[What is WPF 3](#_Toc342400260)

[Features of WPF 3](#_Toc342400261)

[WPF Architecture 3](#_Toc342400262)

[XAML Basics 4](#_Toc342400263)

[Layout Phases 6](#_Toc342400264)

[Measure Phase: 6](#_Toc342400265)

[Arrange: 6](#_Toc342400266)

[Markup Extensions 6](#_Toc342400267)

[Visual Tree versus Logical Tree 8](#_Toc342400268)

[WPF Class Hierarchy 10](#_Toc342400269)

[Dependency Property 13](#_Toc342400270)

[Change Notification 13](#_Toc342400271)

[Property Value Inheritance 13](#_Toc342400272)

[Support for Multiple Providers 14](#_Toc342400273)

[The Magic Behind Dependency Properties 15](#_Toc342400274)

[How to create a DependencyProperty 16](#_Toc342400275)

[ReadOnly DependencyProperty 16](#_Toc342400276)

[Attached Properties 17](#_Toc342400277)

[Observable Collection 18](#_Toc342400278)

[Layout 19](#_Toc342400279)

[Panels 19](#_Toc342400280)

[StackPanel 19](#_Toc342400281)

[Grid 19](#_Toc342400282)

[UniformGrid 20](#_Toc342400283)

[GridSplitter 20](#_Toc342400284)

[WPF Content Control 20](#_Toc342400285)

[Canvas (for precision layout) 20](#_Toc342400286)

[ScrollViewer 21](#_Toc342400287)

[Decorators 21](#_Toc342400288)

[Viewbox 21](#_Toc342400289)

[Data Binding 25](#_Toc342400290)

[Styles, Templates, Skins And Themes 32](#_Toc342400291)

[Styles 32](#_Toc342400292)

[Tip 1: 33](#_Toc342400293)

[Tip 2: 33](#_Toc342400294)

[Restricting Use of Styles 33](#_Toc342400295)

[Triggers 34](#_Toc342400296)

[Trick to use triggers everywhere 34](#_Toc342400297)

[Pointers 35](#_Toc342400298)

[Difference Between Margin and Padding 35](#_Toc342400299)

[To Practice 37](#_Toc342400300)

[Amazing Website Links 37](#_Toc342400301)

[Overall WPF Links 38](#_Toc342400302)

**Windows Presentation Foundation**

What is WPF

**Short Answer** - WPF is an API for building graphical user interfaces (UI) for desktop applications with the .NET Framework.  
**Long Answer** - It’s a set of .NET assemblies and supporting tools. It’s intended to provide a unified API for creating rich, sophisticated user interfaces on Windows XP and Windows Vista. WPF combines the good things from web development, such as style sheets and a markup language for declarative UI, with good things from Rich Internet Applications, such as scalable vector graphics, animation, and media support. These good things are wrapped up with the good things from traditional Windows development—things like strong integration with the OS and data binding. In WPF, these concepts are strengthened and unified.

**Relation to Windows Forms**

The immediate predecessor to WPF is Windows Forms, the graphical API available to developers in .NET 2.0 and earlier. Windows Forms provides a managed wrapper for accessing the graphical functions of the traditional Windows API. WPF differs fundamentally in that it builds on top of DirectX. The DirectX API was originally focused on multimedia and game programming in particular. As such, you are able to do some nifty visual tricks in WPF that were practically impossible with Windows Forms. It also means that WPF will take advantage of hardware acceleration when it is available.

Features of WPF

* Declarative UI - WPF allows you to construct your interface using a markup language called XAML. XAML is a much richer markup language than HTML, and it has less ambiguity.
* Intelligent Layout - WPF provides an extensible layout system for visually arranging the elements of a user interface. It can intelligently resize and adjust, depending on how you define the layout.
* Scalable Graphics - Graphics in WPF are vector based, in contrast to raster based. Vector graphics are inherently scalable and typically require less storage than a comparable raster image. The net result for developers with WPF is that applications scale nicely without a loss in visual quality.
* Templates - WPF makes it very easy to create reusable elements for your user interfaces. There are two types of templates in WPF: control templates and data templates. Control templates enable you to redefine the way a control looks. Data templates are similar, except that instead of defining the way a control looks, they define the way certain types of data are rendered. In practice data templates are really handy when dealing with lists or other collections of data.
* Binding - Although WPF has significant data binding features, It also allows you to declaratively bind other things such as commands, key bindings, animation, and events. For example, you can declaratively bind a button control to a command for pasting.
* Styling - Styles in WPF are similar to cascading style sheets for HTML. Though again, WPF styles are richer and have less ambiguity. They encompass all the visual characteristics you would expect, such as padding, margin, position, color, and so on.
* Triggers - Both templates and styles in WPF support the notion of triggers. A trigger enables you to tell WPF something like this: “When the mouse is over the button, make the background purple.” In other words, triggers enable you to declaratively handle changes of state.
* Animation - Most properties in WPF can be animated, and support exists for timelines, key frames, and interpolation. Animations are easily integrated with templates and styles.

WPF Architecture

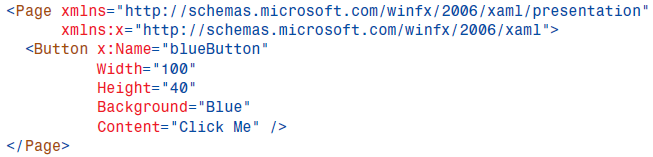
WPF has Presentation Framework, Presentation Core & Composition Engine (MIL). Each one sit on the top of the other in the following order

1. Presentation Framework - Most of the important WPF elements are in Presentation Framework. Holds top level WPF types which includes Window, Controls, Styles, and Layout Panels etc. The code and controls written in WPF Application is mostly interacting with this layer. Also provides high-level services like layout, data binding, command handling.
2. Presentation Core - Holds base types such as UI Element and Visual. Almost all the controls you are directly interacting with are derived from these types. Presentation Framework uses most of the types defined in this layer. It provides .NET API that uses rendering services for the MIL. During graphics programming we work with Core API.
3. Composition Engine - Is also called MIL, Media Integration Layer. It takes Bitmaps, Vectors, and media render them to the DirectX. MIL sits as unmanaged layer to minimize CPU usage.
4. MilCore – Media Integration Library is core rendering system. MIL is unmanaged code. This layer converts WPF elements into the format that Direct3D expects. Windows7 and Windows Vista uses this assembly to render its Desktop.
5. WindowsCodecs– provides supports for imaging like image processing, image displaying and scaling etc.

|  |  |
| --- | --- |
|  |  |

XAML Basics

* XAML is the language used for creating user interfaces in WPF. It’s an XML-based markup language similar to HTML, MXML, or XUL. In fact, it’s even possible to represent data with XAML, such as an array of strings or an instance of object.
* XAML can be either compiled or interpreted, depending on how it is used. When you create a WPF application with Visual Studio, the XAML used in the application is compiled into the resulting executable.
* When authoring XAML, your root element always defines two namespaces. The default namespace is specifically mapped to WPF, whereas the prefix x: is for XAML’s more generic features.
* Syntax - An element in XAML is an instance of an object and attributes are properties on that object.



The attributes on the Button element represent properties on an object instance. Thus, we are setting the values for the properties of Width, Height, Background, and Content.  
There is also the x:Name attribute, which breaks the rule here. x:Name is not a property on the Button class. Instead, it’s a special attribute that provides a unique identifier for the object for accessing it in code.

It is important to understand that any XAML element you want to reference, in code or elsewhere in the XAML, must have a unique value for x:Name. Providing a value from x:Name is like creating a variable and then setting the variable to the instance of the object.

* Setting Properties that are not Simple Type - On the Button class, Width and Height are simple value types. However, many properties on controls are not simple value types. Some properties are objects that have lots of properties themselves, which could also be complex types.  
   

In some situations, however, this shorthand is not sufficient for telling WPF what you want. **Property element syntax** is an alternative syntax used for providing values for complex types. Instead of setting the Background property using an attribute, we can use a child element. The following snippet demonstrates using this alternative syntax for setting the background to blue:

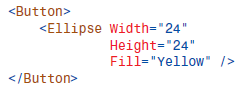


The child element is referred to as a property element. Property elements take the form of <ClassName.PropertyName />. That is, the first part of the element’s name is the class name, followed by a dot, followed by the property’s name.  
This syntax can become very verbose, but in many situations it is the only way to provide the exact value that you desire.

* Content Property - Many of the WPF controls that you will encounter have a property named Content. Content is a special property. we set the Content of a Button to a string value, Click Me. However, you can also set the Content property implicitly using a child element. For example, the following XAML elements are equivalent:

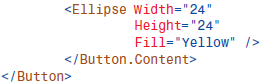


Both buttons will be rendered the same in WPF. What’s exciting is that Content is actually of type object. That means that we can make a button’s content much more than just a simple string. For example, perhaps we want to draw a yellow circle inside our button.



