<http://msdn.microsoft.com/en-us/library/aa480192.aspx#Mtps_DropDownFilterText>

WPF, Application = Code + Markup

|  |
| --- |
| Chapter 3. The Concept of Content You can set the Content property to a string, you can set it to a bitmap, you can set it to a drawing, and you can set it to a button, or a scrollbar, or any one of 50-odd controls supported by the Windows Presentation Foundation. You can set the Content property to just about anything. But there's only one little problem:  You can only set the Content property to one object.    There is no such thing as a Font class in the WPF    Like every measurement in the WPF, you specify the FontSize in device-independent units, which are 1/96 inch |

StackPanel and WrapPanel

DockPanel and GridPanel

Canvas

In Visual Tree, some properties has priorities. It is possible to formulate some generalizations about the FontSize property. There is a default value of FontSize, but that default is low in priority. A value inherited from an ancestor in an element tree is higher in priority than the default, and a value explicitly set on the object has the highest priority.

Even though the DependencyProperty object passed to the SetValue and GetValue methods is static, SetValue and GetValue are instance methods, and they are setting and obtaining values associated with the particular instance. The DependencyObject is maintaining the current value, and handling all the routine stuff as well. For example, if SetValue hasn't yet been called for the particular Control instance, GetValue returns the value of the DefaultValue property associated with the metadata of FontSizeProperty

An interesting topic about DependencyProperty.

DockProperty is defined as a DependencyProperty but it is registered with a DependencyProperty.RegisterAttached method, so it's called an attached property. If it were a normal dependency property, DockProperty would be associated with a property named Dock with calls to SetValue and GetValue. There is no property named Dock. Instead, DockProperty is referred to in two static methods defined by DockPanel named SetDock and GetDock.

Event routing in the WPF allows both scenarios. There is always an element that is considered the "source" of an event. For mouse and stylus events, the source of the event is generally the element most in the foreground underneath the mouse pointer or stylus. This element must be both visible and enabled. (That is, the element's Visibility property must equal the enumeration member Visibility.Visible, and the IsEnabled property must be true.) If the element is not visible and enabled, the source of the event is the topmost underlying element that is enabled and visible.

The source of a keyboard event is the element that has the input focus. For both keyboard and mouse events, only one element is the source of any particular event, but all ancestors of the source element in the element tree also potentially have access to that event.

The Click event is defined by ButtonBase and inherited by Button. The Window class knows nothing about Click events and neither does Grid. It would normally be impossible to install a Click event handler on a Window or Grid object. But UIElement defines an AddHandler method (and a corresponding RemoveHandler method) that accepts a first argument of type RoutedEvent and lets you install an event handler for any routed event of any other element in the same tree. Here's how the ExamineRoutedEvents program uses AddHandler to install a Click event handler on the other elements:

el.AddHandler(Button.ClickEvent,

new RoutedEventHandler(AllPurposeEventHandler));

Installing a handler on a panel (for example) is a very useful technique for consolidating the handling of particular events coming from a number of child elements.

Defining a routed event in your own class is similar to defining a dependency property. Suppose you have a control in which you need a Click-like event that you'd rather call Knock. You first define a static read-only field of type RoutedEvent for Knock (and optionally PreviewKnock):

public static readonly RoutedEvent KnockEvent;

public static readonly RoutedEvent PreviewKnockEvent;

By convention, the field names consist of the name of the event followed by the word Event.

In the field definition itself or in a static constructor, you call the static EventManager.RegisterRoutedEvent method. The first argument is the text name of the event:

KnockEvent =

EventManager.RegisterRoutedEvent("Knock", RoutingStrategy.Bubble,

typeof(RoutedEventHandler), typeof(YourClass));

PreviewKnockEvent =

EventManager.RegisterRoutedEvent("PreviewKnock", RoutingStrategy.Tunnel,

typeof(RoutedEventHandler), typeof(YourClass));

Although I've been discussing user input events as implemented in the UIElement class, these events are implemented in ContentElement as well. This is the class from which TextElement, FixedDocument, and FlowDocument derive. The ContentElement objects are elements that cannot render themselves on the screen, but are rendered by other elements. Both UIElement and ContentElement implement the IInputElement interface, which also includes most (but not all) of the user input events defined by UIElement and ContentElement

When creating a custom element, you'll almost certainly be inheriting from FrameworkElement, just like Image, Panel, TextBlock, and Shape do. (You could alternatively inherit from UIElement, but the process is somewhat different than what I'll be describing.) When creating a custom control, you'll probably inherit from Control or (if you're lucky) from one of the classes that derive from Control such as ContentControl.

Of course, what a program draws in its OnRender method does not go directly to the screen. The graphical object defined in OnRender is retained by the WPF graphics system and displayed along with other visual objects in a composition. The graphical object is retained until a subsequent call to OnRender replaces it. Calls to OnRender can occur any time the system detects a need to update the visual rendition of the element. This generally happens much less often than WM\_PAINT messages in Win32 or Paint events in Windows Forms programs, however, because the graphics are retained. OnRender needn't be called when a visual object is exposed while moving another window, for example. But OnRender is called if the element size changes. An explicit call to the InvalidateVisual method (defined by UIElement) can also force a call to OnRender.

WPF布局引擎采用了一种递归的方式来实现控件及其子控件的布局,大致过程是这样的:要实现控件A的布局,那么先要实现A的子控件a1,a2,a3...的布局,要实现a1的布局,那么得实现a1的子控件a11,a12,a13...的布局,如此循环,然后但子控件的布局完成后,在完成父控件的布局,最后递归回去,递归结束,布局才算完成.

控件的最终大小和位置是由该控件和父控件相磋商来完成的,父控件先给出其能给予子控件的力所能及的布局空间,子控件在**MeasureOverride**反馈给父控件一个自己的期望值,父控件最后根据自己所拥有的空间大小与子控件的期望值ArrangeOverride分配一定的空间给子控件并返回自己的大小.这一系列过程是通过重写面板的MeasureOverride和ArrangeOverride方法来完成的

layout in the Windows Presentation Foundation is a two-pass process that begins with a root element and works downward through the element tree. The root element is an object of type Window. This object has a single visual child of type Border. (That's the sizing border around the window.) The border has a visual child of type Grid. The grid has two children: an AdornerDecorator and a ResizeGrip (the latter of which only optionally appears in the window). The AdornerDecorator has one visual child of type ContentPresenter, which is responsible for hosting the Content property of the window. In the preceding program, the single visual child of the ContentPresenter is an EllipseWithChild, and its single visual child is a TextBlock. (I obtained information about the visual tree of this program through static methods of the VisualTreeHelper class.)

After calling Arrange on its children, ArrangeOverride generally returns its sizeFinal parameter. (The base implementation of ArrangeOverride does precisely that.) However, it could return something different. Whatever ArrangeOverride returns becomes RenderSize

In general, controls and elements are built up of other elements. Often, a control begins with some kind of Decorator object. Decorator inherits from FrameworkElement and defines a Child property of type UIElement. For example, the Border class inherits from Decorator to define properties Background, BorderBrush, BorderThickness, CornerRadius (for rounded corners), and Padding. The ButtonChrome class (in the Microsoft.Windows.Themes namespace) also inherits from Decorator and provides the look of the standard Button. The Button is basically a ButtonChrome object and a ContentPresenter object that is a child of the ButtonChrome object.

<http://www.cnblogs.com/jax/articles/1133487.html>

The key property of Control that allows this approach is Template, which is of type ControlTemplate. This Template property essentially defines the look and feel of the control. As you know by now, a normal Button is basically a ButtonChrome object and a ContentPresenter object. The ButtonChrome object gives the button the look of its background and border, while the ContentPresenter is responsible for hosting whatever you've set to the button's Content property. The template defines the links between these elements as well as "triggers" that cause the control to react to certain changes in element properties.

