source collection, which contains an `Offices` property that is also a collection. Its current item is an object that contains a `ManagerName` property.
- Optionally, a period (.) path can be used to bind to the current source. For example, `Text="{Binding}"` is equivalent to `Text="{Binding Path=.}"`.

Escaping Mechanism

- Inside indexers ([ ]), the caret character (^) escapes the next character.
- If you set Path in XAML, you also need to escape (using XML entities) certain characters that are special to the XML language definition:
  - Use `&` to escape the character "&".
  - Use `>` to escape the end tag ">".
- Additionally, if you describe the entire binding in an attribute using the markup extension syntax, you need to escape (using backslash \) characters that are special to the WPF markup extension parser:
  - Backslash (\) is the escape character itself.
  - The equal sign (=) separates property name from property value.
  - Comma (,) separates properties.
  - The right curly brace (}) is the end of a markup extension.

Default Behaviors

The default behavior is as follows if not specified in the declaration.

- A default converter is created that tries to do a type conversion between the binding source value and the binding target value. If a conversion cannot be made, the default converter returns `null`.
- If you do not set ConverterCulture, the binding engine uses the `Language` property of the binding target object. In XAML, this defaults to "en-US" or inherits the value from the root element (or any element) of the page, if one has been explicitly set.
- As long as the binding already has a data context (for instance, the inherited data context coming from a parent element), and whatever item or collection being returned by that context is appropriate for binding without requiring further path modification, a binding declaration can have no clauses at all: `{Binding}` This is often the way a binding is specified for data styling, where the binding acts upon a collection. For more information, see the "Entire Objects Used as a Binding Source" section in the Binding Sources Overview.
- The default Mode varies between one-way and two-way depending on the dependency property that is being bound. You can always declare the binding mode explicitly to ensure that your binding has the desired behavior. In general, user-editable control properties, such as TextBox.Text and RangeBase.Value, default to two-way bindings, whereas most other properties default to one-way bindings.
- The default UpdateSourceTrigger value varies between PropertyChanged and LostFocus depending on the bound dependency property as well. The default value for most dependency properties is PropertyChanged, while the TextBox.Text property has a default value of LostFocus.

See also

- Data Binding Overview
- How-to Topics
- Data Binding
- PropertyPath XAML Syntax

Data Binding How-to Topics

5/4/2018 • 2 minutes to read • Edit Online

The topics in this section describe how to use data binding to bind elements to data from a variety of data sources in the form of common language runtime (CLR) objects and XML.

In This Section

Create a Simple Binding
Specify the Binding Source
Make Data Available for Binding in XAML
Control When the TextBox Text Updates the Source
Specify the Direction of the Binding
Bind to a Collection and Display Information Based on Selection
Bind to an Enumeration
Bind the Properties of Two Controls
Implement Binding Validation
Implement Validation Logic on Custom Objects
Get the Binding Object from a Bound Target Property
Implement a CompositeCollection
Convert Bound Data
Create a Binding in Code
Get the Default View of a Data Collection
Navigate Through the Objects in a Data CollectionView
Filter Data in a View
Sort Data in a View
Sort and Group Data Using a View in XAML
Use the Master-Detail Pattern with Hierarchical Data
Use the Master-Detail Pattern with Hierarchical XML Data
Produce a Value Based on a List of Bound Items
Implement Property Change Notification
Create and Bind to an ObservableCollection
Implement PriorityBinding
Bind to XML Data Using an XMLDataProvider and XPath Queries
Bind to XDocument, XElement, or LINQ for XML Query Results
Use XML Namespaces in Data Binding
Bind to an ADO.NET Data Source
Bind to a Method
Set Up Notification of Binding Updates
Clear Bindings
Find DataTemplate-Generated Elements
Bind to a Web Service
Bind to the Results of a LINQ Query

Reference

System.Windows.Data

Binding

DataTemplate

DataTemplateSelector

Related Sections

Data Binding

Data Binding

How to: Create a Simple Binding

1/23/2019 • 2 minutes to read • Edit Online

This example shows you how to create a simple Binding.

Example

In this example, you have a `Person` object with a string property named `PersonName`. The `Person` object is defined in the namespace called `SDKSample`.

The highlighted line that contains the `<src>` element in the following example instantiates the `Person` object with a `PersonName` property value of `Joe`. This is done in the `Resources` section and assigned an `x:Key`.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  SizeToContent="WidthAndHeight"
  Title="Simple Data Binding Sample">

  <Window.Resources>
    <src:Person x:Key="myDataSource" PersonName="Joe"/>
    <Style TargetType="{x:Type Label}">
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="Padding" Value="3"/>
    </Style>
  </Window.Resources>
  <Border Margin="5" BorderBrush="Aqua" BorderThickness="1" Padding="8" CornerRadius="3">
    <DockPanel Width="200" Height="100" Margin="35">
      <Label>Enter a Name:</Label>
      <TextBox>
        <TextBox.Text>
          <Binding Source="{StaticResource myDataSource}" Path="PersonName"
                   UpdateSourceTrigger="PropertyChanged"/>
        </TextBox.Text>
      </TextBox>

      <Label>The name you entered:</Label>
      <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}"/>
    </DockPanel>
  </Border>
</Window>
```

The highlighted line that contains the `<TextBlock>` element then binds the TextBlock control to the `PersonName` property. As a result, the TextBlock appears with the value "Joe".

See also

- Data Binding Overview
- How-to Topics

How to: Specify the Binding Source

1/23/2019 • 4 minutes to read • Edit Online

In data binding, the binding source object refers to the object you obtain your data from. This topic describes the different ways of specifying the binding source.

Example

If you are binding several properties to a common source, you want to use the `DataContext` property, which provides a convenient way to establish a scope within which all data-bound properties inherit a common source.

In the following example, the data context is established on the root element of the application. This allows all child elements to inherit that data context. Data for the binding comes from a custom data class, `NetIncome`, referenced directly through a mapping and given the resource key of `incomeDataSource`.

```
<Grid
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.DirectionalBinding"
  xmlns:c="clr-namespace:SDKSample"
  Name="Page1"
>
  <Grid.Resources>
    <c:NetIncome x:Key="incomeDataSource"/>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="Padding" Value="8"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
      <Setter Property="Margin" Value="0,6,0,0"/>
    </Style>
  </Grid.Resources>
  <Grid.DataContext>
    <Binding Source="{StaticResource incomeDataSource}"/>
  </Grid.DataContext>
```

```
</Grid>
```

The following example shows the definition of the `NetIncome` class.

```
public class NetIncome : INotifyPropertyChanged
{
    private int totalIncome = 5000;
    private int rent = 2000;
    private int food = 0;
    private int misc = 0;
    private int savings = 0;
    public NetIncome()
    {
        savings = totalIncome - (rent+food+misc);
    }

    public int TotalIncome
    {
        get
        {
            return totalIncome;
        }
        set
        {
            if( TotalIncome != value)
            {
                totalIncome = value;
                OnPropertyChanged("TotalIncome");
            }
        }
    }
    public int Rent
    {
        get
        {
            return rent;
        }
        set
        {
            if( Rent != value)
            {
                rent = value;
                OnPropertyChanged("Rent");
                UpdateSavings();
            }
        }
    }
    public int Food
    {
        get
        {
            return food;
        }
        set
        {
            if( Food != value)
            {
                food = value;
                OnPropertyChanged("Food");
                UpdateSavings();
            }
        }
    }
    public int Misc
    {
        get
        {
            return misc;
        }
        set
        {
            if( Misc != value)
            {
                misc = value;
                OnPropertyChanged("Misc");
                UpdateSavings();
            }
        }
    }
    public int Savings
    {
        get
        {
            return savings;
        }
        set
        {
            if( Savings != value)
            {
                savings = value;
                OnPropertyChanged("Savings");
                UpdateSavings();
            }
        }
    }

    private void UpdateSavings()
    {
        Savings = TotalIncome - (Rent+Misc+Food);
        if(Savings < 0)
        {}
        else if(Savings >= 0)
        {}
    }
    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(String info)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler !=null)
        {
            handler(this, new PropertyChangedEventArgs(info));
        }
    }
}
```

```
Public Class NetIncome
    Implements INotifyPropertyChanged

    ' Events
    Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged

    ' Methods
    Public Sub New()
        Me._totalIncome = 5000
        Me._rent = 2000
        Me._food = 0
        Me._misc = 0
        Me._savings = 0
        Me._savings = (Me.TotalIncome - ((Me.Rent + Me.Food) + Me.Misc))
    End Sub

    Private Sub OnPropertyChanged(ByVal info As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(info))
    End Sub

    Private Sub UpdateSavings()
        Me.Savings = (Me.TotalIncome - ((Me.Rent + Me.Misc) + Me.Food))
        If ((Me.Savings >= 0) AndAlso (Me.Savings >= 0)) Then
        End If
    End Sub


    ' Properties
    Public Property Food As Integer
        Get
            Return Me._food
        End Get
        Set(ByVal value As Integer)
            If (Me.Food <> value) Then
                Me._food = value
                Me.OnPropertyChanged("Food")
                Me.UpdateSavings()
            End If
        End Set
    End Property
    Public Property Misc As Integer
        Get
            Return Me._misc
        End Get
        Set(ByVal value As Integer)
            If (Me.Misc <> value) Then
                Me._misc = value
                Me.OnPropertyChanged("Misc")
                Me.UpdateSavings()
            End If
        End Set
    End Property

    Public Property Rent As Integer
        Get
            Return Me._rent
        End Get
        Set(ByVal value As Integer)
            If (Me.Rent <> value) Then
                Me._rent = value
                Me.OnPropertyChanged("Rent")
                Me.UpdateSavings()
            End If
        End Set
    End Property

    Public Property Savings As Integer
        Get
            Return Me._savings
        End Get
        Set(ByVal value As Integer)
            If (Me.Savings <> value) Then
                Me._savings = value
                Me.OnPropertyChanged("Savings")
                Me.UpdateSavings()
            End If
        End Set
    End Property

    Public Property TotalIncome As Integer
        Get
            Return Me._totalIncome
        End Get
        Set(ByVal value As Integer)
            If (Me.TotalIncome <> value) Then
                Me._totalIncome = value
                Me.OnPropertyChanged("TotalIncome")
            End If
        End Set
    End Property


    ' Fields
    Private _food As Integer
    Private _misc As Integer
    Private _rent As Integer
    Private _savings As Integer
    Private _totalIncome As Integer
End Class
```

NOTE

The above example instantiates the object in markup and uses it as a resource. If you want to bind to an object that has already been instantiated in code, you need to set the `DataContext` property programmatically. For an example, see Make Data Available for Binding in XAML.

Alternatively, if you want to specify the source on your individual bindings explicitly, you have the following options. These take precedence over the inherited data context.

PROPERTY	DESCRIPTION
Source	You use this property to set the source to an instance of an object. If you do not need the functionality of establishing a scope in which several properties inherit the same data context, you can use the Source property instead of the `DataContext` property. For more information, see Source.
RelativeSource	This is useful when you want to specify the source relative to where your binding target is. Some common scenarios where you may use this property is when you want to bind one property of your element to another property of the same element or if you are defining a binding in a style or a template. For more information, see RelativeSource.
ElementName	You specify a string that represents the element you want to bind to. This is useful when you want to bind to the property of another element on your application. For example, if you want to use a Slider to control the height of another control in your application, or if you want to bind the Content of your control to the SelectedValue property of your ListBox control. For more information, see ElementName.

See also

- FrameworkElement.DataContext
- FrameworkContentElement.DataContext
- Property Value Inheritance
- Data Binding Overview
- Binding Declarations Overview
- How-to Topics

How to: Make Data Available for Binding in XAML

1/23/2019 • 3 minutes to read • Edit Online

This topic discusses various ways you can make data available for binding in Extensible Application Markup Language (XAML), depending on the needs of your application.

Example

If you have a common language runtime (CLR) object you would like to bind to from XAML, one way you can make the object available for binding is to define it as a resource and give it an `x:Key`. In the following example, you have a `Person` object with a string property named `PersonName`. The `Person` object (in the line shown highlighted that contains the `<src>` element) is defined in the namespace called `SDKSample`.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  SizeToContent="WidthAndHeight"
  Title="Simple Data Binding Sample">

  <Window.Resources>
    <src:Person x:Key="myDataSource" PersonName="Joe"/>
    <Style TargetType="{x:Type Label}">
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="Padding" Value="3"/>
    </Style>
  </Window.Resources>
  <Border Margin="5" BorderBrush="Aqua" BorderThickness="1" Padding="8" CornerRadius="3">
    <DockPanel Width="200" Height="100" Margin="35">
      <Label>Enter a Name:</Label>
      <TextBox>
        <TextBox.Text>
          <Binding Source="{StaticResource myDataSource}" Path="PersonName"
                   UpdateSourceTrigger="PropertyChanged"/>
        </TextBox.Text>
      </TextBox>

      <Label>The name you entered:</Label>
      <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}"/>
    </DockPanel>
  </Border>
</Window>
```

You can then bind the TextBlock control to the object in XAML, as the highlighted line that contains the `<TextBlock>` element shows.