You can always set the Content property explicitly, too:





Layout Phases  
Layout occurs in 2 phases:  
 - Measure   
 - Arrange

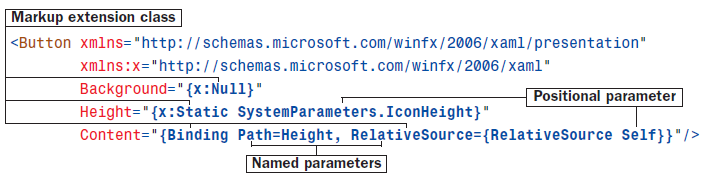
Measure Phase:   
Walk UI tree, determine preferred sizes: In the measure phase, WPF walks the entire visual tree and finds out how large each element needs to be. It does this by calling method on each element called measure.  
Constrained or Unconstrained: Constrained layout occurs when WPF knows how much space will be available or can make a good guess. Unconstrained layout occurs when the amount of space is unknown or unbounded. Occurs for both horizontally and vertically.  
Size to content: Elements are not allowed to ask an infinite amount of space during the measure phase so they must return a finite preferred size regardless of what is passed in.

Arrange: Once the measure phase is complete WPF knows how big each element wants to be. In the second phase of layout WPF walks the tree a second time and calls the Range on each element passing in size and location. Ofcourse this may be impossible – element may ask for more space than available. In this case WPF will compromise and truncate the element.

Markup Extensions

* Markup extensions enables a user to extend the expressiveness of XAML.
* It can evaluate a string attribute value at runtime (except for a few built-in markup extensions that are currently evaluated at compile time for performance reasons) and produce an appropriate object based on the string.
* WPF ships with several markup extensions built in and are preferred approach for extending XAML.

Whenever an attribute value is enclosed in curly braces ({}), the XAML compiler/parser treats it as a markup extension value rather than a literal string (or something that needs to be type-converted).   
The following Button uses three different markup extension values with three different properties:



The first identifier in each set of curly braces is the name of the markup extension class, which must derive from a class called MarkupExtension.

* In this example, NullExtension (seen as x:Null) and StaticExtension (seen as x:Static) are classes in the System.Windows.Markup namespace, so the x prefix must be used to locate them.
* Binding (which doesn’t happen to have the Extension suffix) is in the System.Windows.Data namespace, so it can be found in the default XML namespace.

StaticExtension enables the use of static properties, fields, constants, and enumeration values rather than hard-coding literals in XAML.

* In this case, the Button’s Height is set to the operating system’s current height setting for icons, exposed by the static IconHeight property on a System.Windows.SystemParameters class.

Binding enables Content to be set to the same value as the Height property.

|  |  |
| --- | --- |
| **Common Markup Extensions in WPF** | |
| Binding | The extension used for binding data. |
| x:Null | Used for specifying a null value in XAML. |
| x:Type | This extension is used for supplying a System.Type object. |
| X:Array | This allows you to define an array of objects in XAML. |
| StaticResource | * StaticResources are resolved at compile time. * This is used when the resource is not needed to be re-evaluated. |
| DynamicResource | Similar to StaticResource, except that the data in the resource might change during runtime. |
| RelativeResource | Allows you to bind source object on its relation to the target object which can be self, parent or template parent. |
| Difference between StaticResource & Dynamic Resource  The difference between them lies in how the resources are retrieved by the referencing elements.   * StaticResource are retrieved only once by the referencing element and used for entire life of the resource. * DynamicResource are acquired every time the referenced object is used. * Static resources perform better than dynamic resources. | |

**Scenario** - Suppose we have a specific color we want to use as the background for several buttons in a WPF application. We could set the Background property on each of the buttons to use the same color, but it would become tedious if we ever needed to change that color. With WPF, we can store the color with a lookup key in an application’s resources. Now we can set the background of the buttons to the color we stored in the resources. If we want to change the color, we need do so in only one place. That’s a lovely scenario, but how would we handle this in XAML?

We would use a markup extension.In the preceding scenario, the XAML with the markup extension might look like:



Markup extensions are identified by the presence of curly brackets ({}). The first word in the markup extension tells WPF what kind of extension it is. The name of the extension is optionally followed by a set of named parameters. In this case, the extension is for retrieving a shared resource from a library of resources. The name of the extension is StaticResource, and we provide a value of myColor for the ResourceKey parameter. Many extensions have a default parameter. You don’t have to explicitly reference a default parameter. You are allowed to omit the parameter name and the equal sign. For example, we could restate the snippet with ResourceKey=:

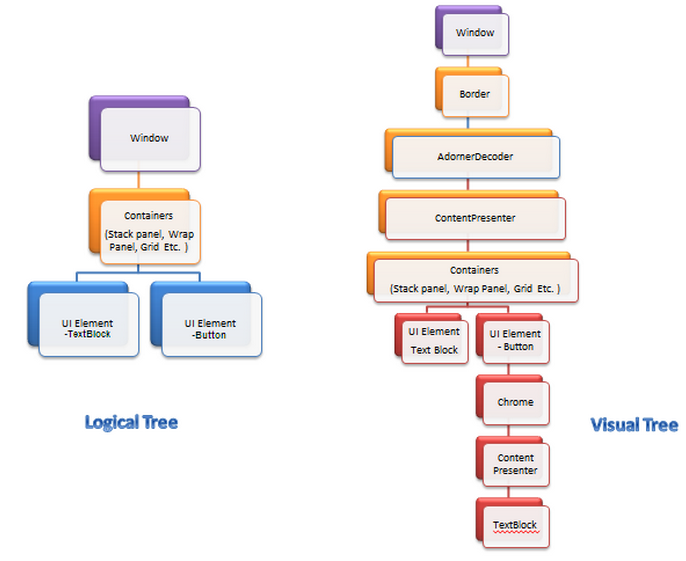


ResourceKey is the default parameter for StaticResource. In some cases, you will have more than one parameter. If you do, you must separate name/value pairs with commas.   
The general pattern is this: {ExtensionName Param1=Value1, Param2=Value2, Param3=Value3}.   
The most frequent mistake in dealing with markup extensions is to include quotation marks around the values. You are not allowed to have quotation marks between the curly brackets. This confuses the parser. It also means that parameter values cannot have whitespace.

More reading on Relative Sources - <http://www.c-sharpcorner.com/UploadFile/yougerthen/relativesources-in-wpf/>

Visual Tree versus Logical Tree

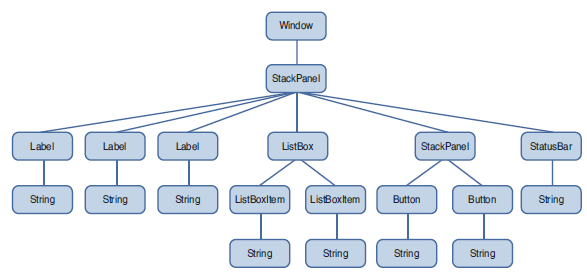
* The logical tree exposed by WPF is a simplification of what is actually going on when the elements are rendered.
* The entire tree of elements actually being rendered is called the visual tree. You can think of the visual tree as an expansion of a logical tree, in which nodes are broken down into their core visual components. For example, although a ListBox is logically a single control, its default visual representation is composed of more primitive WPF elements: a Border, two ScrollBars, and more.
* IMPORTANT - Not all logical tree nodes appear in the visual tree; only the elements that derive from System.Windows.Media.Visual or System.Windows.Media.Visual3D are included. Other elements (and simple string content, as in Listing 3.1) are not included because they don’t have inherent rendering behavior of their own.



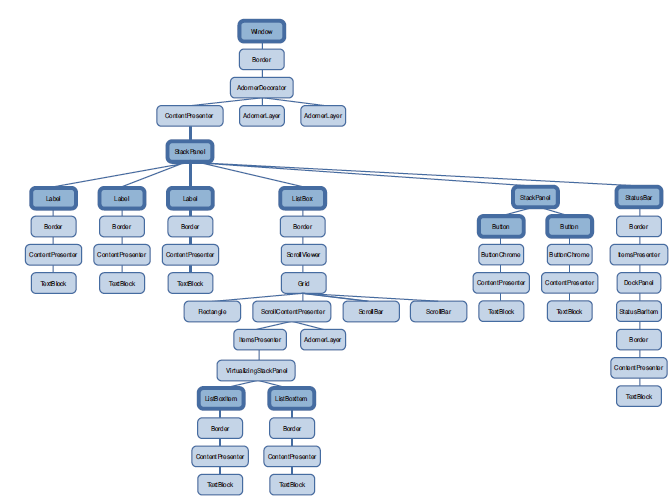
**Another Example –**

|  |  |
| --- | --- |
|  |  |

Logical Tree:

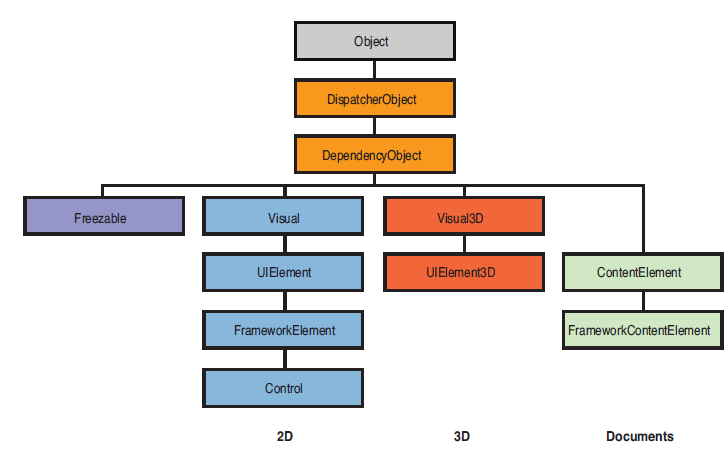


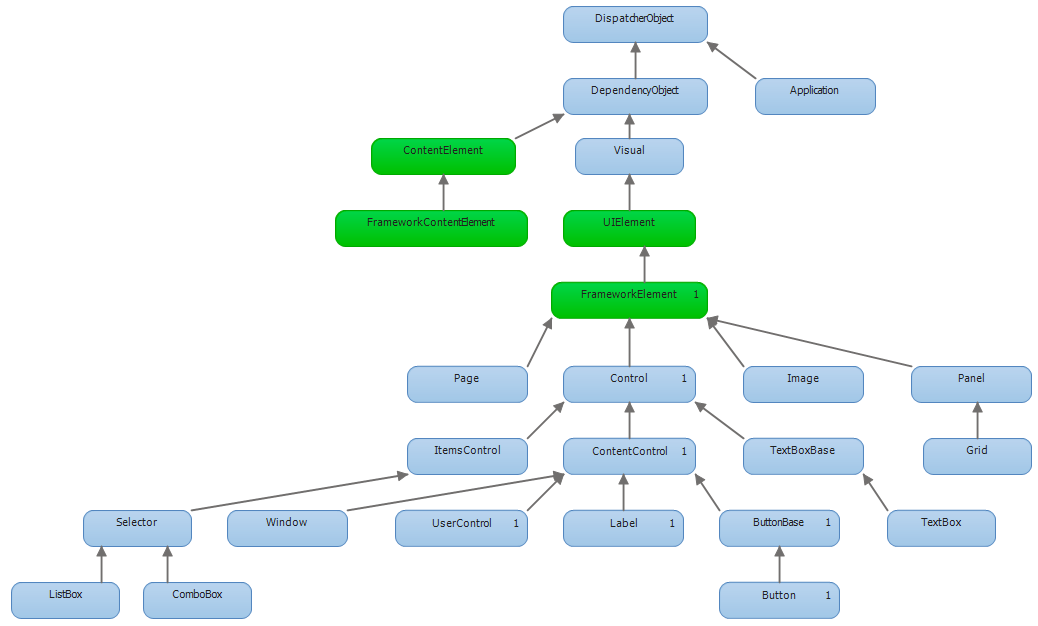
Visual Tree:



WPF Class Hierarchy

These 12 classes have the following significance:





|  |  |
| --- | --- |
|  | **Object**: The base class for all .NET classes and the only class in the figure that isn’t WPF specific.  **DispatcherObject**: The base class meant 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  **DependencyObject**: The base class for any object that can support dependency properties. |

|  |  |
| --- | --- |
|  | **Freezable:** The base class for objects that can be “frozen” into a read-only state for performance reasons. Freezables, once frozen, can 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. Most Freezables are graphics primitives such as brushes, pens, and geometries or animation classes. |

|  |  |
| --- | --- |
|  | **Visual**: The base class for all objects that have their own 2D visual representation.  **UIElement**:  The UIElement class inherits from Visual and adds support for basic user interaction behavior, including:   * **L**ayout behavior   + Parent/child relationship   + Measure/Arrange passes * Responding to user **I**nput * Input events from devices like keyboard/mouse * Command bindings * Manage **F**ocus * Raise (and respond to) routed **E**vents   + Events bubble (up) or tunnel (down) element tree   Note the acronym formed, which helps in thinking about UIElement – “LIFE begins at UIElement“.  **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.  Almost all the WPF elements that you will work with as you build interfaces derive from the System.Windows.FrameworkElement base class and inherit some common layout related properties. These properties serve to fine-tune the way the element is positioned within its parent.  VerticalAlignment, HorizontalAlignment, and Margin are layout affecting properties on all FrameworkElements. They also have a number of width and height related properties that indirectly affect layout.   * Additional input elements (e.g. tooltips, context menus) * Storyboards * Data binding * Styles * Property value inheritance * Support for the logical tree   **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, as well as the ability to be completely restyled. |

|  |  |
| --- | --- |
|  | **Visual3D**:The base class for all objects that have their own 3D visual representation.  **UIElement3D**:The base class for all 3D visual objects with support for routed events, command binding, and focus. |

|  |  |
| --- | --- |
|  | **ContentElement**:  A base class similar to UIElement but for document-related 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. Each ContentElement often requires multiple Visuals to render correctly (spanning lines, columns, and pages).  ContentElement adds (to DependencyObject)   * **I**nput events and commanding * **F**ocus * Raise and respond to routed **e**vents * Animation support   **FrameworkContentElement**:  FrameworkContentElement adds (to ContentElement)   * Additional input elements (e.g. tooltips, context menus) * Storyboards * Data binding * Styles * Property value inheritance   You can think of an FCE as having everything an FE has except support for layout/rendering. |

Dependency Property

* It is a property that is backed by the WPF property system
* The purpose of dependency properties is to provide a way to compute the value of a property based on the value of other inputs. These other inputs might include system properties such as themes and user preference, just-in-time property determination mechanisms such as data binding and animations/storyboards.
* In order to have a dependency property – it needs to be derived from the class DependencyObject. In deriving from this class, you get all the infrastructure needed to participate in the WPF dependency property system.

Additional Features of Dependency Property over CLR Properties:

1. Change Notification
2. Property Value Inheritance
3. Support for multiple providers

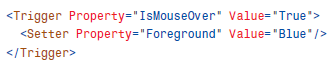
*So one of the first things I thought was weird about the definition of a dependency property is that it is a static. This didn't make sense to me at first - this property needs to store info relevant to a particular instance of a class, how is it going to do that if it is static? We are essentially saying that class A will have a property B - and it makes sense that that definition would be static. The actual storage of a value for a dependency property is deep inside the WPF property system - we never have to worry about it.*

Change Notification

Whenever a value of dependency property changes, WPF can automatically trigger a number of actions, depending on the property’s metadata. These actions be as follows:

* Re-rendering the appropriate elements
* Updating the current layout
* Refreshing data bindings etc

One of the interesting features enabled by this built-in change notification is property triggers – which enable to perform custom actions when a property value changes, without writing any procedural code.



Property Value Inheritance

This refers to flowing of property values down the element tree. Property value inheritance enables child elements in a tree of elements to obtain the value of a particular property from parent elements, inheriting that value as it was set anywhere in the nearest parent element.

An example is setting font size and font style on the window and that is automatically applied to child elements.

|  |  |
| --- | --- |
|  |  |

When does property value inheritance not work :

* Not every dependency property participates in property value inheritance. (Internally, dependency properties can opt in to inheritance by passing FrameworkPropertyMetadataOptions.Inherits to DependencyProperty.Register.)
* There may be other higher-priority sources setting the property value, as explained in the next section.  
  Example: Few controls such as StatusBar, Menu and Tooltip internally set their font properties to match current system settings so users get familiar experience.  
  The result can be confusing, however, because such controls end up “swallowing” any inheritance from proceeding further down the element tree. For example, if you add a Button as a logical child of the StatusBar in Listing 3.4, its FontSize and FontStyle would be the default values of 12 and Normal, respectively, unlike the other Buttons outside of the StatusBar.

Support for Multiple Providers

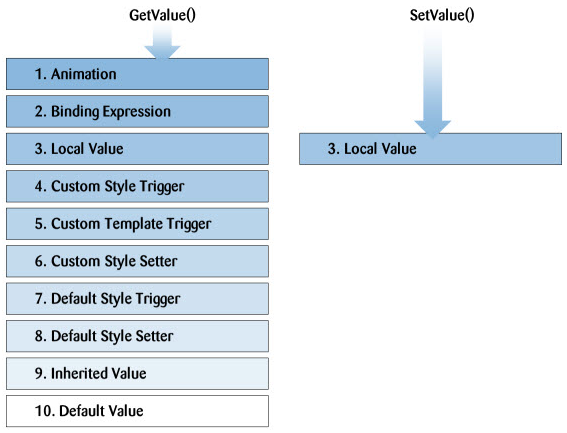
WPF contains many powerful mechanisms that independently attempt to set the value of dependency properties.