The following class hierarchy shows all the descendants of Visual except for the descendents of FrameworkElement:

Object

    DispatcherObject (abstract)

          DependencyObject

                Visual (abstract)

                      ContainerVisual

                            DrawingVisual

                            HostVisual

                            Viewport3DVisual

                            UIElement

                                  FrameworkElement

You are now a proud owner of a DrawingVisual objectin other words, a "visual" that stores a particular image. The parameters to the drawing functions of the DrawingContext that you called gave this visual a specific location and size. That location is relative to some parent element that may not yet exist. To display this visual on the screen, a particular element must indicate the existence of the child visual with the return value of VisualChildrenCount and GetVisualChild. That's it. You can't call Measure or Arrange on this visual because those methods are defined by UIElement. The visual is displayed relative to its parent element and visibly on top of anything the element draws during OnRender. The order with respect to other children of the element depends on the order established by GetVisualChild: Later visuals will appear in the foreground of earlier visuals

how to inherit from Panel and how to support multiple children without inheriting from Panel, and you'll probably understand why inheriting from Panel is easier. The big gift of Panel is the definition of the Children property for storing the children. This property is of type UIElementCollection, and that collection itself handles the calling of AddVisualChild, AddLogicalChild, RemoveVisualChild, and RemoveLogicalChild when children are added to or removed from the collection. UIElementCollection is able to perform this feat because it has knowledge of the parent element. The sole constructor of UIElementCollection requires two arguments: a visual parent of type UIElement and a logical parent of type FrameworkElement. The two arguments can be identical, and usually are.

Some code for binding:

         // Set the items and the property paths.

            lstbox.ItemsSource = NamedBrush.All;

            lstbox.DisplayMemberPath = "Name";

            lstbox.SelectedValuePath = "Brush";

            // Bind the SelectedValue to window

Background.

            lstbox.SetBinding(ListBox

.SelectedValueProperty, "Background");

            lstbox.DataContext = this;

These four classes that derive from Control encompass many familiar controls:

* Controls that derive from ContentControl are characterized by a property named Content. These controls include buttons, labels, tool tips, the scroll viewer, list box items, and the window itself.
* The HeaderedContentControl derives from ContentControl and adds a Header property. The group box falls under this category.
* ItemsControl defines a property named Items that is a collection of other objects. This category includes the list box and combo box.
* HeaderedItemsControls adds a Header property to the properties it inherits from ItemsControl. A menu item is one such control.

By this time you may have concluded that command bindings probably provide a simpler approach, and they certainly do. The CommandTheButton program in Chapter 4 showed how to use command bindings with a button. Using them with menu items is quite similar. Generally you'll be using static properties of type RoutedUICommand from the ApplicationCommands class and (for more esoteric applications) from the ComponentCommands, EditingCommands, MediaCommands, and NavigationCommands classes, but you can also make your own, as I'll demonstrate.

To use one of the predefined static properties, you set the Command property of the MenuItem like this:

itemCut.Command = ApplicationCommands.Cut;

If you don't set the Header property of the MenuItem, it will use the Text property of the RoutedUICommand, which is almost OK except there's no preceding underline. Regardless, the MenuItem automatically adds the "Ctrl+X" text to the menu item.

The other crucial step is creating a command binding based on the RoutedUICommand object, and adding it to the CommandBindings collection of the window:

CommandBindings.Add(new CommandBinding(ApplicationCommands.Cut,

CutOnExecute, CutCanExecute));

Control

       ItemsControl

                 HeaderedItemsControl

                             MenuItem

                             ToolBar

                             TreeViewItem

                 MenuBase (abstract)

                            ContextMenu

                            Menu

                 StatusBar

                 TreeView

The PrintDialog class displays a dialog box, of course, but the class also includes methods to print a single page or to print a multi-page document. In both cases, what you print on the page is an object of type Visual. As you know by now, one important class that inherits from Visual is UIElement, which means that you can print an instance of any class that derives from FrameworkElement, including panels, controls, and other elements. For example, you could create a Canvas or other panel; put a bunch of child controls, elements, or shapes on it; and then print it.

When a program prints an instance of a class derived from UIElement, a crucial step is required: You must subject the element to layout, which means you must call Measure and Arrange on the object. Otherwise, the object will have a zero dimension and won't show up on the page.

You could print a multi-page document by making multiple calls to PrintVisual, but each page would be considered a different print job. To better print a multi-page document, a program calls the PrintDocument method defined by PrintDialog. The arguments to PrintDocument are an instance of a class derived from DocumentPaginator and a text string for the print queue describing the document.

A XAML file can often replace an entire constructor of a class that derives from Window, which is the part of the class that generally performs layout and attaches event handlers. The event handlers themselves must be written in procedural code such as C#. However, if you can replace an event handler with a data binding, that binding can usually go right into the XAML.

The use of XAML separates the visual appearance of an application from its functionality. This separation allows designers to work with XAML files to create an attractive user interface, while the programmers focus more on the run-time interactions among the elements and controls. Design tools that generate XAML are already becoming available.

The root element of a stand-alone XAML file can be anything that derives from FrameworkElement except for Window.

 The System.Windows.Markup namespace contains a class named XamlReader with a static method named Load that can parse XAML and turn it into an initialized object.

It's the same as the URL for WPF except without the additional path of presentation, which refers to the Windows Presentation Foundation. The WPF namespace declaration will continue to appear in all the XAML files in this book:

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

The namespace for XAML-specific elements and attributes is customarily declared with a prefix of x:

xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>, like x:Code, x:Class

Suppose you have a custom control named MyControl in a C# file with a CLR namespace of MyNamespace.

You include this C# file as part of the project. In the XAML file,

you must associate this CLR namespace with a prefixfor example, stuff,

using a namespace declaration like this: xmlns:stuff="clr-namespace:MyNamespace"

Generating menu command:

            // Command to reparse.

            InputGestureCollection collGest = new InputGestureCollection();

            collGest.Add(new KeyGesture(Key.F6));

            RoutedUICommand commReparse =

                new RoutedUICommand("\_Reparse", "Reparse",

                                    GetType(), collGest);

            // Menu item to reparse.

            MenuItem itemReparse = new MenuItem();

            itemReparse.Command = commReparse;

            itemXaml.Items.Add(itemReparse);

            // Command binding to reparse.

            CommandBindings.Add(new CommandBinding(commReparse,

                                ReparseOnExecuted));

T

PresentationFramework assembly contains a number of custom attributes. (An application can interrogate these attributes by calling GetCustomAttributes on the Assembly object.) Several of these attributes are of type XmlnsDefinitionAttribute, and this class contains two properties of importance named XmlNamespace and ClrNamespace. One of the XmlnsDefinitionAttribute objects in PresentationFramework has its XmlNamespace set to the string "<http://schemas.microsoft.com/winfx/2006/xaml/presentation>" and its ClrNamespace property set to the string "System.Windows.Controls." The syntax looks like this:

[assembly:XmlnsDefinition

("http://schemas.microsoft.com/winfx/2006/xaml/presentation",

"System.Windows.Controls")]

Other XmlnsDefinition attributes associate this same XML namespace with the CLR namespaces System.Windows, System.Windows.Controls.Primitives, System.Windows.Input, System.Windows.Shapes, and so forth.

The XAML parser examines the XmlnsDefinition attributes (if any) in all the assemblies loaded by the application. If any of the XML namespaces in these attributes match the XML namespace in the XAML file, the parser knows which CLR namespaces to assume when searching for a Button class in these assemblies.

 However, the System namespace is not among the CLR namespaces associated with the two XML namespaces at the top of the XAML file. To refer to the Byte structure in a XAML file, you need another XML namespace declaration. Let's associate the System namespace with a prefix of s:

xmlns:s="clr-namespace:System;assembly=mscorlib"

<Polygon.Fill>

<SolidColorBrush>

<SolidColorBrush.Color>

<Color xmlns:s="clr-namespace:System;assembly=mscorlib"

A="255" G="0" B="0">

<Color.R>

<s:Byte>

#FF

</s:Byte>

</Color.R>

</Color>

</SolidColorBrush.Color>

</SolidColorBrush>

</Polygon.Fill>

I think you'll agree that in many cases XAML is more concise than equivalent C# code,

and it better represents hierarchical structures such as those that arise in laying out

a window with panels and controls. But markup in general is much more limited than procedural

languages, mostly because there's no concept of flow control. Even the simple sharing

of variables seems unlikely in XAML. As you'll begin to see in the next chapter,

however, XAML has a number of features that compensate for these deficiencies.