Alternatively, you can use the ObjectDataProvider class, as in the following example:

```xml
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  xmlns:system="clr-namespace:System;assembly=mscorlib"
  SizeToContent="WidthAndHeight"
  Title="Simple Data Binding Sample">

  <Window.Resources>
    <ObjectDataProvider x:Key="myDataSource" ObjectType="{x:Type src:Person}">
      <ObjectDataProvider.ConstructorParameters>
        <system:String>Joe</system:String>
      </ObjectDataProvider.ConstructorParameters>
    </ObjectDataProvider>
    <Style TargetType="{x:Type Label}">
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
  </Window.Resources>

  <Border Margin="25" BorderBrush="Aqua" BorderThickness="3" Padding="8">
    <DockPanel Width="200" Height="100">
      <Label>Enter a Name:</Label>
      <TextBox>
        <TextBox.Text>
          <Binding Source="{StaticResource myDataSource}" Path="Name"
                   UpdateSourceTrigger="PropertyChanged"/>
        </TextBox.Text>
      </TextBox>

      <Label>The name you entered:</Label>
      <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=Name}"/>
    </DockPanel>
  </Border>
</Window>
```

You define the binding the same way, as the highlighted line that contains the `<TextBlock>` element shows.

In this particular example, the result is the same: you have a TextBlock with the text content `Joe`. However, the ObjectDataProvider class provides functionality such as the ability to bind to the result of a method. You can choose to use the ObjectDataProvider class if you need the functionality it provides.

However, if you are binding to an object that has already been created, you need to set the `DataContext` in code, as in the following example.

```
DataSet myDataSet;

private void OnInit(object sender, EventArgs e)
{
  string mdbFile = Path.Combine(AppDataPath, "BookData.mdb");
  string connString = string.Format(
      "Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}", mdbFile);
  OleDbConnection conn = new OleDbConnection(connString);
  OleDbDataAdapter adapter = new OleDbDataAdapter("SELECT * FROM BookTable;", conn);

  myDataSet = new DataSet();
  adapter.Fill(myDataSet, "BookTable");

  // myListBox is a ListBox control.
  // Set the DataContext of the ListBox to myDataSet
  myListBox.DataContext = myDataSet;
}
```

```
Private myDataSet As DataSet

Private Sub OnInit(ByVal sender As Object, ByVal e As EventArgs)
  Dim mdbFile As String = Path.Combine(AppDataPath, "BookData.mdb")
  Dim connString As String = String.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}", mdbFile)
  Dim conn As New OleDbConnection(connString)
  Dim adapter As New OleDbDataAdapter("SELECT * FROM BookTable;", conn)

  myDataSet = New DataSet()
  adapter.Fill(myDataSet, "BookTable")

  ' myListBox is a ListBox control.
  ' Set the DataContext of the ListBox to myDataSet
  myListBox.DataContext = myDataSet
End Sub
```

To access XML data for binding using the XmlDataProvider class, see Bind to XML Data Using an XMLDataProvider and XPath Queries. To access XML data for binding using the ObjectDataProvider class, see Bind to XDocument, XElement, or LINQ for XML Query Results.

For information about many ways you can specify the data you are binding to, see Specify the Binding Source. For information about what types of data you can bind to or how to implement your own common language runtime (CLR) objects for binding, see Binding Sources Overview.

See also

- Data Binding Overview
- How-to Topics

How to: Control When the TextBox Text Updates the Source

1/23/2019 • 2 minutes to read • Edit Online

This topic describes how to use the UpdateSourceTrigger property to control the timing of binding source updates. The topic uses the TextBox control as an example.

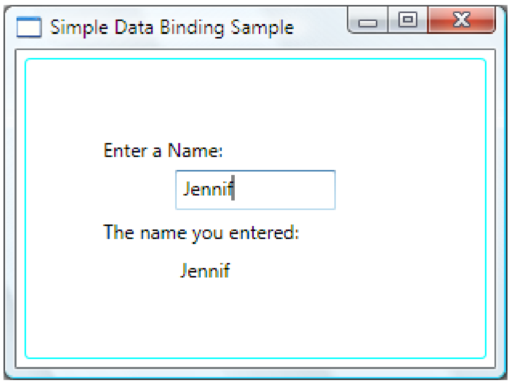
Example

The TextBox.Text property has a default UpdateSourceTrigger value of LostFocus. This means if an application has a TextBox with a data-bound TextBox.Text property, the text you type into the TextBox does not update the source until the TextBox loses focus (for instance, when you click away from the TextBox).

If you want the source to be updated as you type, set the UpdateSourceTrigger of the binding to PropertyChanged. In the following example, the highlighted lines of code show that the `Text` properties of both the TextBox and the TextBlock are bound to the same source property. The UpdateSourceTrigger property of the TextBox binding is set to PropertyChanged.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  xmlns:system="clr-namespace:System;assembly=mscorlib"
  SizeToContent="WidthAndHeight"
  Title="Simple Data Binding Sample">

  <Window.Resources>
    <ObjectDataProvider x:Key="myDataSource" ObjectType="{x:Type src:Person}">
      <ObjectDataProvider.ConstructorParameters>
        <system:String>Joe</system:String>
      </ObjectDataProvider.ConstructorParameters>
    </ObjectDataProvider>
    <Style TargetType="{x:Type Label}">
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
  </Window.Resources>

  <Border Margin="25" BorderBrush="Aqua" BorderThickness="3" Padding="8">
    <DockPanel Width="200" Height="100">
      <Label>Enter a Name:</Label>
      <TextBox>
        <TextBox.Text>
          <Binding Source="{StaticResource myDataSource}" Path="Name"
                   UpdateSourceTrigger="PropertyChanged"/>
        </TextBox.Text>
      </TextBox>

      <Label>The name you entered:</Label>
      <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=Name}"/>
    </DockPanel>
  </Border>
</Window>
```

As a result, the TextBlock shows the same text (because the source changes) as the user enters text into the TextBox, as illustrated by the following screenshot of the sample:



Enter a Name:

Jennif

The name you entered:

Jennif

If you have a dialog or a user-editable form and you want to defer source updates until the user is finished editing the fields and clicks "OK", you can set the UpdateSourceTrigger value of your bindings to Explicit, as in the following example:

```
<TextBox Name="itemNameTextBox"
         Text="{Binding Path=ItemName, UpdateSourceTrigger=Explicit}" />
```

When you set the UpdateSourceTrigger value to Explicit, the source value only changes when the application calls the UpdateSource method. The following example shows how to call UpdateSource for `itemNameTextBox`:

```
// itemNameTextBox is an instance of a TextBox
BindingExpression be = itemNameTextBox.GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();
```

```
Me.itemNameTextBox.GetBindingExpression(TextBox.TextProperty).UpdateSource()
Me.bidPriceTextBox.GetBindingExpression(TextBox.TextProperty).UpdateSource()
```

NOTE

You can use the same technique for properties of other controls, but keep in mind that most other properties have a default UpdateSourceTrigger value of PropertyChanged. For more information, see the UpdateSourceTrigger property page.

NOTE

The UpdateSourceTrigger property deals with source updates and therefore is only relevant for TwoWay or OneWayToSource bindings. For TwoWay and OneWayToSource bindings to work, the source object needs to provide property change notifications. You can refer to the samples cited in this topic for more information. In addition, you can look at Implement Property Change Notification.

See also

- How-to Topics

How to: Specify the Direction of the Binding

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to specify whether the binding updates only the binding target (target) property, the binding source (source) property, or both the target property and the source property.

Example

You use the Mode property to specify the direction of the binding. The following enumeration list shows the available options for binding updates:

- TwoWay updates the target property or the property whenever either the target property or the source property changes.
- OneWay updates the target property only when the source property changes.
- OneTime updates the target property only when the application starts or when the DataContext undergoes a change.
- OneWayToSource updates the source property when the target property changes.
- Default causes the default Mode value of target property to be used.

For more information, see the BindingMode enumeration.

The following example shows how to set the Mode property.

```
<TextBlock Name="IncomeText" Grid.Row="0" Grid.Column="1"
  Text="{Binding Path=TotalIncome, Mode=OneTime}"/>
```

To detect source changes (applicable to OneWay and TwoWay bindings), the source must implement a suitable property change notification mechanism such as INotifyPropertyChanged. See Implement Property Change Notification for an example of an INotifyPropertyChanged implementation.

For TwoWay or OneWayToSource bindings, you can control the timing of the source updates by setting the UpdateSourceTrigger property. See UpdateSourceTrigger for more information.

See also

- Binding
- Data Binding Overview
- How-to Topics

How to: Bind to a Collection and Display Information Based on Selection

1/23/2019 • 2 minutes to read • Edit Online

In a simple master-detail scenario, you have a data-bound ItemsControl such as a ListBox. Based on user selection, you display more information about the selected item. This example shows how to implement this scenario.

Example

In this example, `People` is an ObservableCollection<T> of `Person` classes. This `Person` class contains three properties: `FirstName`, `LastName`, and `HomeTown`, all of type `string`.

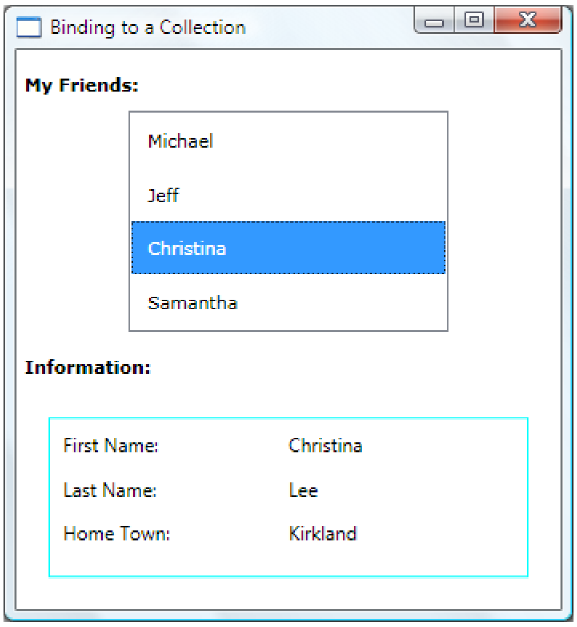
```
<Window x:Class="SDKSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:SDKSample"
  Title="Binding to a Collection"
  SizeToContent="WidthAndHeight">
  <Window.Resources>
    <local:People x:Key="MyFriends"/>
```

```
  </Window.Resources>

  <StackPanel>
    <TextBlock FontFamily="Verdana" FontSize="11"
               Margin="5,15,0,10" FontWeight="Bold">My Friends:</TextBlock>
    <ListBox Width="200" IsSynchronizedWithCurrentItem="True"
             ItemsSource="{Binding Source={StaticResource MyFriends}}"/>
    <TextBlock FontFamily="Verdana" FontSize="11"
               Margin="5,15,0,5" FontWeight="Bold">Information:</TextBlock>
    <ContentControl Content="{Binding Source={StaticResource MyFriends}}"
                    ContentTemplate="{StaticResource DetailTemplate}"/>
  </StackPanel>
</Window>
```

The ContentControl uses the following DataTemplate that defines how the information of a `Person` is presented:

```
<DataTemplate x:Key="DetailTemplate">
  <Border Width="300" Height="100" Margin="20"
          BorderBrush="Aqua" BorderThickness="1" Padding="8">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Text="First Name:"/>
      <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=FirstName}"/>
      <TextBlock Grid.Row="1" Grid.Column="0" Text="Last Name:"/>
      <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=LastName}"/>
      <TextBlock Grid.Row="2" Grid.Column="0" Text="Home Town:"/>
      <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=HomeTown}"/>
    </Grid>
  </Border>
</DataTemplate>
```

The following is a screenshot of what the example produces. The ContentControl shows the other properties of the person selected.



The two things to notice in this example are:

1. The ListBox and the ContentControl bind to the same source. The Path properties of both bindings are not specified because both controls are binding to the entire collection object.

2. You must set the IsSynchronizedWithCurrentItem property to `true` for this to work. Setting this property ensures that the selected item is always set as the CurrentItem. Alternatively, if the ListBox gets it data from a CollectionViewSource, it synchronizes selection and currency automatically.

Note that the `Person` class overrides the `ToString` method the following way. By default, the ListBox calls `ToString` and displays a string representation of each object in the bound collection. That is why each `Person` appears as a first name in the ListBox.