This illustrates the five-step process that WPF runs each dependency property through in order to calculate its final value. This process happens automatically, thanks to the built-in change notification in dependency properties.



Determine the Base Value (Value Resolution Strategy)

Every time you access a dependency property, it internally resolves the value by following the precedence from high to low. It checks if a local value is available, if not if a custom style trigger is active,... and continues until it founds a value. At last the default value is always available.

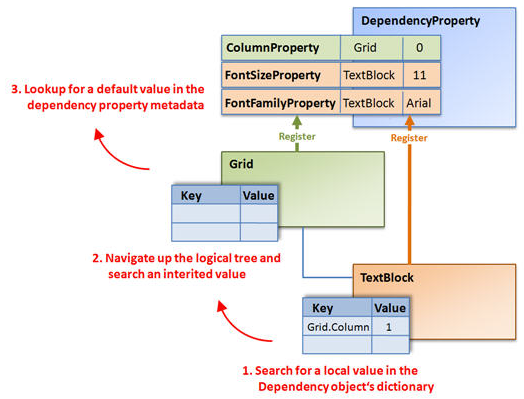


The Magic Behind Dependency Properties

Each WPF control registers a set of DependencyProperties to the static DependencyProperty class. Each of them consists of a key - that must be unique per type - and a metadata that contain callbacks and a default value.

All types that want to use DependencyProperties must derive from DependencyObject. This baseclass defines a key, value dictionary that contains local values of dependency properties. The key of an entry is the key defined with the dependency property.

When you access a dependency property over its .NET property wrapper, it internally calls GetValue(DependencyProperty) to access the value. This method resolves the value by using a value resolution strategy that is explained above. If a local value is available, it reads it directly from the dictionary. If no value is set if goes up the logical tree and searches for an inherited value. If no value is found it takes the default value defined in the property metadata. This sequence is a bit simplified, but it shows the main concept.



How to create a DependencyProperty

In practice, dependency properties are just normal .NET properties hooked into some extra WPF infra.

The steps for standard Dependency Property Implementation are:

1. Declare the dependency property (static readonly)
2. Register the property
3. A. Net property wrapper (optional)
4. A property changed callback (optional)

Tips

* Visual Studio has a snippet called propdp that automatically expands into a definition of a dependency property, which makes defining one much faster than doing all the typing yourself!
* .NET property wrappers are bypassed at runtime when setting dependency properties in XAML!

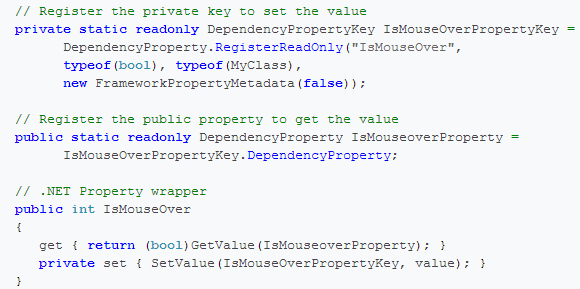
Although the XAML compiler depends on the property wrapper at compile time, WPF calls the underlying GetValue and SetValue methods directly at runtime! Therefore, to maintain parity between setting a property in XAML and procedural code, it’s crucial that property wrappers not contain any logic in addition to the GetValue/SetValue calls. If you want to add custom logic, that’s what the registered callbacks are for. All of WPF’s built-in property wrappers abide by this rule, so this warning is for anyone writing a custom class with its own dependency properties.

ReadOnly DependencyProperty

Some dependency property of WPF controls are readonly. They are often used to report the state of a control, like the IsMouseOver property. Is does not make sense to provide a setter for this value.

Maybe you ask yourself, why not just use a normal .NET property? One important reason is that you cannot set triggers on normal .NET propeties.

Creating a read only property is similar to creating a regular DependencyProperty. Instead of calling DependencyProperty.Register() you call DependencyProperty.RegisterReadonly().



For more detailed reading -   
<http://wpftutorial.net/DependencyProperties.html>  
<http://msdn.microsoft.com/en-us/library/ms752914.aspx>

How to find from where is a control’s dependency property value being set

1. By using tools such as Snoop, WPF inspector
2. By using the GetValueSource method –

If you can’t figure out where a given dependency property is getting its current value, you can use the static DependencyPropertyHelper.GetValueSource method as a debugging aid.

Attached Properties

An attached property is a special form of dependency property, that can effectively be attached to arbitrary objects. The difference is that an attached property applies to an element other than the one where it’s defined. Attached properties are simply a way of connecting additional information to an element. This data can then be used by another source.

The most common example of attached properties is found in the layout containers. For example, the Grid class defines the attached properties Row and Column, which you set on the contained elements to indicate where they should be positioned. Similarly, the DockPanel defines the attached property Dock, and the Canvas defines the attached properties Left, Right, Top, and Bottom.

Another Example is FocusManger.FocussedElement. See the code snippet below

|  |  |
| --- | --- |
|  |  |

This is an attached property that lets the developer tell WPF which control in the Window should receive keyboard focus when first displayed.

Observable Collection

Definition: Represents a dynamic data collection that provides notifications when items get added, removed, or when the whole list is refreshed

Inner Workings:

* The System.Collections.ObjectModel.ObservableCollection(Of T) class inherits from Collection(Of T), which is the base class for generic collections, and implements both the INotifyCollectionChanged and INotifyPropertyChanged interfaces.
* It's important to note that although the ObservableCollection class broadcasts information about changes to its elements, it doesn't know or care about changes to the properties of its elements. In other words, it doesn't watch for property change notification on the items within its collection.
* So if you create a Observable collection of customer object which has first name and last name as its properties, then if they change behind the scenes – the observable collection wouldn’t know about it and reflect it on UI. You need to have to implement INotifyPropertyChanged event implemented at each one of those properties (at a Model level) for them to reflect in the UI
* ObservableCollection class also raises the CollectionChanged event when you add, remove, move, refresh, or replace an item in the collection. This feature makes it possible to react when code outside your window modifies the underlying data.

Additional Points on the Examples I Practiced:

* Observable Collection does not need to be static. The reason you would make it static is to have it available to multiple view models (Same observable collection) and also its thread safe.

Have both examples of static and non-static:

* + For Static – *See example F:\04 - Net Technical Material\!Visual Studio 2010 Projects\02-WPF\aMine\Thread\_Chart\_Application\Thread\_Chart\_Application\_Edit1\Thread\_Chart\_Application*
  + For non-static – *See example F:\04 - Net Technical Material\!Visual Studio 2010 Projects\02-WPF\aMine\OCC\_New\_App*

Layout  
Layout Panels and Decorators: Panels are a special family of classes having the distinguishing capability of being able to arrange controls on the screen.  
Another family of classes, related to panels, derives from the Decorator base class and fulfills this common need. The most common Decorator is Border, which draws a border around its enclosed control.

**Difference between Panel and Decorator**A Panel has a collection of Children that it arranges according to various rules, based on the type of panel. A Decorator, on the other hand, has only one Child to which it applies some additional set of behavior.

Panels

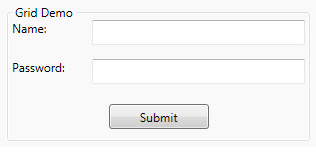
In order to arrange controls, you need to use a layout panel. WPF ships with the following layout panels:

* StackPanel - for stacking elements horizontally or vertically
* DockPanel - for familiar docking functionality
* WrapPanel - automatically handles wrapping elements to a new row as needed
* Grid - for a row and column based layout
* UniformGrid - a specialized form of Grid where all cells are the same size
* Canvas - for specific (X,Y) positioning

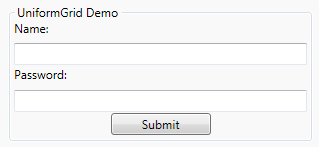
|  |  |  |
| --- | --- | --- |
|  |  |  |

StackPanel is the simplest and one of the most useful layout controls for WPF. By default, it organizes its child elements by stacking them one on top of the other, like a list. StackPanel also has an Orientation property that can be set to Horizontal, causing the panel to stack its children from left to right.  
*Interestingly, StackPanel is often used internally as the default layout for a number of other WPF controls. One example of such a control is ListBox.*  
DockPanelcontrol is capable of attaching its children to any of its four sides and is often used as the root layout control of an application’s UI.  
WrapPanel is used less often than StackPanel, DockPanel, and Grid, but offers useful functionality nonetheless. Essentially, a WrapPanel is like a StackPanel, but it has the ability to “wrap” what it is stacking to the next line or column if it runs out of space.

Gridis WPF’s all-purpose layout panel. It can achieve most of the same behavior as the previous controls and much more. This power comes with a price; the requirement of additional XAML and more attached properties.  
In many cases, using the Grid to accomplish a similar design as another panel will result in more verbose markup. Often it is difficult to ascertain the intention of this markup. If you want docking behavior, be explicit and use a DockPanel rather than a Grid. The same goes for using other Panels. Always choose the simplest and most explicit means that will accomplish your design.