You can reuse objects in XAML by first defining them as resources.

Resources are stored in an object of type ResourceDictionary, and three very fundamental classesFrameworkElement, FrameworkContentElement, and Applicationall define a prope]rty named Resources of type ResourceDictionary

To prevent the XAML parser from searching for (and failing to find) a markup extension named just, insert an escape sequence in front consisting of a pair of empty curly brackets:

<!-- Works just fine! -->

<TextBlock Text="{}{just a little text in here}" />

For example, suppose you want to set the Content property of a Button to a static property named SomeStaticProp from a class named SomeClass. The markup extension syntax is:

Content="{x:Static SomeClass:SomeStaticProp}"

StaticResource and DynamicResource represent two different approaches to accessing resources. Both require keys and use those keys to access objects. With StaticResource, the key is used to access the object once, and the object is retained. When you use DynamicResource, the key is retained and the object is accessed when it's needed.

DynamicResource is expecting the object referenced by the key to be recreated. The brush objects are not being recreated, so DynamicResource doesn't bother with updating the Foreground and Background properties.

Resources that you want to share among multiple projects can be collected in XAML files with a root element of ResourceDictionary. Each resource is a child of that root element.

|  |
| --- |
| UseCommonResourcesApp.xaml |
| [View full width]  <!-- ===  =====================================================  UseCommonResourcesApp.xaml (c) 2006 by  Charles Petzold  =============================================  =========== -->  <Application xmlns="http://schemas.microsoft.com/  winfx/2006/xaml/presentation"  StartupUri="UseCommonResourcesWindow  .xaml">  <Application.Resources>  <ResourceDictionary>  <ResourceDictionary.MergedDictionaries>  <ResourceDictionary  Source="MyResources1.xaml" />  <ResourceDictionary  Source="MyResources2.xaml" />  </ResourceDictionary.MergedDictionaries>  </ResourceDictionary>  </Application.Resources>  </Application> |

This chapter also discusses the three file formats you can use for distributing a WPF application. The first, of course, is the traditional .exe format, and you've already seen numerous WPF applications that result in .exe files. At the other extreme is the stand-alone XAML file that can be developed in XAML Cruncher (or a similar program) and hosted in Microsoft Internet Explorer.

Between these two extremes is the XAML Browser Application, which has a file name extension of .xbap. As the name implies, these XAML Browser Applications are hosted in Internet Explorer just like stand-alone XAML files. Yet they generally consist of both XAML and C# code, and they are compiled. Because they're intended to run in the context of Internet Explorer, security restrictions limit what these applications can do. In short, they can't do anything that could harm the user's computer. Consequently, they can be run on a user's computer without asking for specific permission or causing undue anxiety.

Source is one of several properties that NavigationWindow defines beyond the properties it inherits from Window. You'll recall that Frame also has a Source property. Both NavigationWindow and Frame derive from ContentControl, but if you set the Content of either element, that Content property will take precedence over the Source property. Generally you'll be using NavigationWindow or Frame to take advantage of their navigational abilities, so you'll want to focus on the Source property rather than on Content.

Welcome to the world of WPF navigation applications. As you've discovered, the Hyperlink element normally doesn't work unless you install an event handler for it. However, if the Hyperlink is inside a NavigationWindow (or a Frame element), clicking the link automatically navigates to the page specified in the NavigateUri property. The NavigateUri property can reference a URI of a Web site, but it's more commonly the name of another XAML file in the program.

Taken to the extreme, the use of NavigationWindow and Frame lets you structure your entire WPF application much like a Web site, but without giving up any of the power of WPF elements, controls, and graphics.

A Page element can be a child only of a NavigationWindow or a Frame.

The two Hyperlink elements contain not only NavigateUri attributes that reference two XAML files, but also TargetName attributes that reference the Frame at the left of the BookList.xaml page. This is how a Hyperlink can deposit a XAML page in another Frame.

Because XAML Browser Applications are hosted in the Web browser, there are certain limitations to what they can do. These programs cannot create objects of type Window or NavigationWindow. They are essentially organized around Page objects. If necessary, they move from page to page through navigation. XAML Browser Applications can have menus, but they cannot create dialog boxes or objects of type Popup. These programs run with security permissions associated with the Internet zone. They cannot use the file system.

Data bindings defined in XAML can eliminate the need for an event handler in the code-behind file and, in some cases, eliminate the code-behind file entirely. The result is code that I like to think of as having "no moving parts." Everything is initialization, and much less can go wrong. (Of course, the event handlers still exist, but they're behind the scenes, and presumably they come to us already debugged and robust enough for heavy lifting.)

Regardless of how you do it, the control or element in which the Binding definition occurs is always the target of the binding. A binding target must derive from DependencyObject.The property on which the binding is set must be backed by a dependency property.

The bound property of the source need not be a dependency property. In the ideal case, the property should be associated with an event that indicates when the property changes, but some bindings can work even without that notification event.

When binding, in some cases the DataContext property is extremely valuable. DataContext is inherited through the element tree, so if you set it for one element, it also applies to all the children of that element.

The target of a data binding must be a dependency property.One of the primary incentives behind the invention of dependency properties was data binding, and the dependency property system has built-in notification support. The binding source doesn't have to be a dependency property, but it really helps if it is.

The FrameworkPropertyMetadataOptions.BindsTwoWayByDefault flag affects only bindings defined where that dependency property is the target.

As data is transferred from a binding source to a target (and sometimes back) the data might need to be converted from one type to another. The Binding class includes a property named Converter that lets you specify a class that includes two methods named Convert and ConvertBack to perform this conversion.

MultiBinding element always contains one or more Binding elements as children. The converter defined in the Resources section of the file and referenced by the key name of "convRgbToColor" must take the three values from the three bindings and convert them into an object appropriate for the Background property of the Border control

You can change this behavior by setting the UpdateSourceTrigger property of the Binding. You can set it to a member of the UpdateSourceTrigger enumeration, either LostFocus (the default for the Text property of a TextBox), PropertyChanged (which is normal for most properties), or Explicit, which requires special action by the program for the changes to be reflected in the source.

It is not necessary to inherit from FrameworkElement to define dependency properties. If the source of your data is not a visual object, you can instead inherit from DependencyObject. That's the class that defines the SetValue and GetValue methods you need to implement dependency properties

So far you've seen Binding markup using ElementName and Source. The only other alternative is RelativeSource, which lets you reference an ancestor element in the element tree, or the element itself. This RelativeSourceDemo.xaml file shows three bindings using RelativeSource.

Although the Resources sections of XAML files are useful for defining miscellaneous objects that you refer to in markup, many resource sections are used primarily for the definition of Style objects. Styles are essentially collections of property values that are applied to elements. Styles are partially the compensation for not being able to use loops in XAML to create multiple elements with identical properties

the WPF implementation of styles is more powerful because changes in properties can also be specified, which are triggered by changes in other properties, or by events.

Setter is the content property of Style, so Setter and EventSetter elements are children of the Style element:

<Style ...>

    <Setter ... />

    <EventSetter ... />

    <Setter ... />

</Style>

 local settingswhich is the term used for properties set directly on the elementtake precedence over Style settings. But styles take precedence over properties inherited through the visual tree.

The differences between resources and styles are:

1. Resources can only be referenced

2. Resources only works on one property of one control.

3. Resources can be contained in the style

4. Style may be triggered.

5. Style work on class.

6. Style is a kind of resource.

For any particular element, only one Style applies. This is the Style encountered first when searching up the visual tree where the key matches, or the class of the element matches the TargetType, or both, if the Style has both

Any object referred to in a particular Style is created only once to be part of that Style, and hence is shared among all elements that use that Style. An element like Imageand indeed, any element derived from Visual or ContentElementcan have only one parent, and that would not be the case if the Style object were shared. Again, consider a custom class or a template if you run into this problem.

The Value attribute of a Setter element can be a data binding

<Setter Property="Y2"

        Value="{Binding RelativeSource={RelativeSource self}, Path=Y1}" />

You then need only specify the value Y1 in the individual Line elements, and that value is also used for Y2.