```
public override string ToString()
{
    return firstname.ToString();
}
```

```
Public Overrides Function ToString() As String
    Return Me._firstname.ToString
End Function
```

See also

- Use the Master-Detail Pattern with Hierarchical Data
- Use the Master-Detail Pattern with Hierarchical XML Data
- Data Binding Overview
- Data Templating Overview
- How-to Topics

How to: Bind to an Enumeration

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to bind to an enumeration by binding to the enumeration's GetValues method.

Example

In the following example, the ListBox displays the list of HorizontalAlignment enumeration values through data binding. The ListBox and the Button are bound such that you can change the HorizontalAlignment property value of the Button by selecting a value in the ListBox.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  SizeToContent="WidthAndHeight"
  Title="Show Enums in a ListBox using Binding">

  <Window.Resources>
    <ObjectDataProvider MethodName="GetValues"
                        ObjectType="{x:Type sys:Enum}"
                        x:Key="AlignmentValues">
      <ObjectDataProvider.MethodParameters>
        <x:Type TypeName="HorizontalAlignment" />
      </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
  </Window.Resources>

  <Border Margin="10" BorderBrush="Aqua"
          BorderThickness="3" Padding="8">
    <StackPanel Width="300">
      <TextBlock>Choose the HorizontalAlignment value of the Button:</TextBlock>
      <ListBox Name="myComboBox" SelectedIndex="0" Margin="8"
               ItemsSource="{Binding Source={StaticResource AlignmentValues}}"/>
      <Button Content="Click Me!"
              HorizontalAlignment="{Binding ElementName=myComboBox,
                                            Path=SelectedItem}"/>
    </StackPanel>
  </Border>
</Window>
```

See also

- Bind to a Method
- Data Binding Overview
- How-to Topics

How to: Bind the Properties of Two Controls

1/23/2019 • 2 minutes to read • Edit Online

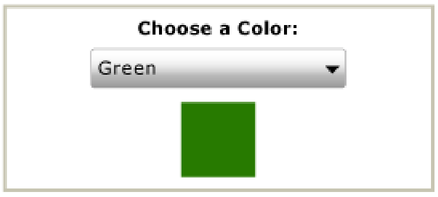
This example shows how to bind the property of one instantiated control to that of another using the ElementName property.

Example

The following example shows how to bind the Background property of a Canvas to the SelectedItem.Content property of a ComboBox:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="460" Height="200"
  Title="Binding the Properties of Two Controls">

  <Window.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="16"/>
      <Setter Property="FontWeight" Value="Bold"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="HorizontalAlignment" Value="Center"/>
    </Style>
    <Style TargetType="Canvas">
      <Setter Property="Height" Value="50"/>
      <Setter Property="Width" Value="50"/>
      <Setter Property="Margin" Value="8"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
    <Style TargetType="ComboBox">
      <Setter Property="Width" Value="150"/>
      <Setter Property="Margin" Value="8"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
    </Style>
  </Window.Resources>

  <Border Margin="10" BorderBrush="Silver" BorderThickness="3" Padding="8">
    <DockPanel>
      <TextBlock>Choose a Color:</TextBlock>
      <ComboBox Name="myComboBox" SelectedIndex="0">
        <ComboBoxItem>Green</ComboBoxItem>
        <ComboBoxItem>Blue</ComboBoxItem>
        <ComboBoxItem>Red</ComboBoxItem>
      </ComboBox>
      <Canvas>
        <Canvas.Background>
          <Binding ElementName="myComboBox" Path="SelectedItem.Content"/>
        </Canvas.Background>
      </Canvas>
    </DockPanel>
  </Border>
</Window>
```

When this example is rendered it looks like the following:



Note The binding target property (in this example, the Background property) must be a dependency property. For more information, see Data Binding Overview.

See also

- Specify the Binding Source
- How-to Topics

How to: Implement Binding Validation

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to use an ErrorTemplate and a style trigger to provide visual feedback to inform the user when an invalid value is entered, based on a custom validation rule.

Example

The text content of the TextBox in the following example is bound to the `Age` property (of type int) of a binding source object named `ods`. The binding is set up to use a validation rule named `AgeRangeRule` so that if the user enters non-numeric characters or a value that is smaller than 21 or greater than 130, a red exclamation mark appears next to the text box and a tool tip with the error message appears when the user moves the mouse over the text box.

```
<TextBox Name="textBox1" Width="50" FontSize="15"
         Validation.ErrorTemplate="{StaticResource validationTemplate}"
         Style="{StaticResource textBoxInError}"
         Grid.Row="1" Grid.Column="1" Margin="2">
  <TextBox.Text>
    <Binding Path="Age" Source="{StaticResource ods}"
             UpdateSourceTrigger="PropertyChanged" >
      <Binding.ValidationRules>
        <c:AgeRangeRule Min="21" Max="130"/>
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

The following example shows the implementation of `AgeRangeRule`, which inherits from ValidationRule and overrides the Validate method. The Int32.Parse() method is called on the value to make sure that it does not contain any invalid characters. The Validate method returns a ValidationResult that indicates if the value is valid based on whether an exception is caught during the parsing and whether the age value is outside of the lower and upper bounds.

```
public class AgeRangeRule : ValidationRule
{
    private int _min;
    private int _max;

    public AgeRangeRule()
    {
    }

    public int Min
    {
        get { return _min; }
        set { _min = value; }
    }

    public int Max
    {
        get { return _max; }
        set { _max = value; }
    }

    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        int age = 0;

        try
        {
            if (((string)value).Length > 0)
                age = Int32.Parse((String)value);
        }
        catch (Exception e)
        {
            return new ValidationResult(false, "Illegal characters or " + e.Message);
        }

        if ((age < Min) || (age > Max))
        {
            return new ValidationResult(false,
              "Please enter an age in the range: " + Min + " - " + Max + ".");
        }
        else
        {
            return ValidationResult.ValidResult;
        }
    }
}
```

The following example shows the custom ControlTemplate `validationTemplate` that creates a red exclamation mark to notify the user of a validation error. Control templates are used to redefine the appearance of a control.

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

As shown in the following example, the ToolTip that shows the error message is created using the style named `textBoxInError`. If the value of HasError is `true`, the trigger sets the tool tip of the current TextBox to its first validation error. The RelativeSource is set to Self, referring to the current element.

```
<Style x:Key="textBoxInError" TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={x:Static RelativeSource.Self},
                        Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

For the complete example, see Binding Validation Sample.

Note that if you do not provide a custom ErrorTemplate the default error template appears to provide visual feedback to the user when there is a validation error. See "Data Validation" in Data Binding Overview for more information. Also, WPF provides a built-in validation rule that catches exceptions that are thrown during the update of the binding source property. For more information, see ExceptionValidationRule.

See also

- Data Binding Overview
- How-to Topics

How to: Implement Validation Logic on Custom Objects

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to implement validation logic on a custom object and then bind to it.

Example

You can provide validation logic on the business layer if your source object implements IDataErrorInfo, as in the following example, which defines a `Person` object that implements IDataErrorInfo:

```
public class Person : IDataErrorInfo
{
    private int age;

    public int Age
    {
        get { return age; }
        set { age = value; }
    }

    public string Error
    {
        get
        {
            return null;
        }
    }

    public string this[string name]
    {
        get
        {
            string result = null;

            if (name == "Age")
            {
                if (this.age < 0 || this.age > 150)
                {
                    result = "Age must not be less than 0 or greater than 150.";
                }
            }
            return result;
        }
    }
}
```

```
Public Class Person
    Implements IDataErrorInfo

    Private _age As Integer
    Public Property Age() As Integer
        Get
            Return _age
        End Get
        Set(ByVal value As Integer)
            _age = value
        End Set
    End Property

    Public ReadOnly Property [Error]() As String Implements IDataErrorInfo.Error
        Get
            Return Nothing
        End Get
    End Property

    Default Public ReadOnly Property Item(ByVal columnName As String) As String Implements IDataErrorInfo.Item
        Get
            Dim result As String = Nothing

            If columnName = "Age" Then
                If Me._age < 0 OrElse Me._age > 150 Then
                    result = "Age must not be less than 0 or greater than 150."
                End If
            End If
            Return result
        End Get
    End Property
End Class
```

In the following example, the text property of the text box binds to the `Person.Age` property, which has been made available for binding through a resource declaration that is given the `x:Key` `data`. The DataErrorValidationRule checks for the validation errors raised by the IDataErrorInfo implementation.

```
<Window x:Class="BusinessLayerValidation.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF IDataErrorInfo Sample" Width="350" Height="150"
        xmlns:src="clr-namespace:BusinessLayerValidation">

    <Window.Resources>
        <src:Person x:Key="data"/>

        <!--The tool tip for the TextBox to display the validation error message.-->
        <Style x:Key="textBoxInError" TargetType="TextBox">
            <Style.Triggers>
                <Trigger Property="Validation.HasError" Value="true">
                    <Setter Property="ToolTip"
                            Value="{Binding RelativeSource={x:Static RelativeSource.Self},
                        Path=(Validation.Errors)[0].ErrorContent}"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

    <StackPanel Margin="20">
        <TextBlock>Enter your age:</TextBlock>

        <TextBox Style="{StaticResource textBoxInError}">
            <TextBox.Text>
                <!--By setting ValidatesOnExceptions to True, it checks for exceptions
                that are thrown during the update of the source property.
                An alternative syntax is to add <ExceptionValidationRule/> within
                the <Binding.ValidationRules> section.-->
                <Binding Path="Age" Source="{StaticResource data}"
                         ValidatesOnExceptions="True"
                         UpdateSourceTrigger="PropertyChanged">
                    <Binding.ValidationRules>
                        <!--DataErrorValidationRule checks for validation
                            errors raised by the IDataErrorInfo object.-->
                        <!--Alternatively, you can set ValidationOnDataErrors="True" on the Binding.-->
                        <DataErrorValidationRule/>
                    </Binding.ValidationRules>
                </Binding>
            </TextBox.Text>
        </TextBox>

        <TextBlock>Mouse-over to see the validation error message.</TextBlock>
    </StackPanel>
</Window>
```

Alternatively, instead of using the DataErrorValidationRule, you can set the ValidatesOnDataErrors property to `true`.

See also

- ExceptionValidationRule
- Implement Binding Validation
- How-to Topics

How to: Get the Binding Object from a Bound Target Property

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to obtain the binding object from a data-bound target property.

Example

You can do the following to get the Binding object:

```
// textBox3 is an instance of a TextBox
// the TextProperty is the data-bound dependency property
Binding myBinding = BindingOperations.GetBinding(textBox3, TextBox.TextProperty);
```

NOTE

You must specify the dependency property for the binding you want because it is possible that more than one property of the target object is using data binding.

Alternatively, you can get the BindingExpression and then get the value of the ParentBinding property.

For the complete example see Binding Validation Sample.

NOTE

If your binding is a MultiBinding, use BindingOperations.GetMultiBinding. If it is a PriorityBinding, use BindingOperations.GetPriorityBinding. If you are uncertain whether the target property is bound using a Binding, a MultiBinding, or a PriorityBinding, you can use BindingOperations.GetBindingBase.

See also

- Create a Binding in Code
- How-to Topics

How to: Implement a CompositeCollection

1/23/2019 • 2 minutes to read • Edit Online

Example

The following example shows how to display multiple collections and items as one list using the CompositeCollection class. In this example, `GreekGods` is an ObservableCollection<T> of `GreekGod` custom objects. Data templates are defined so that `GreekGod` objects and `GreekHero` objects appear with a gold and a cyan foreground color respectively.