UniformGridis a limited version of the grid where all rows and columns are the same size and where a cell can only hold one control (determined by the grid). Because it's unnecessary to declare each row and column, the UniformGrid contains two properties, Rows and Columns, for setting the number of rows and columns. Controls are added to the grid in the order that they are declared.  
The UniformGrid is useful for simple scenarios, but for the most amount of control, you are better off using the Grid.



GridSplitter is a special control capable of letting a user resize rows or columns of a Grid at runtime. You must place the control within a Grid, between the rows or columns you want to be resizable. Use the ResizeDirection property to indicate what the control will resize (rows or columns) and use ResizeBehavior to declare how the splitter will specifically interact with its own row/column as well as those around it.  
A preferred practice is to place the GridSplitter in a row/column by itself and set the ResizeBehavior to PreviousAndNext. This makes the GridSplitter easier to manage and understand.

WPF Content Control

Content Model - <http://msdn.microsoft.com/en-us/library/bb613548.aspx>

<http://www.wpftutorials.com/2011/03/content-controls.html>

<http://www.c-sharpcorner.com/UploadFile/cook451/XMLDataBinding04102007062916AM/XMLDataBinding.aspx>

Sample code - G:\04 - Net Technical Material\!Visual Studio 2010 Projects\02-WPF\Extract

1. ConfigurationDialogExample
2. MvvmDemoApp

Canvas (for precision layout)Canvas is different from all the Panels we have discussed so far. This difference lies in the fact that Canvas does not add any special dynamic layout behavior to its child controls. A canvas must have an exact Width and Height, and all its child elements must have an exact size and position as well. Canvas arranges controls at strict Left (x) and Top (y) positions using attached properties.  
Avoid using Canvas for most Control layout scenarios. Using exact positioning and sizing undercuts the power of WPF’s dynamic layout mechanisms. The Canvas was originally designed for the layout of Drawings and not Controls. It is best to stick to this practice when possible.

|  |  |
| --- | --- |
|  |  |

ScrollViewerScrollViewer is not a panel but a content control

|  |  |
| --- | --- |
|  |  |

So in the above example – the image things it has all the space it needs by wrapping it in the scrollviewer makes it oblivious of the space allocated. So nothing special has to be done except wrapping it in the scroll viewer.

Decorators

Decorators add graphical decoration or behavior to other elements. A Decorator always has one Child that it decorates (although this child can be a Panel containing many other Controls). Two of the most common Decorators that you will use or encounter in WPF programming: Border and Viewbox. Each of these decorators has a unique capability controlled by a specific set of properties.

ViewboxUnlike Scrollviewer – viewbox scales the content to fit into the space allocated. The bitmap images generally don’t need a viewbox since the Image element already has size specifications but Viewbox can do it for any content – like button

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Layout recommendations**Following is a list of recommendations to help you in your interface design:  
- Begin by using the simplest and most explicit Panel.  
- Do not be afraid to combine multiple Panels to achieve the effect you desire.  
- Pay close attention to the runtime behavior of your layout. You may need to change your strategy to accommodate window resizing.  
- Try to choose layout options that allow for flexible sizing. Avoid setting hardcoded Height and Width properties when possible. Instead, if necessary, consider using MinHeight, MinWidth, MaxHeight, and MaxWidth. These properties give WPF’s layout engine some flexible parameters by which it can work, rather than force it into a brittle layout strategy.  
- Use Canvas only as a last resort. This panel was designed primarily for rendering Drawings, not UI. Using Canvas for ordinary layout scenarios can defeat the purpose of the WPF dynamic layout capabilities.

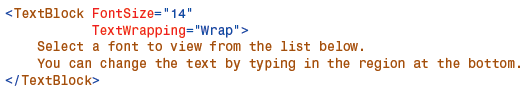
**Navigation**

Need to fill this section

**Controls**

* Most of the elements inherit from the common base class Control.
* This base derives from FrameworkElement and adds a large amount of additional and important functionality.
* Among the notable features are basic properties defining the control’s colors, border, font, and accessibility characteristics. All controls also have some additional mouse events and enable templating.

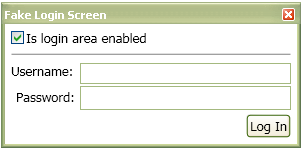
**TextBlock –**



The TextWrapping property is what enables this dynamic reflowing of the text. Removing it will cause the text to be truncated when resized. Also notice how the FontSize property affects the contained text. The TextBlock is quite versatile in its ability to display rich flowing text.

|  |  |
| --- | --- |
| **Inline Elements Used in a TextBlock** | |
| Span | Groups Inline elements together |
| Bold | Bolds the text |
| Italic | Italicizes the text |
| Underline | Underlines the text. |
| Hyperlink | Creates a web-style hyperlink in the text. |
| Run | A sequence of text |
| LineBreak | Forces a new line in the text |
| InlineUIContainer | Allows UIElements to exist in the text flow |
| Figure | Allows text placement to vary from the main flow |
| Floater | Displays content parallel to the main flow |

**Differences between Label and TextBlock**Base Example: We will refer to a demo application which looks like this when you first run it. The “Username:” text is a TextBlock and the “Password:” text is a Label.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr..** | | **TextBlock** | | **Label** |
|  | | **Advantages of Label** | | |
| 1 | |  | | |
|  | | TextBlock is **not** a Control. Even though TextBlock lives in the System.Windows.Controls namespace, it is not a control. It derives directly from FrameworkElement. | | Label, on the other hand, derives from ContentControl. This means that Label can:  1. Be given a custom control template (via the Template property)  2. Display data other than just a string (via the Content property)  3. Apply a DataTemplate to its content (via the ContentTemplate property)  4. Do whatever else a ContentControl can do that a FrameworkElement cannot. |
| 2 | |  | | Label supports access keys –    This is the standard explanation of why Label exists. You can associate a Label with another control, such as a PasswordBox, and allow the user to type an access key defined by the Label to set focus to the other control. The access key is represented in the UI by drawing a line under the appropriate character. If the user presses the Alt key and then the designated “access character” the target control will be given focus. |
| 3 | |  | | |
|  | | TextBlock does not have this behavior by default. | | Label text is grayed out when disabled –  When a Label’s IsEnabled property returns false its text is “grayed out.” |
| **Sr.** | **TextBlock** | | **Label** | |
|  | **Dis-advantages of Label** | | | |
| 1 |  | | Label is much heavier than TextBlock –  So far we have examined why Label can be considered better than TextBlock, but now its time to discuss the benefits of using a TextBlock instead.  <Label> internally calls Textblock for text rendering. So <Label> is an overkill if you are using this for text rendering.  Label has a higher runtime overhead than TextBlock. Not only does Label inherit from two classes further down the inheritance hierarchy than TextBlock, but its visual tree is much more involved.    The “Username” TextBlock’s visual tree contains no child elements. The Label, however, is much more involved. It has a Border, which contains a ContentPresenter, which hosts an AccessText element, which finally uses a TextBlock to display the text. So it turns out that using a Label is really just an elaborate and customizable way of using a TextBlock. | |

Data Binding  
  
DataBinding Overview: <http://msdn.microsoft.com/en-us/library/ms752347.aspx>

Data binding is a means of connecting the elements of a user interface with the underlying data that the application is interested in. Data binding is declarative rather than imperative. Declarative means that instead of writing a series of commands to be executed, you describe the relation that the data has to the elements in the UI.

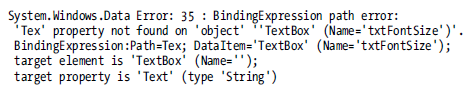
At its simplest, data binding is a relationship that tells WPF to extract some information from a source object and use it to set a property in a target object. The target property is always a dependency property, and it’s usually in a WPF element—after all, the ultimate goal of WPF data binding is to display some information in your user interface. However, the source object can be just about anything, ranging from another WPF element to an ADO.NET data object (like the DataTable and DataRow) or a data-only object of your own creation.

**Binding Elements Together**The simplest data binding scenario occurs when your source object is a WPF element and your source property is a dependency property.