Although Setter is the most common child of Style, you can also use the EventSetter element to set an event handler of a particular routed event. This is yet another way in which you can share an event handler among multiple elements.

       <Style TargetType="{x:Type Button}">

            <EventSetter Event="Click"

Handler="ButtonOnClick" />

        </Style>

The last property, named Triggers, lets you control how elements or controls respond to changes in properties of the element, or to changes in a data binding, or to events.

 Setter effectively sets properties when the element is first created. A Trigger sets properties only when something happensthat is, when something "triggers" the property to change

<Style.Triggers>

    <Trigger Property="Control.IsMouseOver" Value="True">

        <Setter Property="Control.FontStyle" Value="Italic" />

        <Setter Property="Control.Foreground" Value="Blue" />

    </Trigger>

</Style.Triggers>

The MultiTrigger is similar to the Trigger except that it only kicks in when two or more conditions hold

The DataTrigger class is similar to Trigger except that it substitutes Binding for Property

The MultiDataTrigger is to DataTrigger what MultiTrigger is to Trigger

As you've seen, styles give you the convenience of organizing the appearance of your controls in a very systematic way. Styles can be as simple as a few lines in a XAML file to reduce repetition among similar elements, or they can fill entire files and provide every control in your applications with a unique look. Whenever you're about to give two elements the same property value, use a style instead, and you'll be thankful if you ever need to change that value.

The ControlTemplate is one important type of template supported by the Windows Presentation Foundation. As its name suggests, you use the ControlTemplate to define the visual appearance of a control. The Control class defines the Template property that you set to a ControlTemplate object.

Although styles and templates may seem to overlap, they really have quite different roles. An element or control does not have a default Style property, and consequently, the Style property of an element is normally null. You use the Style property to define property settings or triggers that you want associated with that element.

All controls defined within the Windows Presentation Foundation that have a visual appearance already have a Template property that is set to an object of type ControlTemplate. A Button looks like a Button and a ScrollBar looks like a ScrollBar as a direct result of these ControlTemplate objects. The ControlTemplate object defines the entire visual appearance of the control, and you have the power to replace that object. This is what is meant when the controls in the WPF are referred to (rather awkwardly) as "lookless" controls. They certainly have a "look" but it's not intrinsic to the functionality of the control and it can be replaced.

Template may contain other controls and control the total visual appearance of the template container.

Object

       FrameworkTemplate (abstract)

                 ControlTemplate

                 DataTemplate

                         HierarchicalDataTemplate

                 ItemsPanelTemplate

Obviously some practice is required before these different types of templates (and the properties you set them to) assume individual and familiar personalities. Here's a brief summary:

You create objects of type ControlTemplate to set to the Template property defined by Control. These ControlTemplate objects define the entire visual appearance of the control, including triggered changes to the visual appearance. Setting the Template property is quite powerful but also (of course) involves much responsibility.

You use objects of type ItemsPanelTemplate to set to the ItemsPanel property defined by ItemsControl to specify the type of panel used to display the multiple items in an ItemsControl (such as a ListBox or ComboBox). This is the simplest type of template.

All other templates are of type DataTemplate, and in actual practice are certainly the most common form of template. DataTemplate objects show up in all controls that derive from ContentControl and ItemsControl and let you govern how content and listed items are displayed.

The abstract FrameworkTemplate class defines only three properties. The read-only Boolean IsSealed property indicates whether the template can be altered. The Resources property of type ResourceDictionary lets you define resources that are accessible only within the template. The most important of the three properties is VisualTree, which defines the layout of elements that make up the visual appearance of the control (or the content of the control, or the listed items in the control).

When you define templates in XAML, FrameworkElementFactory works entirely behind the scenes. The XAML syntax used in defining templates is the same as that which you use when defining a layout of elements and their properties, except that these elements aren't actually created until they're required**.**

I want the Text property of this TextBlock to be set to the Content property of the Button. The markup extension TemplateBinding is defined specifically for the purpose of linking a property in the template's visual tree with a property defined by the control. The syntax for binding the Text property of the TextBlock to the Content property of the Button is:

-Text="{TemplateBinding ContentControl.Content}"

You can combine a Style and a ControlTemplate in the same resource. The Style begins normally and then the Template property is set to an object of type ControlTemplate:

<Style ... >

    ...

    <Setter Property="Template">

        <Setter.Value>

            <ControlTemplate ...>

                ...

            </ControlTemplate>

        </Setter.Value>

    </Setter>

</Style>

That way, you can give a control both a style and a template just by setting the Style property.

It so happens that all classes that derive from ContentControl use an object of type ContentPresenter to display their content. ContentPresenter derives from FrameworkElement, and you can include a ContentPresenter object within the visual tree of the template.

The ContentPresenter element is responsible for displaying content in all controls that derive from ContentControl. ContentPresenter divides the world into two types of objects: those that inherit from UIElement and those that don't. For those that don't, ContentPresenter uses the object's ToString method to display a text representation of the object.

Certainly the DataType attribute provides some helpful information for people who must examine the XAML file. But it actually works in reverse from what you might expect. It's actually used for locating a resource that matches the data type of a Content property.

That's one way to have different types of presentation of different types of content. A more flexible approach is provided by the ContentTemplateSelector property defined by ContentControl. This property is of type DataTemplateSelector, a class that contains a virtual method named SelectTemplate. You derive a class from DataTemplateSelector and override SelectTemplate

Keep in mind that you can use the ContentTemplate property to define a custom layout for any control that derives from ContentControl, which is certainly a significant proportion of WPF controls. After ContentControl, the ItemsControl class is the basis of the next most significant subset of controls. ItemsControl is the root of all classes that display multiple objects, including MenuItem, TreeViewItem, ListBox, ComboBox, and ListView.

You can access an XML file in the Resources section of a XAML file by defining an object of type XmlDataProvider with a Source attribute referencing the URI of the file:

<XmlDataProvider x:Key="emps" Source="Employees.xml" XPath="Employees" />

It is very advantageous that such a class implement the INotifyPropertyChanged interface. Strictly speaking, the INotifyPropertyChanged interface requires only that the class have an event named PropertyChanged defined in accordance with the PropertyChangedEventHandler delegate. But such an event is worthless unless the properties of the class fire the event whenever the values of the properties change.

Implementing the INotifyPropertyChanged interface lets the properties of this class participate as sources in data bindings. They cannot be targets because they are not backed by dependency properties, but usually it's enough that they can be sources.

Object

    DispatcherObject (abstract)

        DependencyObject

             Visual (abstract)

                  UIElement

                       FrameworkElement

                            Shape (abstract)

                                 Ellipse

                                 Line

                                 Path

                                 Polygon

                                 Polyline

                                 Rectangle

As a result of this illustrious ancestry, objects based on the Shape classes can render themselves on the screen and handle mouse, stylus, and keyboard input, much like regular controls.

The Canvas panel is excellent for displaying graphics because that's what it was made for**.**

 Path could potentially be the only vector-drawing class

you'll ever need. The only real drawback of Path is that it tends to be a little verbose in

comparison with the other Shape classes. However, toward the end of this chapter I'll show you

a shortcut that Path implements that lets you be quite concise.The only property that Path defines is Data, which you set to an object of type Geometry.

Because all the Geometry objects in the GeometryGroup are part of the same Path element,

they all have the same Stroke and Fill brushes. That's one major difference between using

GeometryGroup rather than multiple Path elements. CombinedGeometry is a combination of two and only two other geometries. The second difference between GeometryGroup and CombinedGeometry is that CombinedGeometry doesn't have a FillRule property. Instead it has a GeometryCombineMode

property that you set to a member of the GeometryCombineMode enumeration: Union,

Intersect, Xor, or Exclude. The first three options work like visual Venn diagrams; the Exclude

option creates a geometry that consists of everything in Geometry2 that's not also in

Geometry1.

It is with the Geometry derivative named PathGeometry that we really get into the central focus

of this whole geometry system. A graphics path is a collection of straight lines and curves,

some of which might or might not be connected to each other. Any set of connected lines and

curves within the path is known as a subpath, orto use the synonymous term consistent with

the WPF classesa figure. Thus, a path is composed of zero or more figures.