```xml
<Window Background="Cornsilk"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c="clr-namespace:SDKSample"
  x:Class="SDKSample.Window1"
  Title="CompositeCollections"
  SizeToContent="WidthAndHeight"
  >
  <Window.Resources>
    <c:GreekGods x:Key="GreekGodsData"/>

    <XmlDataProvider x:Key="GreekHeroesData" XPath="GreekHeroes/Hero">
      <x:XData>
      <GreekHeroes xmlns="">
        <Hero Name="Jason" />
        <Hero Name="Hercules" />
        <Hero Name="Bellerophon" />
        <Hero Name="Theseus" />
        <Hero Name="Odysseus" />
        <Hero Name="Perseus" />
      </GreekHeroes>
      </x:XData>
    </XmlDataProvider>

    <DataTemplate DataType="{x:Type c:GreekGod}">
      <TextBlock Text="{Binding Path=Name}" Foreground="Gold"/>
    </DataTemplate>
    <DataTemplate DataType="Hero">
      <TextBlock Text="{Binding XPath=@Name}" Foreground="Cyan"/>
    </DataTemplate>
    </Window.Resources>

  <StackPanel>
    <TextBlock FontSize="18" FontWeight="Bold" Margin="10"
      HorizontalAlignment="Center">Composite Collections Sample</TextBlock>
    <ListBox Name="myListBox" Height="300" Width="200" Background="White">
      <ListBox.ItemsSource>
        <CompositeCollection>
          <CollectionContainer
            Collection="{Binding Source={StaticResource GreekGodsData}}" />
          <CollectionContainer
            Collection="{Binding Source={StaticResource GreekHeroesData}}" />
          <ListBoxItem Foreground="Red">Other Listbox Item 1</ListBoxItem>
          <ListBoxItem Foreground="Red">Other Listbox Item 2</ListBoxItem>
        </CompositeCollection>
      </ListBox.ItemsSource>
    </ListBox>
  </StackPanel>

</Window>
```

See also

- CollectionContainer
- ItemsSource
- XmlDataProvider
- DataTemplate
- Data Binding Overview
- How-to Topics

How to: Convert Bound Data

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to apply conversion to data that is used in bindings.

To convert data during binding, you must create a class that implements the IValueConverter interface, which includes the Convert and ConvertBack methods.

Example

The following example shows the implementation of a date converter that converts the date value passed in so that it only shows the year, the month, and the day. When implementing the IValueConverter interface, it is a good practice to decorate the implementation with a ValueConversionAttribute attribute to indicate to development tools the data types involved in the conversion, as in the following example:

```
[ValueConversion(typeof(DateTime), typeof(String))]
public class DateConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        DateTime date = (DateTime)value;
        return date.ToShortDateString();
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        string strValue = value as string;
        DateTime resultDateTime;
        if (DateTime.TryParse(strValue, out resultDateTime))
        {
            return resultDateTime;
        }
        return DependencyProperty.UnsetValue;
    }
}
```

```
Public Class DateConverter
    Implements System.Windows.Data.IValueConverter

    Public Function Convert(ByVal value As Object,
                            ByVal targetType As System.Type,
                            ByVal parameter As Object,
                            ByVal culture As System.Globalization.CultureInfo) _
             As Object Implements System.Windows.Data.IValueConverter.Convert

        Dim DateValue As DateTime = CType(value, DateTime)
        Return DateValue.ToShortDateString

    End Function

    Public Function ConvertBack(ByVal value As Object,
                                ByVal targetType As System.Type,
                                ByVal parameter As Object,
                                ByVal culture As System.Globalization.CultureInfo) _
            As Object Implements System.Windows.Data.IValueConverter.ConvertBack

        Dim strValue As String = value
        Dim resultDateTime As DateTime
        If DateTime.TryParse(strValue, resultDateTime) Then
            Return resultDateTime
        End If
        Return DependencyProperty.UnsetValue

    End Function
End Class
```

Once you have created a converter, you can add it as a resource in your Extensible Application Markup Language (XAML) file. In the following example, *src* maps to the namespace in which *DateConverter* is defined.

```
<src:DateConverter x:Key="dateConverter"/>
```

Finally, you can use the converter in your binding using the following syntax. In the following example, the text content of the TextBlock is bound to *StartDate*, which is a property of an external data source.

```
<TextBlock Grid.Row="2" Grid.Column="0" Margin="0,0,8,0"
           Name="startDateTitle"
           Style="{StaticResource smallTitleStyle}">Start Date:</TextBlock>
<TextBlock Name="StartDateDTKey" Grid.Row="2" Grid.Column="1"
    Text="{Binding Path=StartDate, Converter={StaticResource dateConverter}}"
    Style="{StaticResource textStyleTextBlock}"/>
```

The style resources referenced in the above example are defined in a resource section not shown in this topic.

See also

- Implement Binding Validation
- Data Binding Overview
- How-to Topics

How to: Create a Binding in Code

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to create and set a Binding in code.

Example

The FrameworkElement class and the FrameworkContentElement class both expose a `SetBinding` method. If you are binding an element that inherits either of these classes, you can call the SetBinding method directly.

The following example creates a class named, `MyData`, which contains a property named `MyDataProperty`.

```
public class MyData : INotifyPropertyChanged
{
    private string myDataProperty;

    public MyData() { }

    public MyData(DateTime dateTime)
    {
        myDataProperty = "Last bound time was " + dateTime.ToLongTimeString();
    }

    public String MyDataProperty
    {
        get { return myDataProperty; }
        set
        {
            myDataProperty = value;
            OnPropertyChanged("MyDataProperty");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string info)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(info));
        }
    }
}
```

```
Public Class MyData
    Implements INotifyPropertyChanged

    ' Events
    Public Event PropertyChanged As PropertyChangedEventHandler _
        Implements INotifyPropertyChanged.PropertyChanged

    ' Methods
    Public Sub New()
    End Sub

    Public Sub New(ByVal dateTime As DateTime)
        Me.MyDataProperty = ("Last bound time was " & dateTime.ToLongTimeString)
    End Sub

    Private Sub OnPropertyChanged(ByVal info As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(info))
    End Sub


    ' Properties
    Public Property MyDataProperty As String
        Get
            Return Me._myDataProperty
        End Get
        Set(ByVal value As String)
            Me._myDataProperty = value
            Me.OnPropertyChanged("MyDataProperty")
        End Set
    End Property


    ' Fields
    Private _myDataProperty As String
End Class
```

The following example shows how to create a binding object to set the source of the binding. The example uses SetBinding to bind the Text property of `myText`, which is a TextBlock control, to `MyDataProperty`.

```
// Make a new source.
MyData myDataObject = new MyData(DateTime.Now);
Binding myBinding = new Binding("MyDataProperty");
myBinding.Source = myDataObject;
// Bind the new data source to the myText TextBlock control's Text dependency property.
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

```
' Make a new source.
Dim data1 As New MyData(DateTime.Now)
Dim binding1 As New Binding("MyDataProperty")
binding1.Source = data1
' Bind the new data source to the myText TextBlock control's Text dependency property.
Me.myText.SetBinding(TextBlock.TextProperty, binding1)
```

For the complete code sample, see Code-only Binding Sample.

Instead of calling SetBinding, you can use the SetBinding static method of the BindingOperations class. The following example, calls BindingOperations.SetBinding instead of FrameworkElement.SetBinding to bind `myText` to `myDataProperty`.

```
//make a new source
MyData myDataObject = new MyData(DateTime.Now);
Binding myBinding = new Binding("MyDataProperty");
myBinding.Source = myDataObject;
BindingOperations.SetBinding(myText, TextBlock.TextProperty, myBinding);
```

```
Dim myDataObject As New MyData(DateTime.Now)
Dim myBinding As New Binding("MyDataProperty")
myBinding.Source = myDataObject
BindingOperations.SetBinding(myText, TextBlock.TextProperty, myBinding)
```

See also

- Data Binding Overview
- How-to Topics

How to: Get the Default View of a Data Collection

1/23/2019 • 2 minutes to read • Edit Online

Views allow the same data collection to be viewed in different ways, depending on sorting, filtering, or grouping criteria. Every collection has one shared default view, which is used as the actual binding source when a binding specifies a collection as its source. This example shows how to get the default view of a collection.

Example

To create the view, you need an object reference to the collection. This data object can be obtained by referencing your own code-behind object, by getting the data context, by getting a property of the data source, or by getting a property of the binding. This example shows how to get the DataContext of a data object and use it to directly obtain the default collection view for this collection.

```
myCollectionView = (CollectionView)
    CollectionViewSource.GetDefaultView(rootElem.DataContext);
```

```
myCollectionView = CType(CollectionViewSource.GetDefaultView(rootElem.DataContext), CollectionView)
```

In this example, the root element is a StackPanel. The DataContext is set to *myDataSource*, which refers to a data provider that is an ObservableCollection<T> of *Order* objects.

```
<StackPanel.DataContext>
  <Binding Source="{StaticResource myDataSource}"/>
</StackPanel.DataContext>
```

Alternatively, you can instantiate and bind to your own collection view using the CollectionViewSource class. This collection view is only shared by controls that bind to it directly. For an example, see the How to Create a View section in the Data Binding Overview.

For examples of the functionality provided by a collection view, see Sort Data in a View, Filter Data in a View, and Navigate Through the Objects in a Data CollectionView.

See also

- Sort and Group Data Using a View in XAML
- How-to Topics

How to: Navigate Through the Objects in a Data CollectionView

1/23/2019 • 2 minutes to read • Edit Online

Views allow the same data collection to be viewed in different ways, depending on sorting, filtering, or grouping. Views also provide a current record pointer concept and enable moving the pointer. This example shows how to get the current object as well as navigate through the objects in a data collection using the functionality provided in the CollectionView class.

Example

In this example, `myCollectionView` is a CollectionView object that is a view over a bound collection.

In the following example, `OnButton` is an event handler for the `Previous` and `Next` buttons in an application, which are buttons that allow the user to navigate the data collection. Note that the IsCurrentBeforeFirst and IsCurrentAfterLast properties report whether the current record pointer has come to the beginning and the end of the list respectively so that MoveCurrentToFirst and MoveCurrentToLast can be called as appropriately.

The CurrentItem property of the view is cast as an `Order` to return the current order item in the collection.

```
//OnButton is called whenever the Next or Previous buttons
//are clicked to change the currency
  private void OnButton(Object sender, RoutedEventArgs args)
  {
      Button b = sender as Button;

      switch (b.Name)
      {
          case "Previous":
              myCollectionView.MoveCurrentToPrevious();

              if (myCollectionView.IsCurrentBeforeFirst)
              {
                  myCollectionView.MoveCurrentToLast();
              }
              break;

          case "Next":
              myCollectionView.MoveCurrentToNext();
              if (myCollectionView.IsCurrentAfterLast)
              {
                  myCollectionView.MoveCurrentToFirst();
              }
              break;

          o = myCollectionView.CurrentItem as Order;
          // TODO: do something with the current Order o
      }
  }
```

```
'OnButton is called whenever the Next or Previous buttons
'are clicked to change the currency
  Private Sub OnButton(ByVal sender As Object, ByVal args As RoutedEventArgs)
      Dim b As Button = TryCast(sender, Button)

      Select Case b.Name
          Case "Previous"
              myCollectionView.MoveCurrentToPrevious()

              If myCollectionView.IsCurrentBeforeFirst Then
                  myCollectionView.MoveCurrentToLast()
              End If

          Case "Next"
              myCollectionView.MoveCurrentToNext()
              If myCollectionView.IsCurrentAfterLast Then
                  myCollectionView.MoveCurrentToFirst()
              End If
              Exit Select

          o = TryCast(myCollectionView.CurrentItem, Order)
          ' TODO: do something with the current Order o
      End Select
  End Sub
```

See also

- Data Binding Overview
- Sort Data in a View
- Filter Data in a View
- Sort and Group Data Using a View in XAML
- How-to Topics

How to: Filter Data in a View

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to filter data in a view.

Example

To create a filter, define a method that provides the filtering logic. The method is used as a callback and accepts a parameter of type `object`. The following method returns all the `Order` objects with the `filled` property set to "No", filtering out the rest of the objects.

```
public bool Contains(object de)
{
    Order order = de as Order;
    //Return members whose Orders have not been filled
    return(order.Filled== "No");
}
```

```
Public Function Contains(ByVal de As Object) As Boolean
    Dim order1 As Order = TryCast(de, Order)
    Return (order1.Filled Is "No")
End Function
```

You can then apply the filter, as shown in the following example. In this example, `myCollectionView` is a ListCollectionView object.

```
myCollectionView.Filter = new Predicate<object>(Contains);
```

```
Me.myCollectionView.Filter = New Predicate(Of Object)(AddressOf Me.Contains)
```

To undo filtering, you can set the Filter property to `null`:

```
myCollectionView.Filter = null;
```

```
Me.myCollectionView.Filter = Nothing
```

For information about how to create or obtain a view, see Get the Default View of a Data Collection. For the complete example, see Sorting and Filtering Items in a View Sample.