* Built in support for change notification: As a result when you change the value of the dependency property in the source object, the bound property in the target object is updated immediately without any additional infrastructure.  
  *Note: Real world scenario is binding bulk of your data to data objects. However, element-to-element binding is often useful. For example, you can use element-to-element binding to automate the way elements interact so that when a user modifies a control, another element is updated automatically. This is a valuable shortcut that can save you from writing boilerplate code (and it’s a technique that wasn’t possible in the previous generation of Windows Forms applications).*
* Data binding expressions use a XAML markup extension and begin with the word Binding, because you’re creating an instance of the System.Windows.Data.Binding class
* In this situation, you need to set just two properties: the ElementName that indicates the source element and a Path that indicates the property in the source element.  
  *The name Path is used instead of Property because the Path might point to a property of a property (for example, FontFamily.Source) or an indexer used by a property (for example, Content.Children[0]). You can build up a path with multiple periods to dig into a property of a property of a property, and so on.*

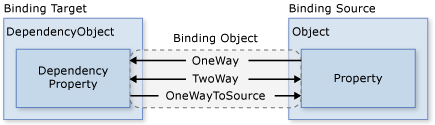
**Binding Errors**

* WPF doesn’t raise exceptions to notify you about data binding problems. If you specify an element or a property that doesn’t exist, you won’t receive any indication; instead, the data will simply fail to appear in the target property.
* WPF does output trace information that details binding failures. This information appears in Visual Studio’s Output window when you’re debugging the application. For example, if you try to bind to a nonexistent property, you’ll see a message like this in the Output window:



**Binding Mode**

WPF allows you to use one of five values from the System.Windows.Data.BindingMode enumeration when setting the Binding.Mode property



The other two are:

* Default: The type of binding depends on the target property. It’s either TwoWay (for user-settable properties, such as the TextBox.Text) or OneWay (for everything else). All bindings use this approach unless you specify otherwise.

*Note: Initially, it seems logical to assume that all bindings are one-way unless you explicitly specify otherwise. For the below example (Example 1 specifically) - if you remove the Mode=TwoWay setting, this example still works just as well.*

* *That’s because WPF uses a different Mode default depending on the property you’re binding. (Technically, there’s a bit of metadata on every dependency property—the FrameworkPropertyMetadata.BindsTwoWayByDefault flag—that indicates whether that property should use one-way or two-way binding.)*
* *Often, the default is exactly what you want. However, you can imagine an example with a read-only text box that the user can’t change. In this case, you can reduce the overhead slightly by setting the mode to use one-way binding.*
* *General Rule of Thumb - It’s never a bad idea to explicitly set the mode. Even in the case of a text box, it’s worth emphasizing that you want a two-way binding by including the Mode property.*
* OneTime: The target property is set initially based on the source property value. However, changes are ignored from that point onward (unless the binding is set to a completely different object or you call BindingExpression.UpdateTarget(). Usually, you’ll use this mode to reduce overhead if you know the source property won’t change.

*Note: You might wonder why there’s both a OneWay and a OneWayToSource option—after all, both values create a one-way binding that works in the same way. The only difference is where the binding expression is placed. Essentially, OneWayToSource allows you to flip the source and target by placing the expression in what would ordinarily be considered the binding source. The most common reason to use this trick is to set a property that isn’t a dependency property. As you learned at the beginning of this chapter, binding expressions can be used only to set dependency properties. But by using OneWayToSource, you can overcome this limitation, provided the property that’s supplying the value is itself a dependency property.*

**Create Bindings with Code**

When you’re building a window, it’s usually most efficient to declare your binding expression in the XAML markup using the Binding markup extension. However, it’s also possible to create a binding using code.

Specialized scenarios where you would use code to create binding:

* Dynamic Binding: If you want to tailor a binding based on other runtime information or create a different binding depending on the circumstances
* Removing a Binding: If you want to remove a binding so that you can set a property in the usual way, you need the help of the ClearBinding() or ClearAllBindings() method.
* Creating Custom Controls: To make it easier for other people to modify the visual appearance of a custom control you build, you’ll need to move certain details (such as event handlers and data binding expressions) into your code and out of your markup.

You can also remove a binding with code using two static methods of the BindingOperations class. The ClearBinding() method takes a reference to the dependency property that has the binding you want to remove, while ClearAllBindings() removes all the data binding for an element:  
BindingOperations.ClearAllBindings(lblSampleText);  
Both ClearBinding() and ClearAllBindings() use the ClearValue() method that every element inherits from the based DependencyObject class. ClearValue() simply removes a property’s local value (which, in this case, is a data binding expression).

*Note You can remove any binding using the ClearBinding() and ClearAllBindings() methods. It doesn’t matter whether the binding was applied programmatically or in XAML markup.*

|  |  |  |
| --- | --- | --- |
| **TextBlock** | **Slider (Source Element)** | **TextBox** |
| **Example 1**  One--Way Binding on all controls.  Need to tab out of Textbox for the size to take effect | | |
|  |  |  |
| **Example 2**  Two-Way Binding from TextBlock to Slider since TextBox is linked to the slider through this  Need to tab out of Textbox for the size to take effect | | |
|  |  |  |
| **Example 3**  Two-Way Binding from TextBox to Slider since TextBlock is linked to the slider through this  Changes in the Textbox take immediately for the size | | |
|  |  |  |

*Note: Setting TickFrequency property to 1 and setting the IsSnapToTickEnabled property to True doesn’t show any numbers with decimal points in the textbox.*

**Binding Cheat Sheet**

****

**Binding Updates**

Changes that flow in reverse direction (from target to source) don’t necessarily happen immediately. Instead, their behavior is governed by the Binding.UpdateSourceTrigger property, which takes one of the values listed. In example 1 & 2 above you’re witnessing an example of a target-to-source update that uses the UpdateSourceTrigger.LostFocus behavior  
*Note: The default behavior of the TextBox.Text property is LostFocus, simply because the text in a text box will change repeatedly as the user types, causing multiple refreshes. Depending on how the source control updates itself, the PropertyChanged update mode can make the application feel more sluggish. Additionally, it might cause the source object to refresh itself before an edit is complete, which can cause problems for validation.*

|  |  |
| --- | --- |
| **Name** | **Description** |
| PropertyChanged | The source is updated immediately when the target property changes |
| LostFocus | The source is updated when the target property changes and the target loses focus |
| Explicit | The source is not updated unless you call the BindingExpression.UpdateSource() method. |
| Default | The updating behavior is determined by the metadata of the target property (technically, its FrameworkPropertyMetadata.DefaultUpdateSourceTrigger property). *For most properties, the default behavior is PropertyChanged, although the TextBox.Text property has a default behavior of LostFocus.* |

For complete control over when the source object is updated, you can choose the UpdateSourceTrigger.Explicit mode. If you use this approach in the text box example, nothing happens when the text box loses focus. Instead, it’s up to your code to manually trigger the update. For example, you could add an Apply button that calls the BindingExpression.UpdateSource() method, triggering an immediate refresh and updating the font size.

**Binding to Objects That aren’t Elements**

So far, we focused on adding bindings that link two elements. But in data-driven applications, it’s more common to create binding expressions that draw their data from a nonvisual object. The only requirement is that the information you want to display must be stored in public properties.  
When binding to an object that isn’t an element, you need to give up the Binding.ElementName property and use one of the following properties instead:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Source | This is a reference that points to the source object—in other words, the object that’s supplying the data. |
| RelativeSource | This points to the source object using a RelativeSource object, which allows you to base your reference on the current element. This is a specialized tool that’s handy when writing control templates and data templates |
| DataContext | If you don’t specify a source using the Source or RelativeSource property, WPF searches up the element tree, starting at the current element. It examines the DataContext property of each element and uses the first one that isn’t null. The DataContext property is extremely useful if you need to bind several properties of the same object to different elements, because you can set the DataContext property of a higher-level container object, rather than setting it directly on the target element. |

**Source**: The Source property is quite straightforward. The only catch is that you need to have your data object handy in order to bind it. As you’ll see, you can use several approaches for getting the data object: pull it out of a resource, generate it programmatically, or get it with the help of a data provider.  
Simplest Option:   
Point the Source to some static object that’s readily available. For example, you could create a static object in your code and use that. Or, you could use an ingredient from the .NET class library, as shown here:



This binding expression gets the FontFamily object that’s provided by the static SystemFonts.IconFontFamily property. (Notice that you need the help of the static markup extension to set the Binding.Source property.) It then sets the Binding.Path property to the FontFamily.Source property, which gives the name of the font family. The result is a single line of text. In Windows Vista or Windows 7, the font name Segoe UI appears.  
Another Option:   
Bind to an object that you’ve previously created as a resource. For example, this markup creates a FontFamily object that points to the Calibri font:  
  
Here is TextBlock that binds to this resource:   


**RelativeSource**

The RelativeSource property allows you to point to a source object based on its relation to the target object. For example, you can use RelativeSource property to bind an element to itself or to bind to a parent element that’s found an unknown number of steps up the element tree.  
The RelativeSource object uses the FindAncestor mode, which tells it to search up the element tree until it finds the type of element defined by the AncestorType property.  
  