Here's the brief rundown: A PathGeometry object is a collection of one or more PathFigure

objects. Each PathFigure is a collection of connected PathSegment objects. A PathSegment is a

single straight line or curve.

You use geometries not only for drawing but for clipping. UIElement defines a property named

Clip of type Geometry that you can set to any Geometry object, or you can set Clip directly to a

path mini-language string directly in XAML. Any part of the geometry that would not normally

be filled if the geometry were being drawn is not displayed by the element.

RenderTransform supports re-locate; LayoutTransform won't relocate the UIElement.

The WPF graphics and layout system handles RenderTransform and LayoutTransform very

differently. With RenderTransform, the system takes the image drawn by the element's

OnRender method, applies the transform, and slaps the image on the screen. If that image

happens to overlay some other control in the program (or be buried underneath), it doesn't

matter. The element will be clipped to the border of the application's window, but it's otherwise

free to roam.

Any change in an element's LayoutTransform, however, precipitates a new layout pass with

calls to MeasureOverride and ArrangeOverride so that the transform can be respected when the

element is accommodated in the layout. The MeasureOverride, ArrangeOverride, and OnRender

methods don't need to know about the LayoutTransform, but certain values are finagled to

account for layout. The DesiredSize property of an element's child reflects the LayoutTransform,

for example, even though the child's MeasureOverride method doesn't take the transform into

account.

 If you are transforming an element in the context of a layout, and you want the transformed

element to be reflected in that layout, you'll probably be using LayoutTransform. If the

transform is set just once and doesn't change, there's really no performance difference between

RenderTransform and LayoutTransform. However, if you're animating a transform or binding

the transform to something that could change a lot, you'll probably want to use

RenderTransform for the sake of efficiency.

Any translation applied to a LayoutTransform is ignored, so when you're setting a LayoutTransform, you needn't bother with translation or any CenterX or CenterY properties. The

enlarged, skewed, or rotated element is always positioned in the space allocated for it during

layout.

The Geometry class defines a property named Transform of type Transform, so you can apply a transform directly to a geometry object. This is useful if you're using a Geometry object for clipping, for example. You can also set the transform even if the Geometry object is part of a Path. In effect, the geometry transform is applied before any possible RenderTransform or LayoutTransform of the Path itself. Because the geometry transform affects the coordinates and sizes that make up the Path, the transform will be reflected in layout.

Watch out: Whenever you create a coordinate system in which values of Y increase going up, any text you display on the Canvas will be upside down! If that's not what you want, you'll have to compensate by setting a ScaleTransform on the element displaying the text. In addition, rotations based on positive angles are counterclockwise rather than clockwise.

A couple of features of the WPF help make this animation a success. The retained graphics system is designed so that visual objects can change size or position without flickering or causing excessive redrawing. The dependency property system ensures that changing just the FontSize property of the button will make the button larger to accommodate the new size.

WPF animations always target dependency properties. The System.Windows.Media.Animation namespace has many classes because different classes exist for animating properties of various types. In alphabetical order, the 22 animatable types are Boolean, Byte, Char, Color, Decimal, Double, Int16, Int32, Int64, Matrix, Object, Point, Point3D, Quaternion, Rect, Rotation3D, Single, Size, String, Thickness, Vector, and Vector3D.

The FontSize property is of type double, so to animate the FontSize property, you can use a double animationan animation that changes the size of a dependency property of type double. Properties of type double are very common throughout the WPF, and consequently, the double is probably the most common animated type.

The two C# programs shown so far begin the animation in response to a Click event from a Button. However, the animation could be started based on other criteria. For example, you could set a timer that checks the time of day and start an animation at the stroke of midnight. Or a program might use the FileSystemWatcher class to monitor the file system and start an animation whenever a new directory is created. Or a program might display an animation based on information obtained from a second thread of execution in the program.

Animations defined in XAML, however, are always associated with triggers. You are familiar with triggers and trigger collections from Chapters 24 and 25. The Style, ControlTemplate, and DataTemplate classes all define properties named Triggers that are collections of TriggerBase objects. In Chapter 24, I discussed four of the classes that derive from TriggerBase. These are Trigger, MultiTrigger, DataTrigger, and MultiDataTrigger. As you'll recall, these triggers have the power to change properties of elements based on changes in other properties or changes in data bindings.

The Triggers collection defined by FrameworkElement can contain only EventTrigger objects, and you can't do much with EventTrigger objects except to trigger animations or play sounds.

EventTrigger defines three properties. The SourceName property is a string that refers to an element's Name or x:Name attribute. This is the element with the event that triggers the animation. (In common practice, the SourceName property is often implicitly defined by the context of EventTrigger and doesn't need to be explicitly specified.) The RoutedEvent property of EventTrigger is the particular triggering event. EventTrigger also defines an Actions property that indicates what happens when that event occurs. The Actions property is a collection of TriggerAction objects, and the most important class that derives from TriggerAction is BeginStoryboard.

A Triggers section of a Button control might look something like this:

<Button.Triggers>     <EventTrigger RoutedEvent="Button.Click">         <BeginStoryboard ... >             ...         </BeginStoryboard>     </EventTrigger> </Button.Triggers>

Why didn't I use a Style to reduce the repetitious markup in ColorRadiusButtons.xaml? It is certainly possible to include animations in a Triggers section of a Style definition, but the animations in the Style cannot target an element other than the one being styled.

When you define an animation in an element, it needs to be triggered by an EventTrigger in the Triggers section of that element. In a Style, however, you don't need to trigger the animation with EventTrigger. Here's an alternative approach to the fish-eye buttons that triggers the animation with a regular Trigger for the property IsMouseOver.

An animation can be controlled while in progress by making use of classes that derive from ControllableStoryboardAction. The following class hierarchy shows how these classes fit in with the other derivatives of TriggerAction.

Object

     DispatcherObject (abstract)

          DependencyObject

               TriggerAction (abstract)

                    SoundPlayerAction

                    BeginStoryboard

                    ControllableStoryboardAction (abstract)

                         PauseStoryboard

                         RemoveStoryboard

                         ResumeStoryboard

                         SeekStoryboard

                         SetStoryboardSpeedRatio

                         SkipStoryboardToFill

                         StopStoryboard

Although AccelerationRatio and DecelerationRatio are handy, you have no control over the rate at which the animation speeds up or slows down. If you need that controlor if you want to create an animation that is more complex than a linear change from one value to anotheryou'll want to explore the key-frame animation.

A <Type>KeyFrame class and a <Type>KeyFrameCollection class exist for all 22 animatable types. All 22 types have a Discrete<Type>KeyFrame class; all but five types have Linear<Type> KeyFrame and Spline<Type>KeyFrame classes.

Boolean

Byte

Char

Color

Decimal

Double

Int16

Int32

Int64

Matrix

Object

Point

Point3D

Quaternion

Rect

Rotation3D

Single

Size

String

Thickness

Vector

Vector3D

The Spline<Type>KeyFrame interpolates between the starting and ending values based on a spline rather than a straight line. Using this key frame, you can get effects similar to the Acceleration and Deceleration properties but with much more control.

The Spline<Type>KeyFrame classes inherit the KeyTime and Value properties and define just one additional property named KeySpline, which consists of just two control points of a Bezier curve. The Bezier curve is assumed to begin at the point (0, 0) and end at the point (1, 1). The two points specified in the KeySpline must have X and Y coordinates not less than 0 or greater than 1. With the values of the control points restricted in this way, it's not possible for the Bezier curve to loop. The resultant curve defines a relationship between time (the X axis) and the value of the animation (the Y axis).

 With SplinePointKeyFrame, the interpolated point always lies on the straight line between a starting point and an ending point. Elapsed time is used to calculate an X coordinate of the spline (between 0 and 1), and the Y coordinate of the spline (also between 0 and 1) is then used to calculate the interpolated value.