If your view object comes from a CollectionViewSource object, you apply filtering logic by setting an event handler for the Filter event. In the following example, `listingDataView` is an instance of CollectionViewSource.

```
listingDataView.Filter += new FilterEventHandler(ShowOnlyBargainsFilter);
```

```
AddHandler listingDataView.Filter, AddressOf ShowOnlyBargainsFilter
```

The following shows the implementation of the example `ShowOnlyBargainsFilter` filter event handler. This event handler uses the Accepted property to filter out `AuctionItem` objects that have a `CurrentPrice` of $25 or greater.

```
private void ShowOnlyBargainsFilter(object sender, FilterEventArgs e)
{
    AuctionItem product = e.Item as AuctionItem;
    if (product != null)
    {
        // Filter out products with price 25 or above
        if (product.CurrentPrice < 25)
        {
            e.Accepted = true;
        }
        else
        {
            e.Accepted = false;
        }
    }
}
```

```
Private Sub ShowOnlyBargainsFilter(ByVal sender As Object, ByVal e As FilterEventArgs)
    Dim product As AuctionItem = CType(e.Item, AuctionItem)
    If Not (product Is Nothing) Then
        'Filter out products with price 25 or above
        If product.CurrentPrice < 25 Then
            e.Accepted = True
        Else
            e.Accepted = False
        End If
    End If
End Sub
```

See also

- CanFilter
- CustomFilter
- Data Binding Overview
- Sort Data in a View
- How-to Topics

How to: Sort Data in a View

1/23/2019 • 2 minutes to read • Edit Online

This example describes how to sort data in a view.

Example

The following example creates a simple ListBox and a Button:

```
<Window x:Class="ListBoxSort_snip.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ListBoxSort_snip" Height="300" Width="300">
    <DockPanel>
      <ListBox Name="myListBox" DockPanel.Dock="Top">
        <ListBoxItem>my</ListBoxItem>
        <!--Or you can set the content this way:-->
        <!--<ListBoxItem Content="my"/>-->
        <ListBoxItem>1</ListBoxItem>
        <ListBoxItem>Sort</ListBoxItem>
        <ListBoxItem>3</ListBoxItem>
        <ListBoxItem>ListBox</ListBoxItem>
        <ListBoxItem>2</ListBoxItem>
      </ListBox>
      <Button Click="OnClick" Width="30" Height="20" DockPanel.Dock="Top">Sort</Button>
    </DockPanel>
</Window>
```

The Click event handler of the button contains logic to sort the items in the ListBox in the descending order. You can do this because adding items to a ListBox this way adds them to the ItemCollection of the ListBox, and ItemCollection derives from the CollectionView class. If you are binding your ListBox to a collection using the ItemsSource property, you can use the same technique to sort.

```
private void OnClick(object sender, RoutedEventArgs e)
{
    myListBox.Items.SortDescriptions.Add(
        new SortDescription("Content", ListSortDirection.Descending));
}
```

```
Private Sub OnClick(ByVal sender As Object, ByVal e As RoutedEventArgs)
    myListBox.Items.SortDescriptions.Add(New SortDescription("Content", ListSortDirection.Descending))
End Sub
```

As long as you have a reference to the view object, you can use the same technique to sort the content of other collection views. For an example of how to obtain a view, see Get the Default View of a Data Collection. For another example, see Sort a GridView Column When a Header Is Clicked. For more information about views, see Binding to Collections in Data Binding Overview.

For an example of how to apply sorting logic in Extensible Application Markup Language (XAML), see Sort and Group Data Using a View in XAML.

See also

- CustomSort
- Sort a GridView Column When a Header Is Clicked
- Data Binding Overview
- Filter Data in a View
- How-to Topics

How to: Sort and Group Data Using a View in XAML

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to create a view of a data collection in Extensible Application Markup Language (XAML). Views allow for the functionalities of grouping, sorting, filtering, and the notion of a current item.

Example

In the following example, the static resource named *places* is defined as a collection of *Place* objects, in which each *Place* object is consisted of a city name and the state. The prefix *src* is mapped to the namespace where the data source *Places* is defined. The prefix *scm* maps to `"clr-namespace:System.ComponentModel;assembly=WindowsBase"` and *dat* maps to `"clr-namespace:System.Windows.Data;assembly=PresentationFramework"`.

The following example creates a view of the data collection that is sorted by the city name and grouped by the state.

```
<Window.Resources>

  <src:Places x:Key="places"/>

  <CollectionViewSource Source="{StaticResource places}" x:Key="cvs">
    <CollectionViewSource.SortDescriptions>
      <scm:SortDescription PropertyName="CityName"/>
    </CollectionViewSource.SortDescriptions>
    <CollectionViewSource.GroupDescriptions>
      <dat:PropertyGroupDescription PropertyName="State"/>
    </CollectionViewSource.GroupDescriptions>
  </CollectionViewSource>
```

The view can then be a binding source, as in the following example:

```
<ListBox ItemsSource="{Binding Source={StaticResource cvs}}"
         DisplayMemberPath="CityName" Name="lb">
  <ListBox.GroupStyle>
    <x:Static Member="GroupStyle.Default"/>
  </ListBox.GroupStyle>
</ListBox>
```

For bindings to XML data defined in an XmlDataProvider resource, precede the XML name with an @ symbol.

```
<XmlDataProvider x:Key="myTasks" XPath="Tasks/Task">
    <x:XData>
        <Tasks xmlns="">
            <Task Name="Groceries" Priority="2" Type="Home">
```

```
<CollectionViewSource x:Key="mySortedTasks"
                      Source="{StaticResource myTasks}">
    <CollectionViewSource.SortDescriptions>
        <scm:SortDescription PropertyName="@Priority" />
    </CollectionViewSource.SortDescriptions>
    <CollectionViewSource.GroupDescriptions>
        <dat:PropertyGroupDescription PropertyName="@Priority" />
    </CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

See also

- CollectionViewSource
- Get the Default View of a Data Collection
- Data Binding Overview
- How-to Topics

How to: Use the Master-Detail Pattern with Hierarchical Data

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to implement the master-detail scenario.

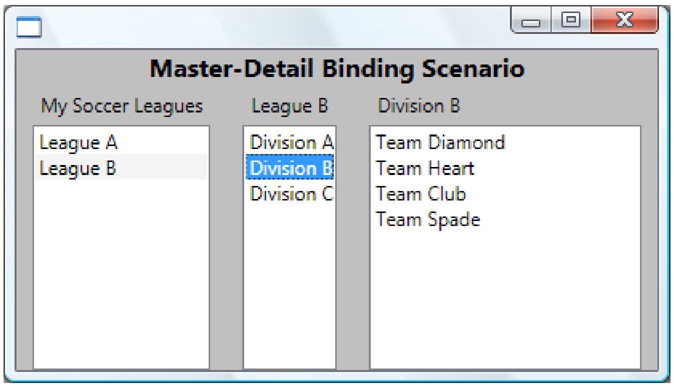
Example

In this example, `LeagueList` is a collection of `Leagues`. Each `League` has a `Name` and a collection of `Divisions`, and each `Division` has a name and a collection of `Teams`. Each `Team` has a team name.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  Width="400" Height="180"
  Title="Master-Detail Binding"
  Background="Silver">
  <Window.Resources>
    <src:LeagueList x:Key="MyList"/>
```

```
  <DockPanel DataContext="{Binding Source={StaticResource MyList}}">
    <StackPanel>
      <Label>My Soccer Leagues</Label>
      <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name"
               IsSynchronizedWithCurrentItem="true"/>
    </StackPanel>

    <StackPanel>
      <Label Content="{Binding Path=Name}"/>
      <ListBox ItemsSource="{Binding Path=Divisions}" DisplayMemberPath="Name"
               IsSynchronizedWithCurrentItem="true"/>
    </StackPanel>

    <StackPanel>
      <Label Content="{Binding Path=Divisions/Name}"/>
      <ListBox DisplayMemberPath="Name" ItemsSource="{Binding Path=Divisions/Teams}"/>
    </StackPanel>
  </DockPanel>
</Window>
```

The following is a screenshot of the example. The `Divisions` ListBox automatically tracks selections in the `Leagues` ListBox and display the corresponding data. The `Teams` ListBox tracks selections in the other two ListBox controls.



The two things to notice in this example are:

1. The three ListBox controls bind to the same source. You set the Path property of the binding to specify which level of data you want the ListBox to display.

2. You must set the IsSynchronizedWithCurrentItem property to `true` on the ListBox controls of which the selection you are tracking. Setting this property ensures that the selected item is always set as the CurrentItem. Alternatively, if the ListBox gets it data from a CollectionViewSource, it synchronizes selection and currency automatically.

The technique is slightly different when you are using XML data. For an example, see Use the Master-Detail Pattern with Hierarchical XML Data.

See also

- HierarchicalDataTemplate
- Bind to a Collection and Display Information Based on Selection
- Data Binding Overview
- Data Templating Overview
- How-to Topics

How to: Use the Master-Detail Pattern with Hierarchical XML Data

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to implement the master-detail scenario with XML data.

Example

This example is the XML data version of the example discussed in Use the Master-Detail Pattern with Hierarchical Data. In this example, the data is from the file `League.xml`. Note how the third ListBox control tracks selection changes in the second ListBox by binding to its SelectedValue property.

```
<Window x:Class="SDKSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Multiple ListBox Binding Sample"
  Width="400" Height="200"
  Background="Cornsilk">
    <Window.Resources>
      <XmlDataProvider x:Key="MyList" Source="Data\Leagues.xml"
                       XPath="Leagues/League"/>
      <DataTemplate x:Key="dataTemplate">
        <TextBlock Text="{Binding XPath=@name}" />
      </DataTemplate>
```

```
    </Window.Resources>

    <DockPanel DataContext="{Binding Source={StaticResource MyList}}">
      <StackPanel>
        <Label>My Soccer Leagues</Label>
        <ListBox ItemsSource="{Binding}"
                 ItemTemplate="{StaticResource dataTemplate}"
                 IsSynchronizedWithCurrentItem="true"/>
      </StackPanel>

      <StackPanel>
        <Label Content="{Binding XPath=@name}"/>
        <ListBox Name="divisionsListBox"
                 ItemsSource="{Binding XPath=Division}"
                 ItemTemplate="{StaticResource dataTemplate}"
                 IsSynchronizedWithCurrentItem="true"/>
      </StackPanel>

      <StackPanel>
        <Label Content="{Binding XPath=@name}"/>
        <ListBox DataContext="{Binding ElementName=divisionsListBox,
                                       Path=SelectedItem}"
                 ItemsSource="{Binding XPath=Team}"
                 ItemTemplate="{StaticResource dataTemplate}"/>
      </StackPanel>
    </DockPanel>
</Window>
```

See also

- HierarchicalDataTemplate
- How-to Topics

How to: Produce a Value Based on a List of Bound Items

1/23/2019 • 2 minutes to read • Edit Online

MultiBinding allows you to bind a binding target property to a list of source properties and then apply logic to produce a value with the given inputs. This example demonstrates how to use MultiBinding.

Example

In the following example, `NameListData` refers to a collection of `PersonName` objects, which are objects that contain two properties, `firstName` and `lastName`. The following example produces a TextBlock that shows the first and last names of a person with the last name first.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c="clr-namespace:SDKSample"
  x:Class="SDKSample.Window1"
  Width="400"
  Height="280"
  Title="MultiBinding Sample">

  <Window.Resources>
    <c:NameList x:Key="NameListData"/>
    <c:NameConverter x:Key="myNameConverter"/>
```

```
</Window.Resources>
```

```
<TextBlock Name="textBox2" DataContext="{StaticResource NameListData}">
  <TextBlock.Text>
    <MultiBinding Converter="{StaticResource myNameConverter}"
                  ConverterParameter="FormatLastFirst">
      <Binding Path="FirstName"/>
      <Binding Path="LastName"/>
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

```
</Window>
```

To understand how the last-name-first format is produced, let's take a look at the implementation of the `NameConverter`:

```csharp
public class NameConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)
    {
        string name;

        switch ((string)parameter)
        {
            case "FormatLastFirst":
                name = values[1] + ", " + values[0];
                break;
            case "FormatNormal":
            default:
                name = values[0] + " " + values[1];
                break;
        }

        return name;
    }

    public object[] ConvertBack(object value, Type[] targetTypes, object parameter, CultureInfo culture)
    {
        string[] splitValues = ((string)value).Split(' ');
        return splitValues;
    }
}
```

```vb
Public Class NameConverter
    Implements IMultiValueConverter

Public Function Convert1(ByVal values() As Object, _
                         ByVal targetType As System.Type, _
                         ByVal parameter As Object, _
                         ByVal culture As System.Globalization.CultureInfo) As Object _
                         Implements System.Windows.Data.IMultiValueConverter.Convert
    Select Case CStr(parameter)
        Case "FormatLastFirst"
            Return (values(1) & ", " & values(0))
    End Select
    Return (values(0) & " " & values(1))
End Function

Public Function ConvertBack1(ByVal value As Object, _
                             ByVal targetTypes() As System.Type, _
                             ByVal parameter As Object, _
                             ByVal culture As System.Globalization.CultureInfo) As Object() _
                             Implements System.Windows.Data.IMultiValueConverter.ConvertBack
    Return CStr(value).Split(New Char() {" "c})
End Function
End Class
```

`NameConverter` implements the IMultiValueConverter interface. `NameConverter` takes the values from the individual bindings and stores them in the values object array. The order in which the Binding elements appear under the MultiBinding element is the order in which those values are stored in the array. The value of the ConverterParameter attribute is referenced by the parameter argument of the Converter method, which performs a switch on the parameter to determine how to format the name.