The FindAncestor mode is only one of four options when you create a RelativeSource object. The all 4 modes are as follows:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Self | The expression binds to another property in the same element |
| FindAncestor | The expression binds to a parent element. WPF will search up the element tree until it finds the parent you want. To specify the parent, you must also set the AncestorType property to indicate the type of parent element you want to find. |
| PreviousData | The expression binds to the previous data item in a data-bound list. You would use this in a list item. |
| TemplatedParent | The expression binds to the element on which the template is applied. This mode works only if your binding is located inside a control template or data template. |

*At first glance, the RelativeSource property seems like a way to unnecessarily complicate your markup. After all, why not bind directly to the source you want using the Source or ElementName property? However, this isn’t always possible, usually because the source and target objects are in different chunks of markup. This happens when you’re creating control templates and data templates.*

**DataContext**

In some cases, you’ll have a number of elements that bind to the same object. When the source information is missing from a binding expression, WPF checks the DataContext property of that element. If it’s null, WPF searches up the element tree looking for the first data context that isn’t null. (Initially, the DataContext property of all elements is null.) If it finds a data context, it uses that for the binding. If it doesn’t, the binding expression doesn’t apply any value to the target property.

**Resources**

WPF provides a resource system for sharing objects. Every framework-level element has a Resources property.

Each resource in a resource dictionary must have a unique key. When you define resources in markup, you assign the unique key through the x:Key Directive. Typically, the key is a string; however, you can also set it to other object types by using the appropriate markup extensions. Nonstring keys for resources are used by certain feature areas in WPF, notably for styles, component resources, and data styling.

When you define a resource you always need to have a key:

<Page.Resources>  
 <SolidColorBrush x:Key=”MyGreen” Color=”#34b654” />  
<Page.Resources>

……………………

<GeometryDrawing Brush=”{StaticResource MyGreen}”>  
……………………

Resources Property: All below offer:

* FrameworkElement
* FrameworkContentElement
* Application
* Style
* FrameworkTemplate

Looks like any dictionary object. It ImplementsIDictionary.

XAML offers 2 markup extensions to access a resource:

* StaticResource
* DynamicResource

StaticResource is a one-time look-up and more light-weight than the other

DynamicResource is more like data binding and it remembers that a property is associated with a resourcekey and if the object associated with that key is changed DynamicResource will update the target property.

For either resource the target property which is using the resource gets reference of the underlying resource and doest make a copy of the original resource. If multiple properties use a resource then each one gets the reference of the same object. WPF does not make a copy of resource for each use.

That’s why even if WPF/XAML lets you declare the below in Resource Tags – you should **not** do that:

<Button x:Key=”btn” />

<picture here>

**Binding Options**

Validation Options - <http://updatecontrols.net/doc/>

Converter - <http://www.rhyous.com/2011/02/22/binding-visibility-to-a-bool-value-in-wpf/>

Styles, Templates, Skins And Themes

Arguably the most celebrated feature in WPF is the ability to give any user interface element a radically different look without having to give up all of the built-in functionality that it provides.

This chapter explains the four main components of WPF’s restyling support:

**Styles** - A simple mechanism for separating property values from user interface elements (similar to the relationship between CSS and HTML). Styles are 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.

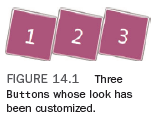
**Skins** - Application-specific collections of styles and/or templates, typically with the ability to be replaced dynamically.

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

Styles

A style, represented by the System.Windows.Style class, is a pretty simple entity. Its main function is to group together property values that could otherwise be set individually. The intent is to then share this group of values among multiple elements.

*See example C:\Visual Studio 2010 Projects\WPF\Basics\* *Styles\_Templates\_Skins\_Themes\_Examples*

**

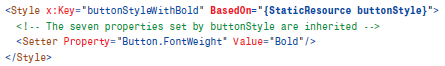
But with a Style, you can add a level of indirection—setting the properties in one place and pointing each Button to this new element, as shown in code example above. Style uses a collection of Setters to set the target properties. Creating a Setter is just a matter of specifying the name of a dependency property (qualified with its class name) and a desired value for it.

Using a Style is nice for several reasons, such as having only one spot to change if you later have second thoughts about rotating the Buttons or if you want to change their Background. Defining a Style as a resource also gives you all the flexibility that the resource mechanism provides. For example, you could define one version of buttonStyle at the application level but override it with a different Style (still with a key of buttonStyle) in an individual Window’s Resources collection.

Indeed, Style only enables the setting of dependency properties, which tend to be visual in nature.

Tip 1:

Styles can even inherit from one another! The following Style adds bold text to the buttonStyle defined in Listing 14.2 by using the BasedOn property:



Tip 2:

Any individual element can override aspects of its Style by directly setting a property to a local value. For example, the Button in Listing 14.3 could do the following to retain the rotation, size, and so on from controlStyle yet have a red Background rather than a blue one:

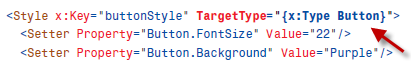


This works because of the order of precedence for dependency property values. The local value trumps anything set from a Style.

Restricting Use of Styles

This can be done by applying the target type.

Feature 1: If you want to enforce that a Style can be applied only to a particular type, you can set its TargetType property accordingly. For example, the following Style can be applied only to a Button (or a subclass of Button):



Any attempt to apply this Style to a non-Button generates a compile-time error.

Feature 2: In addition, when you apply a TargetType to a Style, you no longer need to prefix the property names inside Setters with the type name. So, the previous XAML snippet could be rewritten as follows and have exactly the same meaning:



Feature 3:

If you omit its Key, the Style gets implicitly applied to all elements of that target type within the same scope. This is typically called a typed style as opposed to a named style, which is the only kind of Style you’ve seen so far.

The scope of a typed Style is determined by the location of the Style resource. For example, it could implicitly apply to all relevant elements in a Window if it’s a member of Window.Resources. Or, it could apply to an entire application if you define it as an application- level resource, as follows:



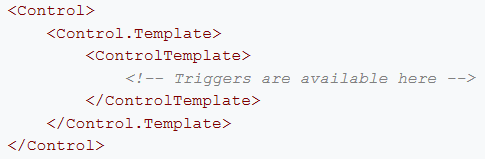


In such an application, all Buttons get this style by default. But each Button can still override its appearance by explicitly setting a different Style or explicitly setting individual properties. Any Button can restore its default Style by setting its Style property to null.

Triggers

Trick to use triggers everywhere

Triggers are only available in styles and templates. Even if every control has a property Triggers, it only works with EventTriggers. If you want to use normal triggers outside a style or a template, you can use a little trick. Just wrap your code with the following snippet:



Pointers

Difference Between Margin and Padding  
  
Margin and Padding are two similar layout concepts that are often confused.   
Margin, present on all FrameworkElements, represents the amount of space around the outside of the element. This space ensures that the FrameworkElement has room between it and neighboring elements.   
Padding functions differently. It is present on elements that inherit from Control (itself derived from FrameworkElement) and allows the control to specify an amount of space inside between the border of the element and the content it contains. This inner space separates the control from its own content. A Button control illustrates this most clearly. Picture the space inside the Button, around its text, that prevents the Button’s border from shrinking to the size of its contents.

|  |  |
| --- | --- |
|  |  |

Padding is only offered for controls which have a content property. For example Rectangle doesn’t have child elements but Button does have.

**ToolTip**

|  |  |
| --- | --- |
| ToolTip is a property on everything that inherits from FrameworkElement. Use this property to set pop-up text that appears when users place their mouse over an element.  Manually control the tab order of controls - Everything that inherits from Control has a TabIndex property. Set this property on each Control with a number indicating the order in which the controls should be focused.  Include formatted text or a graphic into ToolTip - ToolTip is a ContentControl so you can assign anything to its Content property. Try using this markup in the Font Viewer: |  |

**Access Keys**Press the Alt key on the keyboard and you will notice that certain letters in the Label text become underlined. These are called access keys. Access keys allow you to hit Alt + SomeKey to quickly interact with a control in the UI, such as hitting Alt + O to click an “OK” button.  
Pressing Alt in combination with the underlined letter will direct keyboard input to the corresponding TextBox. This makes it easy for users to quickly navigate between noncontiguous fields using only the keyboard. To define an access key, simply precede the character with an underscore. Setting the Target property of the Label tells WPF what control should be accessed when the key is pressed.

|  |  |
| --- | --- |
|  |  |

**MessageBox.Show -**From time to time you need an application to display simple messages or gather yes/no responses from a user. The MessageBox class is the ideal solution for this problem. It is a static class with several method overloads called Show. As you might expect, calling any of these methods results in the appearance of a special kind of modal dialog box. Modal dialog boxes block interaction with other windows in the application, forcing the user to pay them attention until they are closed. Each of the overloads of Show let you choose different options affecting how it will be displayed. Some of these features include the capability to specify caption/message text, icons, and types of buttons to show in the dialog.

**Following During Data-Binding** – If you have 2 controls which are data-bound to the same property of the view Model (example username) in the code below – then you can do following (second control looking upto the first one) to get its value rather than both of them binding to username directly



The following below will have the same effect

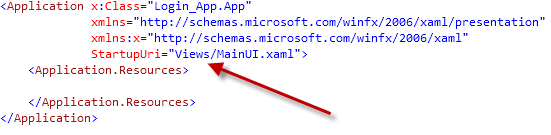


Using the UpdateSourceTrigger immediately refreshes the underlying object.