With PointAnimationUsingPath, time is allocated based on the total length of the path you've set to the PathGeometry property. For example, if the animation has a Duration of four seconds, and one second has elapsed, the animation determines the X and Y coordinates of the path one-quarter of the distance from the beginning to the end. These become the Point value of the animation

However, the graphics capabilities of the Microsoft Windows Presentation Foundation (WPF) are

really not so clearly divided between raster graphics and vector graphics. It's true that the Image class is used mostly to display bitmaps, but the class is not restricted to bitmaps. You can also use Image to display objects of type DrawingImage; you got a little taste of this capability in the About box in the YellowPad program in Chapter 22, which uses DrawingImage to display my signature. A DrawingImage object is always based on a Drawing object, and the word drawing usually refers to a picture composed of vector graphics elements, but Drawing is not restricted to vector graphics. A Drawing object can actually be a mix of vector graphics, raster graphics, and video.

DrawingImage has a rather exalted position in the class hierarchy. It seems to be positioned as the vector equivalent to bitmaps, which might imply that DrawingImage is being groomed to become an interchange medium for vector graphicsperhaps the XAML equivalent of graphics metafiles. There is no indication that Microsoft has such big plans for DrawingImage, but if you want to think of DrawingImage as a graphics metafile, you won't be too far off.

As you'll recall, a class that derives from UIElement or FrameworkElement can override the

OnRender method to draw the element. The single parameter to OnRender is an object of type

DrawingContext. It is also possible to create an object of type DrawingVisual and to use the

RenderOpen method to obtain a DrawingContext object. You then call methods of the

DrawingContext class to draw on this object, and you conclude by calling Close. You're left with a DrawingVisual object that stores the graphics. This is the technique you use when printing and

when displaying graphics on a bitmap of type RenderTargetBitmap.

CanExecute will be frequenetly called and please consider the performance.

If the CanExecute status of a command changes outside of a change in focus (such as when an asynchronous operation completes), call *CommandManager.InvalidateRequerySuggested*.

Depending on a specific application state, my CanExecute method will return true or false.  When the application starts it returns false which makes the button is disabled, as I expected.  As the application runs, its state is changed, but the button is still disabled because it didn't call CanExecute to get the new value.   What's strange is that when I interact with the UI - say I move a slider that has nothing to do with that button - then CanExecute may finally be called and the button becomes enabled.

So at this point I have 3 questions:

\* What causes CanExecute to be called besides the execution of the command ?

\* Why interacting with another interface element **may** cause CanExecute to be called ?

\* How can I force CanExecute to be called when the state of my application changes or when a custom event is raised in my application ?

First is that the CommandSource, your button in this case, is listening to the CanExecuteChanged event on the RoutedCommand it is bound to. So, if the ability of the command to execute on the current command target changes, the command raises the CanExcuteChanged event.  The button (the command source) receives the event and calls the CanExecute method on the command to determine what the current execute status is and if it should enable or disable itself.  If you were to create your own control to act as a command source, you would want to listen for this event to determine when you needed to requery the command status.

What I think you’re seeing when you interact with the UI is that CommandTarget is changing.  The CommandSource can hardcode a CommandTarget and it will always use that object as the target, but if a CommandTarget is not set, then the element with keyboard focus becomes the target.  So the CommandTarget can move around.  Consider if you have a number of objects, like Textboxes, that will handle the same RoutedCommand, say the Paste command.  If a CommandSource, say a MenuItem or a KeyGesture, invokes the command you’ll want the TextBox with Focus to handle the command and not one of the other text boxes.

It may be the case that it is valid for the command to execute on one TextBox but not the other (Read only or some other condition), so when focus changes to a different TextBox and the ability of the command to execute on the current target changes, the CanExecuteChanged event is raised.   The CommandSource handles the CanExecuteChanged event and querys the RoutedCommand using the CanExecute method to see if it can invoke the command on the current target.

One thing to keep in mind is that RoutedCommands route through the element tree.  So, you can run into some unexpected behavior if you have a CommandSource that does not set the CommandTarget (so the command target is the element with focus), but the element you are expecting to be the target is a sibling of the button.  The button will never receive the CanExecuteChanged event, so it will never be enabled. Also, when a command is being handled, it will traverse the tree looking for a CommandBindings.  In the case of our TextBox, the binding to handle the ApplicationCommands.Paste command is attached to the TextBox, but it may be more desirable from a design standpoint to attach all your CommandBindings to the root window.  This allows you to centralize your bindings in one place.

CommandManager.InvalidateRequerySuggested()  only works if it's called from the main thread, so use the delegate in a timer.

These ten classes have the following significance:

. Object—The base class for all .NET classes.

. DispatcherObject—The base class for any object that wishes to be accessed only on

the thread that created it. Most WPF classes derive from DispatcherObject, and are

therefore inherently thread-unsafe. The Dispatcher part of the name refers to WPF’s

version of a Win32-like message loop, discussed further in Chapter 7, “Structuring

and Deploying an Application.”

. DependencyObject—The base class for any object that can support dependency

properties. DependencyObject defines the GetValue and SetValue methods that are

central to the operation of dependency properties.

. Freezable—The base class for objects that can be “frozen” into a read-only state for

performance reasons. Freezables, once frozen, can even be safely shared among

multiple threads, unlike all other DispatcherObjects. Frozen objects can never be

unfrozen, but you can clone them to create unfrozen copies.

. Visual—The base class for all objects that have their own visual representation.

Visuals are discussed in depth in Chapter 11.

. UIElement—The base class for all visual objects with support for routed events,

command binding, layout, and focus.

. ContentElement—A base class similar to UIElement, but for pieces of content that

don’t have rendering behavior on their own. Instead, ContentElements are hosted in

a Visual-derived class to be rendered on the screen.

. FrameworkElement—The base class that adds support for styles, data binding,

resources, and a few common mechanisms for Windows-based controls such as

tooltips and context menus.

. FrameworkContentElement—The analog to FrameworkElement for content. Chapter

14 examines the FrameworkContentElements in WPF.

. Control—The base class for familiar controls such as Button, ListBox, and

StatusBar. Control adds many properties to its FrameworkElement base class, such

as Foreground, Background, and FontSize. Controls also support templates that

enable you to completely replace their visual tree, discussed in Chapter 10. The next

chapter examines WPF’s Controls in depth.

                            Object

                                  |

                            DispatcherObject

                                  |

                            DependencyObject

                                  |

        ---------------------------------------------------------------

        |                         |                                                       |

Freezable          Visual                                                   |

                                  |                                                       |

                            UIElement                            ContentElement

                                  |                                                   |

                            FrameworkElement         FrameworkContentElement

                                  |

                            Control

Content and Arbitrary Objects

Given that a content control’s Content can be set to any managed object, it’s natural to

wonder what happens if you set the content to a non-UI object, such as an instance of a

Hashtable or RegistryKey. The way it works is fairly simple. If the content derives from

WPF’s UIElement class, it gets rendered via UIElement’s OnRender method. Otherwise, if a

data template is applied to the item (described in Chapter 9, “Data Binding”), this template

can provide the rendering behavior on behalf of the object. Otherwise, the content’s

ToString method is called and the returned text gets rendered inside a TextBlock.

Elements hosting content that isn’t native to WPF do not support transforms, despite inheriting

the LayoutTransform and RenderTransform properties

In WPF, you can even give your application multiple UI threads by calling Dispatcher.Run in any new thread that you spawn. Therefore, you could make each Window run on a separate thread if your application has more than one top-level Window. Doing this is certainly not necessary for most applications, but such a scheme could improve your application’s responsiveness if it’s likely that one Window could start activities that would dominate the thread.

Resource—Embeds the resource

into the assembly (or a culturespecific

satellite assembly).

Content—Leaves the resource as a loose file, but adds a custom attribute to the

assembly (AssemblyAssociatedContentFile) that records the existence and relative

location of the file.

logical resources are arbitrary .NET objects stored (and named) in an element’s

Resources property, typically meant to be shared by multiple child elements, which are often styles or data providers.

WPF provides two ways to access a logical resource:

. Statically with StaticResource, meaning the resource is applied only once (the first

time it’s needed)

. Dynamically with DynamicResource, meaning the resource is reapplied every time it

changes

Data-binding errors don’t appear as unhandled exceptions!

Value converters are the key to plugging in any kind of custom logic into the data-binding

process.