See also

- Convert Bound Data
- Data Binding Overview

- How-to Topics

How to: Implement Property Change Notification

1/23/2019 • 2 minutes to read • Edit Online

To support OneWay or TwoWay binding to enable your binding target properties to automatically reflect the dynamic changes of the binding source (for example, to have the preview pane updated automatically when the user edits a form), your class needs to provide the proper property changed notifications. This example shows how to create a class that implements INotifyPropertyChanged.

Example

To implement INotifyPropertyChanged you need to declare the PropertyChanged event and create the `OnPropertyChanged` method. Then for each property you want change notifications for, you call `OnPropertyChanged` whenever the property is updated.

```
using System.ComponentModel;

namespace SDKSample
{
  // This class implements INotifyPropertyChanged
  // to support one-way and two-way bindings
  // (such that the UI element updates when the source
  // has been changed dynamically)
  public class Person : INotifyPropertyChanged
  {
      private string name;
      // Declare the event
      public event PropertyChangedEventHandler PropertyChanged;

      public Person()
      {
      }

      public Person(string value)
      {
          this.name = value;
      }

      public string PersonName
      {
          get { return name; }
          set
          {
              name = value;
              // Call OnPropertyChanged whenever the property is updated
              OnPropertyChanged("PersonName");
          }
      }

      // Create the OnPropertyChanged method to raise the event
      protected void OnPropertyChanged(string name)
      {
          PropertyChangedEventHandler handler = PropertyChanged;
          if (handler != null)
          {
              handler(this, new PropertyChangedEventArgs(name));
          }
      }
  }
}
```

```vb
Imports System.ComponentModel

' This class implements INotifyPropertyChanged
' to support one-way and two-way bindings
' (such that the UI element updates when the source
' has been changed dynamically)
Public Class Person
    Implements INotifyPropertyChanged

    Private personName As String

    Sub New()
    End Sub

    Sub New(ByVal Name As String)
        Me.personName = Name
    End Sub

    ' Declare the event
    Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged

    Public Property Name() As String
        Get
            Return personName
        End Get
        Set(ByVal value As String)
            personName = value
            ' Call OnPropertyChanged whenever the property is updated
            OnPropertyChanged("Name")
        End Set
    End Property

    ' Create the OnPropertyChanged method to raise the event
    Protected Sub OnPropertyChanged(ByVal name As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(name))
    End Sub

End Class
```

To see an example of how the `Person` class can be used to support TwoWay binding, see Control When the TextBox Text Updates the Source.

See also

- Binding Sources Overview
- Data Binding Overview
- How-to Topics

How to: Create and Bind to an ObservableCollection

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to create and bind to a collection that derives from the ObservableCollection<T> class, which is a collection class that provides notifications when items get added or removed.

Example

The following example shows the implementation of a `NameList` collection:

```
public class NameList : ObservableCollection<PersonName>
{
    public NameList() : base()
    {
        Add(new PersonName("Willa", "Cather"));
        Add(new PersonName("Isak", "Dinesen"));
        Add(new PersonName("Victor", "Hugo"));
        Add(new PersonName("Jules", "Verne"));
    }
  }

  public class PersonName
  {
      private string firstName;
      private string lastName;

      public PersonName(string first, string last)
      {
          this.firstName = first;
          this.lastName = last;
      }

      public string FirstName
      {
          get { return firstName; }
          set { firstName = value; }
      }

      public string LastName
      {
          get { return lastName; }
          set { lastName = value; }
      }
  }
```

```
Public Class NameList
    Inherits ObservableCollection(Of PersonName)

    ' Methods
    Public Sub New()
        MyBase.Add(New PersonName("Willa", "Cather"))
        MyBase.Add(New PersonName("Isak", "Dinesen"))
        MyBase.Add(New PersonName("Victor", "Hugo"))
        MyBase.Add(New PersonName("Jules", "Verne"))
    End Sub

End Class

Public Class PersonName
    ' Methods
    Public Sub New(ByVal first As String, ByVal last As String)
        Me._firstName = first
        Me._lastName = last
    End Sub

    ' Properties
    Public Property FirstName() As String
        Get
            Return Me._firstName
        End Get
        Set(ByVal value As String)
            Me._firstName = value
        End Set
    End Property

    Public Property LastName() As String
        Get
            Return Me._lastName
        End Get
        Set(ByVal value As String)
            Me._lastName = value
        End Set
    End Property

    ' Fields
    Private _firstName As String
    Private _lastName As String
End Class
```

You can make the collection available for binding the same way you would with other common language runtime (CLR) objects, as described in Make Data Available for Binding in XAML. For example, you can instantiate the collection in XAML and specify the collection as a resource, as shown here:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c="clr-namespace:SDKSample"
  x:Class="SDKSample.Window1"
  Width="400"
  Height="280"
  Title="MultiBinding Sample">

  <Window.Resources>
    <c:NameList x:Key="NameListData"/>

...

</Window.Resources>
```

You can then bind to the collection:

```
<ListBox Width="200"
         ItemsSource="{Binding Source={StaticResource NameListData}}"
         ItemTemplate="{StaticResource NameItemTemplate}"
         IsSynchronizedWithCurrentItem="True"/>
```

The definition of `NameItemTemplate` is not shown here.

NOTE

The objects in your collection must satisfy the requirements described in the Binding Sources Overview. In particular, if you are using OneWay or TwoWay (for example, you want your UI to update when the source properties change dynamically), you must implement a suitable property changed notification mechanism such as the INotifyPropertyChanged interface.

For more information, see the Binding to Collections section in the Data Binding Overview.

See also

- Sort Data in a View
- Filter Data in a View
- Sort and Group Data Using a View in XAML
- Data Binding Overview
- How-to Topics

How to: Implement PriorityBinding

1/23/2019 • 2 minutes to read • Edit Online

PriorityBinding in Windows Presentation Foundation (WPF) works by specifying a list of bindings. The list of bindings is ordered from highest priority to lowest priority. If the highest priority binding returns a value successfully when it is processed then there is never a need to process the other bindings in the list. It could be the case that the highest priority binding takes a long time to be evaluated, the next highest priority that returns a value successfully will be used until a binding of a higher priority returns a value successfully.

Example

To demonstrate how PriorityBinding works, the `AsyncDataSource` object has been created with the following three properties: `FastDP`, `SlowerDP`, and `SlowestDP`.

The get accessor of `FastDP` returns the value of the `_fastDP` data member.

The get accessor of `SlowerDP` waits for 3 seconds before returning the value of the `_slowerDP` data member.

The get accessor of `SlowestDP` waits for 5 seconds before returning the value of the `_slowestDP` data member.

NOTE

This example is for demonstration purposes only. The Microsoft .NET guidelines recommend against defining properties that are orders of magnitude slower than a field set would be. For more information, see NIB: Choosing Between Properties and Methods.

```
public class AsyncDataSource
{
  private string _fastDP;
  private string _slowerDP;
  private string _slowestDP;

  public AsyncDataSource()
  {
  }

  public string FastDP
  {
    get { return _fastDP; }
    set { _fastDP = value; }
  }

  public string SlowerDP
  {
    get
    {
      // This simulates a lengthy time before the
      // data being bound to is actualy available.
      Thread.Sleep(3000);
      return _slowerDP;
    }
    set { _slowerDP = value; }
  }

  public string SlowestDP
  {
    get
    {
      // This simulates a lengthy time before the
      // data being bound to is actualy available.
      Thread.Sleep(5000);
      return _slowestDP;
    }
    set { _slowestDP = value; }
  }
}
```

```
Public Class AsyncDataSource
    ' Properties
    Public Property FastDP As String
        Get
            Return Me._fastDP
        End Get
        Set(ByVal value As String)
            Me._fastDP = value
        End Set
    End Property

    Public Property SlowerDP As String
        Get
            Thread.Sleep(3000)
            Return Me._slowerDP
        End Get
        Set(ByVal value As String)
            Me._slowerDP = value
        End Set
    End Property

    Public Property SlowestDP As String
        Get
            Thread.Sleep(5000)
            Return Me._slowestDP
        End Get
        Set(ByVal value As String)
            Me._slowestDP = value
        End Set
    End Property


    ' Fields
    Private _fastDP As String
    Private _slowerDP As String
    Private _slowestDP As String
End Class
```

The Text property binds to the above `AsyncDS` using PriorityBinding:

```
<Window.Resources>
  <c:AsyncDataSource SlowestDP="Slowest Value" SlowerDP="Slower Value"
                     FastDP="Fast Value" x:Key="AsyncDS" />
</Window.Resources>

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
  DataContext="{Binding Source={StaticResource AsyncDS}}">
  <TextBlock FontSize="18" FontWeight="Bold" Margin="10"
    HorizontalAlignment="Center">Priority Binding</TextBlock>
  <TextBlock Background="Honeydew" Width="100" HorizontalAlignment="Center">
    <TextBlock.Text>
      <PriorityBinding FallbackValue="defaultvalue">
        <Binding Path="SlowestDP" IsAsync="True"/>
        <Binding Path="SlowerDP" IsAsync="True"/>
        <Binding Path="FastDP" />
      </PriorityBinding>
    </TextBlock.Text>
  </TextBlock>
</StackPanel>
```

When the binding engine processes the Binding objects, it starts with the first Binding, which is bound to the `SlowestDP` property. When this Binding is processed, it does not return a value successfully because it is sleeping for 5 seconds, so the next Binding element is processed. The next Binding does not return a value successfully because it is sleeping for 3 seconds. The binding engine then moves onto the next Binding element, which is bound to the `FastDP` property. This Binding returns the value "Fast Value". The TextBlock now displays the value "Fast Value".

After 3 seconds elapses, the `SlowerDP` property returns the value "Slower Value". The TextBlock then displays the value "Slower Value".

After 5 seconds elapses, the `SlowestDP` property returns the value "Slowest Value". That binding has the highest priority because it is listed first. The TextBlock now displays the value "Slowest Value".

See PriorityBinding for information about what is considered a successful return value from a binding.

See also

- Binding.IsAsync
- Data Binding Overview
- How-to Topics

How to: Bind to XML Data Using an XMLDataProvider and XPath Queries

1/23/2019 • 3 minutes to read • Edit Online

This example shows how to bind to XML data using an XmlDataProvider.

With an XmlDataProvider, the underlying data that can be accessed through data binding in your application can be any tree of XML nodes. In other words, an XmlDataProvider provides a convenient way to use any tree of XML nodes as a binding source.

Example

In the following example, the data is embedded directly as an XML *data island* within the Resources section. An XML data island must be wrapped in `<x:XData>` tags and always have a single root node, which is *Inventory* in this example.

NOTE

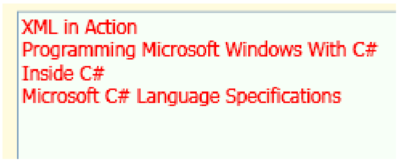
The root node of the XML data has an **xmlns** attribute that sets the XML namespace to an empty string. This is a requirement for applying XPath queries to a data island that is inline within the XAML page. In this inline case, the XAML, and thus the data island, inherits the System.Windows namespace. Because of this, you need to set the namespace blank to keep XPath queries from being qualified by the System.Windows namespace, which would misdirect the queries.

```
<StackPanel
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Background="Cornsilk">

  <StackPanel.Resources>
    <XmlDataProvider x:Key="InventoryData" XPath="Inventory/Books">
      <x:XData>
        <Inventory xmlns="">
          <Books>
            <Book ISBN="0-7356-0562-9" Stock="in" Number="9">
              <Title>XML in Action</Title>
              <Summary>XML Web Technology</Summary>
            </Book>
            <Book ISBN="0-7356-1370-2" Stock="in" Number="8">
              <Title>Programming Microsoft Windows With C#</Title>
              <Summary>C# Programming using the .NET Framework</Summary>
            </Book>
            <Book ISBN="0-7356-1288-9" Stock="out" Number="7">
              <Title>Inside C#</Title>
              <Summary>C# Language Programming</Summary>
            </Book>
            <Book ISBN="0-7356-1377-X" Stock="in" Number="5">
              <Title>Introducing Microsoft .NET</Title>
              <Summary>Overview of .NET Technology</Summary>
            </Book>
            <Book ISBN="0-7356-1448-2" Stock="out" Number="4">
              <Title>Microsoft C# Language Specifications</Title>
              <Summary>The C# language definition</Summary>
            </Book>
          </Books>
          <CDs>
            <CD Stock="in" Number="3">
```

```
              <Title>Classical Collection</Title>
              <Summary>Classical Music</Summary>
            </CD>
            <CD Stock="out" Number="9">
              <Title>Jazz Collection</Title>
              <Summary>Jazz Music</Summary>
            </CD>
          </CDs>
        </Inventory>
      </x:XData>
    </XmlDataProvider>
  </StackPanel.Resources>

  <TextBlock FontSize="18" FontWeight="Bold" Margin="10"
    HorizontalAlignment="Center">XML Data Source Sample</TextBlock>
  <ListBox
    Width="400" Height="300" Background="Honeydew">
    <ListBox.ItemsSource>
      <Binding Source="{StaticResource InventoryData}"
               XPath="*[@Stock='out'] | *[@Number>=8 or @Number=3]"/>
    </ListBox.ItemsSource>

    <!--Alternatively, you can do the following. -->
    <!--<ListBox Width="400" Height="300" Background="Honeydew"
      ItemsSource="{Binding Source={StaticResource InventoryData},
      XPath=*[@Stock\=\'out\'] | *[@Number>\=8 or @Number\=3]}">-->

    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock FontSize="12" Foreground="Red">
          <TextBlock.Text>
            <Binding XPath="Title"/>
          </TextBlock.Text>
        </TextBlock>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</StackPanel>
```

As shown in this example, to create the same binding declaration in attribute syntax you must escape the special characters properly. For more information, see XML Character Entities and XAML.

The ListBox will show the following items when this example is run. These are the *Title*s of all of the elements under *Books* with either a *Stock* value of "*out*" or a *Number* value of 3 or greater than or equals to 8. Notice that no *CD* items are returned because the XPath value set on the XmlDataProvider indicates that only the *Books* elements should be exposed (essentially setting a filter).


XML in Action
Programming Microsoft Windows With C#
Inside C#
Microsoft C# Language Specifications

In this example, the book titles are displayed because the XPath of the TextBlock binding in the DataTemplate is set to "*Title*". If you want to display the value of an attribute, such as the *ISBN*, you would set that XPath value to " `@ISBN` ".

The **XPath** properties in WPF are handled by the XmlNode.SelectNodes method. You can modify the **XPath** queries to get different results. Here are some examples for the XPath query on the bound ListBox from the previous example:

- `XPath="Book[1]"` will return the first book element ("XML in Action"). Note that **XPath** indexes are based on 1, not 0.

- `XPath="Book[@*]"` will return all book elements with any attributes.
- `XPath="Book[last()-1]"` will return the second to last book element ("Introducing Microsoft .NET").
- `XPath="*[position()>3]"` will return all of the book elements except for the first 3.

When you run an **XPath** query, it returns an XmlNode or a list of XmlNodes. XmlNode is a common language runtime (CLR) object, which means you can use the Path property to bind to the common language runtime (CLR) properties. Consider the previous example again. If the rest of the example stays the same and you change the TextBlock binding to the following, you will see the names of the returned XmlNodes in the ListBox. In this case, the name of all the returned nodes is "*Book*".

```
<TextBlock FontSize="12" Foreground="Red">
  <TextBlock.Text>
    <Binding Path="Name"/>
  </TextBlock.Text>
</TextBlock>
```

In some applications, embedding the XML as a data island within the source of the XAML page can be inconvenient because the exact content of the data must be known at compile time. Therefore, obtaining the data from an external XML file is also supported, as in the following example:

```
<XmlDataProvider x:Key="BookData" Source="data\bookdata.xml" XPath="Books"/>
```

If the XML data resides in a remote XML file, you would define access to the data by assigning an appropriate URL to the Source attribute as follows:

```
<XmlDataProvider x:Key="BookData" Source="http://MyUrl" XPath="Books"/>
```

See also

- ObjectDataProvider
- Bind to XDocument, XElement, or LINQ for XML Query Results
- Use the Master-Detail Pattern with Hierarchical XML Data
- Binding Sources Overview
- Data Binding Overview
- How-to Topics

How to: Bind to XDocument, XElement, or LINQ for XML Query Results

1/23/2019 • 2 minutes to read • Edit Online

This example demonstrates how to bind XML data to an ItemsControl using XDocument.

Example

The following XAML code defines an ItemsControl and includes a data template for data of type `Planet` in the `http://planetsNS` XML namespace. An XML data type that occupies a namespace must include the namespace in braces, and if it appears where a XAML markup extension could appear, it must precede the namespace with a brace escape sequence. This code binds to dynamic properties that correspond to the Element and Attribute methods of the XElement class. Dynamic properties enable XAML to bind to dynamic properties that share the names of methods. To learn more, see LINQ to XML Dynamic Properties. Notice how the default namespace declaration for the XML does not apply to attribute names.

```
<StackPanel Name="stacky">
  <StackPanel.Resources>
    <DataTemplate DataType="{}{http://planetsNS}Planet" >
      <StackPanel Orientation="Horizontal">
        <TextBlock Width="100" Text="{Binding Path=Element[{http://planetsNS}DiameterKM].Value}" />
        <TextBlock Width="100" Text="{Binding Path=Attribute[Name].Value}" />
        <TextBlock Text="{Binding Path=Element[{http://planetsNS}Details].Value}" />
      </StackPanel>
    </DataTemplate>
  </StackPanel.Resources>
```

```
  <ItemsControl
    ItemsSource="{Binding }" >
  </ItemsControl>
</StackPanel>
```

The following C# code calls Load and sets the stack panel data context to all subelements of the element named `SolarSystemPlanets` in the `http://planetsNS` XML namespace.

```
planetsDoc = XDocument.Load("../../Planets.xml");
stacky.DataContext = planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements();
```

```
planetsDoc = XDocument.Load("../../Planets.xml")
stacky.DataContext = planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements()
```

XML data can be stored as a XAML resource using ObjectDataProvider. For a complete example, see L2DBForm.xaml Source Code. The following sample shows how code can set the data context to an object resource.

```
planetsDoc = (XDocument)((ObjectDataProvider)Resources["justTwoPlanets"]).Data;
stacky.DataContext = planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements();
```

```
planetsDoc = CType((CType(Resources("justTwoPlanets"), ObjectDataProvider)).Data, XDocument)
stacky.DataContext = planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements()
```

The dynamic properties that map to Element and Attribute provide flexibility within XAML. Your code can also bind to the results of a LINQ for XML query. This example binds to query results ordered by an element value.

```
stacky.DataContext =
from c in planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements()
orderby Int32.Parse(c.Element("{http://planetsNS}DiameterKM").Value)
select c;
```

```
stacky.DataContext = From c In planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements()
                     Order By Int32.Parse(c.Element("{http://planetsNS}DiameterKM").Value)
                     Select c
```

See also

- Binding Sources Overview
- WPF Data Binding with LINQ to XML Overview
- WPF Data Binding Using LINQ to XML Example
- LINQ to XML Dynamic Properties

How to: Use XML Namespaces in Data Binding

1/23/2019 • 2 minutes to read • Edit Online

Example

This example shows how to handle namespaces specified in your XML binding source.

If your XML data has the following XML namespace definition:

```
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
```

You can use the XmlNamespaceMapping element to map the namespace to a Prefix, as in the following example. You can then use the Prefix to refer to the XML namespace. The ListBox in this example displays the *title* and *dc:date* of each *item*.

```
<StackPanel.Resources>
  <XmlNamespaceMappingCollection x:Key="mapping">
    <XmlNamespaceMapping Uri="http://purl.org/dc/elements/1.1/" Prefix="dc" />
  </XmlNamespaceMappingCollection>

  <XmlDataProvider Source="http://msdn.microsoft.com/subscriptions/rss.xml"
                   XmlNamespaceManager="{StaticResource mapping}"
                   XPath="rss/channel/item" x:Key="provider"/>

  <DataTemplate x:Key="dataTemplate">
    <Border BorderThickness="1" BorderBrush="Gray">
      <Grid Width="600" Height="50">
        <Grid.RowDefinitions>
          <RowDefinition Height="25"/>
          <RowDefinition Height="25"/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Text="{Binding XPath=title}" />
        <TextBlock Grid.Row="1" Text="{Binding XPath=dc:date}" />
      </Grid>
    </Border>
  </DataTemplate>
</StackPanel.Resources>

<ListBox
  Width="600"
  Height="600"
  Background="Honeydew"
  ItemsSource="{Binding Source={StaticResource provider}}"
  ItemTemplate="{StaticResource dataTemplate}"/>
```

Note that the Prefix you specify does not have to match the one used in the XML source; if the prefix changes in the XML source your mapping still works.

In this particular example, the XML data comes from a web service, but the XmlNamespaceMapping element also works with inline XML or XML data in an embedded file.

See also

- Bind to XML Data Using an XMLDataProvider and XPath Queries
- Data Binding Overview
- How-to Topics

How to: Bind to an ADO.NET Data Source

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to bind a Windows Presentation Foundation (WPF) ListBox control to an ADO.NET `DataSet`.

Example

In this example, an `OleDbConnection` object is used to connect to the data source which is an `Access MDB` file that is specified in the connection string. After the connection is established, an `OleDbDataAdpater` object is created. The `OleDbDataAdpater` object executes a select Structured Query Language (SQL) statement to retrieve the recordset from the database. The results from the SQL command are stored in a `DataTable` of the `DataSet` by calling the `Fill` method of the `OleDbDataAdapter`. The `DataTable` in this example is named `BookTable`. The example then sets the DataContext property of the ListBox to the `DataSet` object.

```
DataSet myDataSet;

private void OnInit(object sender, EventArgs e)
{
  string mdbFile = Path.Combine(AppDataPath, "BookData.mdb");
  string connString = string.Format(
      "Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}", mdbFile);
  OleDbConnection conn = new OleDbConnection(connString);
  OleDbDataAdapter adapter = new OleDbDataAdapter("SELECT * FROM BookTable;", conn);

  myDataSet = new DataSet();
  adapter.Fill(myDataSet, "BookTable");

  // myListBox is a ListBox control.
  // Set the DataContext of the ListBox to myDataSet
  myListBox.DataContext = myDataSet;
}
```

```
Private myDataSet As DataSet

Private Sub OnInit(ByVal sender As Object, ByVal e As EventArgs)
  Dim mdbFile As String = Path.Combine(AppDataPath, "BookData.mdb")
  Dim connString As String = String.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}", mdbFile)
  Dim conn As New OleDbConnection(connString)
  Dim adapter As New OleDbDataAdapter("SELECT * FROM BookTable;", conn)

  myDataSet = New DataSet()
  adapter.Fill(myDataSet, "BookTable")

  ' myListBox is a ListBox control.
  ' Set the DataContext of the ListBox to myDataSet
  myListBox.DataContext = myDataSet
End Sub
```

We can then bind the ItemsSource property of the ListBox to `BookTable` of the `DataSet`:

```
<ListBox Name="myListBox" Height="200"
  ItemsSource="{Binding Path=BookTable}"
  ItemTemplate  ="{StaticResource BookItemTemplate}"/>
```

`BookItemTemplate` is the DataTemplate that defines how the data appears:

```
<StackPanel.Resources>
  <c:IntColorConverter x:Key="MyConverter"/>

  <DataTemplate x:Key="BookItemTemplate">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250" />
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <TextBlock Text="{Binding Path=Title}" Grid.Column="0"
        FontWeight="Bold" />
      <TextBlock Text="{Binding Path=ISBN}" Grid.Column="1" />
      <TextBlock Grid.Column="2" Text="{Binding Path=NumPages}"
                 Background="{Binding Path=NumPages,
          Converter={StaticResource MyConverter}}"/>
    </Grid>
  </DataTemplate>
</StackPanel.Resources>
```

The `IntColorConverter` converts an `int` to a color. With the use of this converter, the Background color of the third TextBlock appears green if the value of `NumPages` is less than 350 and red otherwise. The implementation of the converter is not shown here.

See also

- BindingListCollectionView
- Data Binding Overview
- How-to Topics

How to: Bind to a Method

1/23/2019 • 2 minutes to read • Edit Online

The following example shows how to bind to a method using ObjectDataProvider.

Example

In this example, `TemperatureScale` is a class that has a method `ConvertTemp`, which takes two parameters (one of `double` and one of the `enum` type `TempType)` and converts the given value from one temperature scale to another. In the following example, an ObjectDataProvider is used to instantiate the `TemperatureScale` object. The `ConvertTemp` method is called with two specified parameters.