In WPF, the term data is generally used to describe an arbitrary .NET object.

The key to data binding is a System.Windows.Data.Binding object that“glues” two properties together and keeps a channel of communication open between them .

BindingOperations.SetBinding(currentFolder, TextBlock.TextProperty, binding);

The benefit of this static method is that the first parameter is defined as a

DependencyObject, so it enables data binding on objects that don’t derive from

FrameworkElement or FrameworkContentElement (such as Freezables).

WPF supports any .NET property on any .NET object as a data-binding source.

Because such properties have no automatic plumbing for change notification, the target is

not kept up to date as the source property value changes without doing a little extra

work. Therefore, the value displayed in Figure 9.1 does not change as the current folder

changes, which is clearly incorrect.

To keep the target and source properties synchronized, the source object must do one of

the following:

. Implement the System.ComponentModel.INotifyPropertyChanged interface, which

has a single PropertyChanged event.

. Implement an XXXChanged event, where XXX is the name of the property whose

value changed.

How Binding to a Plain .NET Property Works

When retrieving the value of a source property that’s a plain .NET property, WPF uses reflection.

However, if the source object implements ICustomTypeDescriptor, WPF uses this interface

instead. Implementing this interface is an advanced technique, but it can be useful for

boosting performance or supporting additional scenarios (such as changing the set of properties

exposed on the fly).

Controling the data display though DataTemplate.

Bridging Incompatible Data Types using Converter.

Whereas XmlDataProvider exposes XML as a data source, ObjectDataProvider exposes a

.NET object as a data source

ObjectDataProvider provides more funcitions, compared with direct object binding:

. Declaratively instantiate the source object with a parameterized constructor

. Bind to a method on the source object

. Have more options for asynchronous data binding

Asynchronous Data Binding

Whenever binding to data isn’t a quick operation, it should be done asynchronously to

avoid freezing your user interface. WPF exposes two independent knobs for making binding

happen asynchronously: Binding has an IsAsync property and both XmlDataProvider

and ObjectDataProvider have an IsAsynchronous property.

Why would I ever use a Binding with a Mode of OneWayToSource? In such a

case, it sounds like the target should really be the source and the source

should really be the target.

One reason could be that you’re using multiple Bindings, some with data flowing from the

source to the target and others with data flowing from the target to the source. For example,

you might want to share a source among many data-bound targets, but want one of these

target elements to update that source via data binding.

OneWayToSource can also be used as a sneaky way to get around the restriction that a

Binding’s target property must be a dependency property. If you want to bind a source

dependency property to a target property that is not a dependency property, OneWayToSource

enables you to accomplish this by marking your “real source” as the target and your “real

target” as the source!

You can define you Binding.ValidationRule for validating the data.

Resources can be objects, styles, templates, triggers, or even system resources (like systemfont).

Styles—A simple mechanism for separating property values from user interface elements (similar to the relationship between Cascading Style Sheets [CSS] and HTML). They’re also the foundation for applying the other mechanisms in this chapter.

. Templates—Powerful objects that most people are really referring to when they talk about “restyling” in WPF.

. Skins—Application-specific collections of styles and/or templates, typically with the ability to be replaced dynamically.

. Themes—Visual characteristics of the host operating system, with potential customizations by the end user.

Styles can even inherit from one another

You don’t need to worry if a Style is applied to an element that doesn’t have all the listed

dependency properties; the properties that exist are set and the ones that don’t are

ignored

Remember the shared dependency properties when using setter in the styles.

With a named style, it’s okay for the target element to be a subclass of the TargetType. But

a typed style typically only gets applied to elements whose type matches exactly!

Triggers, first introduced in Chapter 3, have a collection of Setters just like Style (and/or

collections of TriggerActions). But whereas a Style applies its values unconditionally, a trigger performs its work based on one or more conditions. Recall that there are three types of triggers:

. Property triggers—Invoked when the value of a dependency property changes

. Data triggers—Invoked when the value of a plain .NET property changes

. Event triggers—Invoked when a routed event is raised

FrameworkElement, Style, DataTemplate, and ControlTemplate all have a Triggers collection, but whereas Style and the template classes accept all three types, FrameworkElement only accepts event triggers. (This is simply because the team didn’t have enough time to implement the support for version 3.0.)

If multiple triggers that have conflicting Setters are active simultaneously, the last one wins.

The same is true of conflicting Setters inside a single trigger.

If you want to add even more complex event-driven behavior to a Style, you can make use of

an EventSetter (which shares a common base class with Setter) to attach an event

handler to any element that makes use of the Style. EventSetters can be added to a

Style just like Setters

A template, on the other hand, allows you to completely replace an element’s visual tree

with anything you can dream up, while keeping all of its functionality intact

Naming an element with x:Name inside a template does not make it become a field to

access programmatically, unlike its behavior outside of a template. This is because a

template can be applied to multiple elements in the same scope. The main purpose of

naming elements in a template is for referencing them from triggers (typically defined in

XAML). But if you want programmatic access to a named element inside a template, you can

use the template’s FindName method after the template has been applied to a target

The main difference with drawing in WPF

versus GDI or any previous Windows technology is that

WPF is a completely retained-mode graphics system rather

than an immediate-mode graphics system.

In an immediate-mode system (GDI, GDI+, DirectX, and so

on), you can draw “directly” onto the screen, but you must

maintain the state of all visuals. In other words, it’s your

responsibility to draw the correct pixels when a region

of the screen is invalidated. This invalidation can be

caused by user actions, such as resizing the window, or by

application-specific actions that require updated visuals.

In a retained mode system, you can describe higher-level

concepts such as “place a 10x10 blue square at (0,0),” and

the system remembers and maintains the state for you. So,

what you’re really saying is, “place a 10x10 blue square at

(0,0) and keep it there.” You don’t need to worry about invalidation and repainting, so this can save a significant amount of work. It’s also the

key to WPF’s seamless support for overlapping objects, transparency, video, resolution

independence, effective UI remoting, and so on.

To get Drawings rendered appropriately, you can place them inside one of three different

host objects:

. DrawingImage—Derives from ImageSource, so it can be used inside an Image rather

than the typical BitmapImage

. DrawingBrush—Derives from Brush, so it can be applied in many places, such as the

Foreground, Background, or BorderBrush on a Control

. DrawingVisual—Derives from Visual, and is covered in the “Visuals” section

DrawingImage is an ImageSource, enabling a typically vector-based Drawing to be its

content rather than something bitmap-based. Conversely, ImageDrawing is a Drawing,

enabling a typically bitmap-based ImageSource to be its content rather than something

vector based.

A Geometry is the simplest possible abstract representation of a shape or path.

The major advantage of using GroupGeometry is that you can set various Geometry properties independently on each child.

a different Visual subclass provides a lightweight mechanism for rendering

Drawings onto the screen: DrawingVisual. DrawingVisual has a few handy properties for

controlling rendering aspects, such as Opacity and Clip (which DrawingGroup also

happens to have). But it also has support for a minimal amount of interaction with input

devices. This comes in a form of hit testing called visual hit testing.

Therefore, the DrawingContext class is WPF’s closest analog to the Win32 device context or the Windows Forms Graphics object. Note that the use of DrawingContext doesn’t change the fact

that you’re operating within a retainedmode system.

Using DrawingContext is the most lightweight way to perform drawing because it can avoid the overhead of allocating a Drawing object on the managed heap for every line, shape, and so on. Therefore, it’s the best choice for rendering tens of thousands of items.

In WPF, there are two kinds of hit testing: visual hit testing, which is supported by all

Visuals, and input hit testing, which is only supported by UIElements.

A Shape, like a GeometryDrawing, is a basic 2D drawing that combines a Geometry with a

Pen and Brush. Unlike GeometryDrawing, however, Shape derives from FrameworkElement,

so it can be directly placed in a user interface without custom code or a complex hierarchy

of objects.

When you have Shape-based artwork, every single Shape supports Styles, data binding,

resources, layout, keyboard/mouse/stylus input and focus, routed events, and so on. It’s nice

that you can take advantage of all this without extra work, but as discussed in the “Visuals”

section, this is typically unnecessary overhead. Keep this in mind if you find yourself using

more than a small number of Shapes.