```
<Window.Resources>
  <ObjectDataProvider ObjectType="{x:Type local:TemperatureScale}"
                      MethodName="ConvertTemp" x:Key="convertTemp">
    <ObjectDataProvider.MethodParameters>
      <system:Double>0</system:Double>
      <local:TempType>Celsius</local:TempType>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>

  <local:DoubleToString x:Key="doubleToString" />

</Window.Resources>
```

Now that the method is available as a resource, you can bind to its results. In the following example, the Text property of the TextBox and the SelectedValue of the ComboBox are bound to the two parameters of the method. This allows users to specify the temperature to convert and the temperature scale to convert from. Note that BindsDirectlyToSource is set to `true` because we are binding to the MethodParameters property of the ObjectDataProvider instance and not properties of the object wrapped by the ObjectDataProvider (the `TemperatureScale` object).

The Content of the last Label updates when the user modifies the content of the TextBox or the selection of the ComboBox.

```xml
<Label Grid.Row="1" HorizontalAlignment="Right">Enter the degree to convert:</Label>
<TextBox Grid.Row="1" Grid.Column="1" Name="tb">
  <TextBox.Text>
    <Binding Source="{StaticResource convertTemp}" Path="MethodParameters[0]"
             BindsDirectlyToSource="true" UpdateSourceTrigger="PropertyChanged"
             Converter="{StaticResource doubleToString}">
      <Binding.ValidationRules>
        <local:InvalidCharacterRule/>
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
<ComboBox Grid.Row="1" Grid.Column="2"
  SelectedValue="{Binding Source={StaticResource convertTemp},
  Path=MethodParameters[1], BindsDirectlyToSource=true}">
  <local:TempType>Celsius</local:TempType>
  <local:TempType>Fahrenheit</local:TempType>
</ComboBox>
<Label Grid.Row="2" HorizontalAlignment="Right">Result:</Label>
<Label Content="{Binding Source={StaticResource convertTemp}}"
    Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"/>
```

The converter `DoubleToString` takes a double and turns it into a string in the Convert direction (from the binding source to binding target, which is the Text property) and converts a `string` to a `double` in the ConvertBack direction.

The `InvalidationCharacterRule` is a ValidationRule that checks for invalid characters. The default error template, which is a red border around the TextBox, appears to notify users when the input value is not a double value.

See also

- How-to Topics
- Bind to an Enumeration

How to: Set Up Notification of Binding Updates

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to set up to be notified when the binding target (target) or the binding source (source) property of a binding has been updated.

Example

Windows Presentation Foundation (WPF) raises a data update event each time that the binding source or target has been updated. Internally, this event is used to inform the user interface (UI) that it should update, because the bound data has changed. Note that for these events to work, and also for one-way or two-way binding to work properly, you need to implement your data class using the INotifyPropertyChanged interface. For more information, see Implement Property Change Notification.

Set the NotifyOnTargetUpdated or NotifyOnSourceUpdated property (or both) to `true` in the binding. The handler you provide to listen for this event must be attached directly to the element where you want to be informed of changes, or to the overall data context if you want to be aware that anything in the context has changed.

Here is an example that shows how to set up for notification when a target property has been updated.

```
<TextBlock Grid.Row="1" Grid.Column="1" Name="RentText"
  Text="{Binding Path=Rent, Mode=OneWay, NotifyOnTargetUpdated=True}"
  TargetUpdated="OnTargetUpdated"/>
```

You can then assign a handler based on the EventHandler<T> delegate, *OnTargetUpdated* in this example, to handle the event:

```
private void OnTargetUpdated(Object sender, DataTransferEventArgs args)
{

  // Handle event
```

```
}
```

Parameters of the event can be used to determine details about the property that changed (such as the type or the specific element if the same handler is attached to more than one element), which can be useful if there are multiple bound properties on a single element.

See also

- Data Binding Overview
- How-to Topics

How to: Clear Bindings

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to clear bindings from an object.

Example

To clear a binding from an individual property on an object, call ClearBinding as shown in the following example. The following example removes the binding from the TextProperty of *mytext*, a TextBlock object.

```
BindingOperations.ClearBinding(myText, TextBlock.TextProperty);
```

```
BindingOperations.ClearBinding(Me.myText, TextBlock.TextProperty)
```

Clearing the binding removes the binding so that the value of the dependency property is changed to whatever it would have been without the binding. This value could be a default value, an inherited value, or a value from a data template binding.

To clear bindings from all possible properties on an object, use ClearAllBindings.

See also

- BindingOperations
- Data Binding Overview
- How-to Topics

How to: Find DataTemplate-Generated Elements

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to find elements that are generated by a DataTemplate.

Example

In this example, there is a ListBox that is bound to some XML data:

```
<ListBox Name="myListBox" ItemTemplate="{StaticResource myDataTemplate}"
         IsSynchronizedWithCurrentItem="True">
  <ListBox.ItemsSource>
    <Binding Source="{StaticResource InventoryData}" XPath="Books/Book"/>
  </ListBox.ItemsSource>
</ListBox>
```

The ListBox uses the following DataTemplate:

```
<DataTemplate x:Key="myDataTemplate">
  <TextBlock Name="textBlock" FontSize="14" Foreground="Blue">
    <TextBlock.Text>
      <Binding XPath="Title"/>
    </TextBlock.Text>
  </TextBlock>
</DataTemplate>
```

If you want to retrieve the TextBlock element generated by the DataTemplate of a certain ListBoxItem, you need to get the ListBoxItem, find the ContentPresenter within that ListBoxItem, and then call FindName on the DataTemplate that is set on that ContentPresenter. The following example shows how to perform those steps. For demonstration purposes, this example creates a message box that shows the text content of the DataTemplate-generated text block.

```
// Getting the currently selected ListBoxItem
// Note that the ListBox must have
// IsSynchronizedWithCurrentItem set to True for this to work
ListBoxItem myListBoxItem =
    (ListBoxItem)(myListBox.ItemContainerGenerator.ContainerFromItem(myListBox.Items.CurrentItem));

// Getting the ContentPresenter of myListBoxItem
ContentPresenter myContentPresenter = FindVisualChild<ContentPresenter>(myListBoxItem);

// Finding textBlock from the DataTemplate that is set on that ContentPresenter
DataTemplate myDataTemplate = myContentPresenter.ContentTemplate;
TextBlock myTextBlock = (TextBlock)myDataTemplate.FindName("textBlock", myContentPresenter);

// Do something to the DataTemplate-generated TextBlock
MessageBox.Show("The text of the TextBlock of the selected list item: "
    + myTextBlock.Text);
```

```
' Getting the currently selected ListBoxItem
' Note that the ListBox must have
' IsSynchronizedWithCurrentItem set to True for this to work
Dim myListBoxItem As ListBoxItem =
CType(myListBox.ItemContainerGenerator.ContainerFromItem(myListBox.Items.CurrentItem), ListBoxItem)

' Getting the ContentPresenter of myListBoxItem
Dim myContentPresenter As ContentPresenter = FindVisualChild(Of ContentPresenter)(myListBoxItem)

' Finding textBlock from the DataTemplate that is set on that ContentPresenter
Dim myDataTemplate As DataTemplate = myContentPresenter.ContentTemplate
Dim myTextBlock As TextBlock = CType(myDataTemplate.FindName("textBlock", myContentPresenter), TextBlock)

' Do something to the DataTemplate-generated TextBlock
MessageBox.Show("The text of the TextBlock of the selected list item: " & myTextBlock.Text)
```

The following is the implementation of `FindVisualChild`, which uses the VisualTreeHelper methods:

```
private childItem FindVisualChild<childItem>(DependencyObject obj)
    where childItem : DependencyObject
{
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
    {
        DependencyObject child = VisualTreeHelper.GetChild(obj, i);
        if (child != null && child is childItem)
            return (childItem)child;
        else
        {
            childItem childOfChild = FindVisualChild<childItem>(child);
            if (childOfChild != null)
                return childOfChild;
        }
    }
    return null;
}
```

```
Private Function FindVisualChild(Of childItem As DependencyObject)(ByVal obj As DependencyObject) As childItem
    For i As Integer = 0 To VisualTreeHelper.GetChildrenCount(obj) - 1
        Dim child As DependencyObject = VisualTreeHelper.GetChild(obj, i)
        If child IsNot Nothing AndAlso TypeOf child Is childItem Then
            Return CType(child, childItem)
        Else
            Dim childOfChild As childItem = FindVisualChild(Of childItem)(child)
            If childOfChild IsNot Nothing Then
                Return childOfChild
            End If
        End If
    Next i
    Return Nothing
End Function
```

See also

- How to: Find ControlTemplate-Generated Elements
- Data Binding Overview
- How-to Topics
- Styling and Templating
- WPF XAML Namescopes
- Trees in WPF

How to: Bind to a Web Service

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to bind to objects returned by Web service method calls.

Example

This example uses the MSDN/TechNet Publishing System (MTPS) Content Service to retrieve the list of languages supported by a specified document.

Before you call a Web service, you need to create a reference to it. To create a Web reference to the MTPS service using Microsoft Visual Studio, follow the following steps:

1. Open your project in Visual Studio.
2. From the **Project** menu, click **Add Web Reference**.
3. In the dialog box, set the **URL** to http://services.msdn.microsoft.com/contentservices/contentservice.asmx?wsdl.
4. Press **Go** and then **Add Reference**.

Next, you call the Web service method and set the DataContext of the appropriate control or window to the returned object. The **GetContent** method of the MTPS service takes a reference to the **getContentRequest** object. Therefore, the following example first sets up a request object:

```
// 1. Include the web service namespace
using BindtoContentService.com.microsoft.msdn.services;
```

```
' 1. Include the web service namespace
Imports BindtoContentService.com.microsoft.msdn.services
```

```
// 2. Set up the request object
// To use the MSTP web service, we need to configure and send a request
// In this example, we create a simple request that has the ID of the XmlReader.Read method page
getContentRequest request = new getContentRequest();
request.contentIdentifier = "abhtw0f1";

// 3. Create the proxy
ContentService proxy = new ContentService();

// 4. Call the web service method and set the DataContext of the Window
// (GetContent returns an object of type getContentResponse)
this.DataContext = proxy.GetContent(request);
```

```
' 2. Set up the request object
' To use the MSTP web service, we need to configure and send a request
' In this example, we create a simple request that has the ID of the XmlReader.Read method page
Dim request As New getContentRequest()
request.contentIdentifier = "abhtw0f1"

' 3. Create the proxy
Dim proxy As New ContentService()

' 4. Call the web service method and set the DataContext of the Window
' (GetContent returns an object of type getContentResponse)
Me.DataContext = proxy.GetContent(request)
```

After the DataContext has been set, you can create bindings to the properties of the object that the DataContext has been set to. In this example, the DataContext is set to the **getContentResponse** object returned by the **GetContent** method. In the following example, the ItemsControl binds to and displays the **locale** values of **availableVersionsAndLocales** of **getContentResponse**.

```
<ItemsControl Grid.Column="1" Grid.Row="2" Margin="0,3,0,0"
              ItemsSource="{Binding Path=availableVersionsAndLocales}"
              DisplayMemberPath="locale"/>
```

For information about the structure of **getContentResponse**, see Content Service documentation.

See also

- Data Binding Overview
- Binding Sources Overview
- Make Data Available for Binding in XAML

How to: Bind to the Results of a LINQ Query

1/23/2019 • 2 minutes to read • Edit Online

This example demonstrates how to run a LINQ query and then bind to the results.

Example

The following example creates two list boxes. The first list box contains three list items.

```
<ListBox SelectionChanged="ListBox_SelectionChanged"
         SelectedIndex="0" Margin="10,0,10,0" >
    <ListBoxItem>1</ListBoxItem>
    <ListBoxItem>2</ListBoxItem>
    <ListBoxItem>3</ListBoxItem>
</ListBox>
<ListBox Width="400" Margin="10" Name="myListBox"
         HorizontalContentAlignment="Stretch"
         ItemsSource="{Binding}"
         ItemTemplate="{StaticResource myTaskTemplate}"/>
```

Selecting an item from the first list box invokes the following event handler. In this example, `Tasks` is a collection of `Task` objects. The `Task` class has a property named `Priority`. This event handler runs a LINQ query that returns the collection of `Task` objects that have the selected priority value, and then sets that as the DataContext:

```
using System.Linq;
```

```
Tasks tasks = new Tasks();
```

```
private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    int pri = Int32.Parse(((sender as ListBox).SelectedItem as ListBoxItem).Content.ToString());

    this.DataContext = from task in tasks
                       where task.Priority == pri
                       select task;
}
```

The second list box binds to that collection because its ItemsSource value is set to `{Binding}`. As a result, it displays the returned collection (based on the `myTaskTemplate` DataTemplate).

See also

- Make Data Available for Binding in XAML
- Bind to a Collection and Display Information Based on Selection
- What's New in WPF Version 4.5
- Data Binding Overview
- How-to Topics