Shapes internally override UIElement’s OnRender method and use DrawingContext methods

to draw the appropriate geometry.

Input hit testing differs from visual hit testing in that it more closely represents what a user can physically hit with her mouse, keyboard, or stylus. It only supports hitting the topmost element at any coordinate, and only allows elements to be hit if IsEnabled and IsVisible (properties introduced by UIElement) are both true.

As with many things in WPF, there are multiple ways to create and use two-dimensional

graphics. The heart of this chapter focuses on three important data types: Drawings,

Visuals, and Shapes. Their relationship to each other is complex. For the most part,

Drawings are simple descriptions of paths and shapes with associated fill and outline

Brushes. Visuals are one way to draw Drawings onto the screen, but Visuals also unlock a

lower-level and lighter-weight approach for drawing that enable you to ditch Drawing

objects altogether. Finally, Shapes are prebuilt Visuals that are the easiest (but most

heavyweight) approach for drawing custom artwork onto the screen. As we examine

Drawings, Visuals, and Shapes, we’ll look at a simple piece of clip art and see what it

means to create and use it in all three contexts.

3D content is not constrained to a box. Scenes contained within a Viewport3D are composed seamlessly with other UI elements in your application and can be included in templates and ItemsControls. Likewise, 2D media such as video, Drawings, and Visuals can be displayed on the surfaces of 3D models. Services such as hit testing automatically continue into the 3D portions of the Visual tree.

|  |  |  |
| --- | --- | --- |
| Drawing | Model3D | Drawings represent pieces of 2D content, such as  clip art, which may be rendered by a Visual.  Model3Ds represent pieces of 3D models, which may  be rendered by a Visual3D. |
| Geometry | Geometry3D | A Geometry represents a 2D shape. Geometries can  answer questions about bounds and intersections.  By itself, a Geometry cannot be rendered. A  GeometryDrawing combines a Geometry with a  Brush to give it an appearance.  A Geometry3D represents a 3D surface. To render a  Geometry3D it is combined with a Material using a  GeometryModel3D |
| Visual | Visual3D | Visual is the base class for elements that render 2D  content. This includes DrawingVisual and all  FrameworkElements such as Controls and Shapes.  Visual3D is the base class for elements that render  3D content. ModelVisual3D is a concrete Visual3D  that renders 3D content represented as Model3Ds. |
| Transform | Transform3D | Subclasses of the 2D Transform class are used to  position, rotate, and size 2D Drawings and Visuals.  There were no Transform3Ds in Listing 12.2, but  when you encounter the 3D transform objects later in  this chapter you will see that they perform the same  function for Model3Ds and Visual3Ds. |
|  |  |  |

Unlike in 2D where Drawings are one of many ways to add 2D content to a WPF application, Model3Ds are the only way to declare 3D content in the current version of WPF.

If you’re not using a SolidColorBrush, you need TextureCoordinates!

Although you could put your entire scene together using a Model3DGroup displayed under a

single ModelVisual3D, you will be missing out on some important performance optimizations

if you do this. ModelVisual3Ds are optimized to be scene nodes. They cache bounds and

other information that lightweight Model3DGroups do not.

Going to the other extreme, you could use ModelVisual3Ds for each and every

GeometryModel3D in your scene. This is inadvisable because it unnecessarily increases the

working set of your application. Model3DGroups are lightweight constructs intended for group-

ing multiple GeometryModel3Ds into a single model.

In general, you should use Model3DGroups to combine the various pieces of a single model

(for example, the tires, windshield, and body of a car). ModelVisual3Ds should be used for

displaying instances of 3D models (for example, use a ModelVisual3D for each car you add

to the scene).

Whereas rendering primitives like Drawing and Model3D are sealed for extension, Visuals

are not. ModelVisual3D is the right class to extend from if you want to create a reusable 3D

element that renders and potentially has some custom behaviors.

Entire 3D scenes can be used as DataTemplates and ControlTemplates

sing the frame-based Rendering event is not only preferred over a timer-based approach,

but it’s even preferred over the animation classes that are the focus of this chapter when

dealing with hundreds of objects that require high-fidelity animations. For example, colli-

sion detection or other physics-based animations should be done using this approach.

In normal conditions WPF only renders frames

when part of the UI is invalidated. But as long as any event handler is attached to

Rendering, WPF renders frames continuously. Therefore, Rendering is best for short-lived animation.

The choices for audio are represented by several different

classes:

. SoundPlayer

. MediaPlayer

. SoundPlayerAction

. MediaElement and MediaTimeline

WPF’s support for audio, video, speech, and documents rounds out its rich media offer-

ings. The audio support is limited, but is enough to accomplish the most common tasks.

The video support is only a subset of what’s provided by the underlying Windows Media

Player APIs, but the seamless integration with the rest of WPF (so you can transform or

animate video just like any other content) makes it extremely compelling.

WPF’s speech synthesis and recognition support is state of the art, but at the same time

it’s mainly just a wrapper on top of the unmanaged Microsoft SAPI APIs. Contrary to the

other three areas, however, WPF’s rich support for documents is brand-new functionality

that you can’t find anywhere else in Microsoft’s technology offerings. Many people see

documents support as WPF’s “killer app.” It is the technology behind the Times Reader

application from the New York Times, and will undoubtedly be used in other applications

that push the boundaries of user experiences.

Because WPF is built on top of DirectX, isn’t there a more direct way to

have WPF interoperate with DirectX surfaces?

n version 3.0 of WPF, there is no direct “DirectX interoperability” feature. Although there are

managed APIs for DirectX, you should not try to call them within WPF applications. But

because all DirectX content (like WPF content) must ultimately be hosted in an HWND, you can

use HWND interoperability as the means for mixing these two technologies.

?

You cannot overlap WPF content with non-WPF content!

As with hosting HTML content in Frame, any non-WPF content that’s hosted in a WPF applica-

tion has extra limitations that don’t apply to native WPF content. For example, you can’t apply

Transforms to non-WPF content. Furthermore, you cannot overlap content from one

technology over content from another. You can arbitrarily nest (for example) Win32 inside

WPF inside Windows Forms inside WPF, and so on, but every pixel must have one and only

one technology responsible for its rendering.

In fact, C++/CLI is undergoing Ecma standardiza-

tion at the time of writing (like the CLI and C#). Just to put some context around these stan-

dards: Visual C++ is Microsoft’s implementation of C++/CLI, Visual C# is Microsoft’s

implementation of C#, and the common language runtime (CLR) is Microsoft’s implementa-

tion of the CLI.

With Visual C++’s /clr compiler option, you can compile entire projects or individual source

files as managed code. It’s tempting to simply compile entire projects as managed code, but

it’s usually best if you decide on a file-by-file basis what should be compiled as managed and

what should be compiled as unmanaged. Otherwise, you could create extra work for yourself

without any real gain.

The /clr option works well, but it often increases build time and can sometimes require

code changes

WPF must run on an STA thread!

As with Windows Forms and earlier technologies, the main thread in an application using WPF

must live in a single-threaded apartment. In Listing 15.6, the STAThreadAttribute must be

applied to the entry point because the entire file is compiled as managed code, and

managed code defaults to MTA.

What about the reverse direction of exposing WPF controls as ActiveX controls?

There is no built-in support for this above and beyond HWND interoperability, so your best bet

is to use your favorite means of creating a non-WPF ActiveX control (using Active Template

Library [ATL], for example) and injecting WPF content inside of it.

HwndHost Hosting an HWND in WPF

WindowsFormsHost   Hosting Windows Forms in WPF

HwndSource Hosting WPF in an HWND

ElementHost Hosting WPF in Windows Forms

*Custom* ***Control***

*1. The purpose of it is to enhance the existing* ***control****,*

*2. It supports theming for consumers, which means the consumers can style it in whatever way they like,*

*3. It is perfect to be 3rd-party* ***control****, especially because of point 2.*

*Whereas User* ***Control***

*1. Its purpose is to compose controls into one piece,*

*2. It does not support theming for consumers. Therefore, the consumers can not restyle it,*

*3. It better stay with the consumer application in case the consumer needs to change its style somehow.*