Benefit – If later you decide to change the view property from userName to FirstuserName – then instead of changing in XAML at multiple places – you need to only change at once.

**File Organization While Programming**

If you move the MainWindow.xaml to a different folder – then in the App.xaml you need to do the following:



Concepts to Write

Pack URI

<http://msdn.microsoft.com/en-us/library/aa970069.aspx>

<http://www.wiredprairie.us/journal/2007/06/pack_syntax_in_wpf.html>

<http://stackoverflow.com/questions/3747972/wpf-usercontrol-cannot-find-xaml-resource-in-referencing-project>

|  |  |
| --- | --- |
| Resources | |
|  | There are two types of resources in WPF :  1) Binary Resources  2) Logical Resources  **Binary Resources**  Out of the various build actions defined by WPF only following two are important for binary resources :    Resource: Embeds resource into the assembly (or culture specific satellite assembly)  Content: this leaves resource as loose file and upon compilation this resource information is not embedded to assembly. Instead, it adds custom attribute to the assembly which records the existence and relative location of file.  URIs for accessing the binary resources   |  |  | | --- | --- | | **URI** |  | | “Logo.jpg” | embedded in current assembly (using “Resource” as build action)  loose resource however physical file is placed along with xaml file or assembly (if selected build action is “Content”) | | “A/B/Logo.jpg” | same as above but resource file is located in subfolder “A/B” | | “D:\Resource\Logo.jpg” | resource path hard coded | | <http://pinvoke.net/logo.jpg> | loose resource hosted at pinvoke.net website | | ResourceDll;component/Logo.jpg | this URI represents the resource embedded in assembly ResourceDLL.dll or ResourceDLL.exe | | ResourceDll;componen/tA/B/Logo.jpg | same as above except resource is located in subfolder | | pack://siteOfOrigin:,,,/logo.jpg | resource located at site of origin | | pack://siteOfOrigin:,,,/A/B/logo.jpg | resource located at site of origin in sub folder |   Site of Origin :  Site of origin at the runtime gets resolved in different ways depending upon the way in which the application has been deployed.   * For a full-trust application installed with Windows Installer, the site of origin is the application’s root folder. * For a full-trust ClickOnce application, the site of origin is the URL or Universal Naming Convention (UNC) path from which the application was deployed.   **Logical Resources**  Logical resources are arbitrary .net objects stored (and named) in element’s ResourceDictionary. Logical resource can be shared among multiple elements depending upon the scope at which it has been defined. Typical logical resources are styles, bindings, etc.  Logical resources are of two types :  1) StaticResources  2) DynamicResources  StaticResource/DynamicResource markup extension accepts a key name as parameter. So this key is searched while looking up for resource. It is not mandatory that resource has to be defined on the element which is using it. Xaml looks for the resource at the element level first then to parent level and then to application level. Resource markup extension implements the ability to walk the logical tree to find the items upward in the logical tree so if resource is not found, it checks the parent element, its parent and so on until it reaches the root element. At this point, it checks the resources collection at the application level and system level subsequently. If the resource is not found anywhere, it throws “InvalidOperationException”.  Although it is advisable to have unique key associated with resource, however it is not mandatory. While looking up for the resource whatever resource is found first is assigned. |
|  | Merged Dictionaries  Resource Dictionaries are a powerful feature in WPF and Silverlight that enable developers to organize and consume reusable resources.  What is a resource: A resource is an object declared in XAML. One reason to use resources in your applications is to promote object reuse across your application. Object reuse provides consistency across the application. Object reuse also makes it very easy to change the application; since you only need to change the resources and all consumers will pick up the change.  Where Can I Locate Resources?: Resources are located in resource dictionaries. When you add resources to the Resources property, you are actually adding them to a ResourceDictionary exposed by the object. The Resources property is defined on FrameworkElement and is inherited from any object that has FrameworkElement in its inheritance hierarchy. This means that resources can be located on Windows, UserControls, Grids, Buttons, TextBoxes, ListBoxes, etc.  In addition to an objects Resources property, resources can also be located in a ResourceDictionary XAML file.    Show me how a ResourceDictionary Used:  Resource dictionaries are used to group related resources, then those resource dictionaries are then merged at the Application or other required scope. |

To Practice

|  |  |  |
| --- | --- | --- |
| 1 | ContentPresenter/ContentControl POC |  |
| 2 | Sort Collection in Listbox | <http://zamjad.wordpress.com/2010/01/24/sorting-items-in-a-list-box/> |
| 3 | Custom Dependency Property | <http://wpftutorial.net/DependencyProperties.html> |
| 4 | How to find from where is a control’s dependency property value being set | <http://wpf.2000things.com/2010/12/05/146-use-getvaluesource-method-to-find-the-source-of-a-dependency-property-value/> |
| 5 | DataGrid Links to Practice | <http://www.codeproject.com/Articles/442498/Multi-filtered-WPF-DataGrid-with-MVVM>  <http://www.zagstudio.com/blog/411#.UHcd1IrAf6E>  <http://www.dotnetfunda.com/articles/article1000-simple-grid-implementation-with-icollectionview-in-wpf.aspx> |
| 6 | View Talking to ViewModel through Interface | <http://www.wpfsharp.com/2012/07/09/creating-an-interface-for-data-binding-in-views-with-wpf/>  <http://www.wpfsharp.com/category/c-c-sharp/> |
| 7 | MultiBinding/MultiValueConverter | <http://zamjad.wordpress.com/2010/01/15/multi-biniding-and-multi-value-converter/>  <http://miteshsureja.blogspot.com/2011/07/multi-value-converters-in-wpf.html> |
| 8 | Custom Control POC | <http://wpftutorial.net/HowToCreateACustomControl.html> |
| 9 | INotifyCollectionChanged Example (which observable collection implements inherently) |  |
| 10 | Progress Bar XAML based | <http://learnwpf.com/post/2011/10/31/Pure-XAML-pre-loaderindeterminate-progress-indicator-for-WPF.aspx>  <http://www.wpfsharp.com/2011/05/11/wpf-replacement-options-for-an-animated-gif/> |
| 11 | Practice HierarchicalDataTemplate example |  |
| 12 | Custom Routed Commands | <http://www.c-sharpcorner.com/uploadfile/dpatra/custom-routedui-commands-in-wpf/> |
| 13 | WPF ItemsControl Example | <http://rachel53461.wordpress.com/2011/09/17/wpf-itemscontrol-example/> |
| 14 | Design Time Data | <http://stackoverflow.com/questions/1889966/what-approaches-are-available-to-dummy-design-time-data-in-wpf> |
|  |  |  |
| 1 | MVVM Example | <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx> |
| 2 | Attached Behavior Pattern Example | <http://www.codeproject.com/Articles/28959/Introduction-to-Attached-Behaviors-in-WPF> |
| 3 | Mediator Pattern Example | Explanation - <http://joshsmithonwpf.wordpress.com/2009/04/06/a-mediator-prototype-for-wpf-apps/>  <http://mvvmfoundation.codeplex.com/>  <http://www.codeproject.com/Articles/35277/MVVM-Mediator-Pattern>  <http://dotnet.dzone.com/articles/mvvm-light-whats-messenger> |
| 4 | Prism Examples | <http://stackoverflow.com/questions/1097582/wpf-silverlight-prism-resources-for-beginners?rq=1>  <http://www.codeproject.com/Articles/165376/A-Prism-4-Application-Checklist>  MS Prism Book  <http://www.codeproject.com/Articles/48287/Getting-Started-with-Prism-2-1-for-WPF> |
|  |  |  |

Amazing Website Links

Debugging WPF –

1. <http://www.wpftutorial.net/DebugDataBinding.html>
2. <http://blogs.msdn.com/b/mikehillberg/archive/2006/09/14/wpftracesources.aspx>
3. <http://blogs.msdn.com/b/jgoldb/archive/2010/05/25/wpf-visualizer-in-visual-studio-2010.aspx>
4. <http://msdn.microsoft.com/en-us/library/dd409789.aspx>
5. <http://pelebyte.net/blog/2011/07/11/twelve-ways-to-improve-wpf-performance/>
6. <http://blogs.msdn.com/b/wpfsldesigner/archive/2010/06/30/debugging-data-bindings-in-a-wpf-or-silverlight-application.aspx>

Property Value Inheritance - <http://msdn.microsoft.com/en-us/library/ms753197.aspx>

Dependency Injection versus Messenger (Mediator Pattern) - <http://stackoverflow.com/questions/4207569/mvvm-communication-between-the-model-and-viewmodels?rq=1>

Easing Functions - <http://blogs.msdn.com/b/maximelamure/archive/2009/11/10/add-easing-effects-on-your-animations-with-wpf-4-easing-functions.aspx>

Overall WPF Links

<http://wpfcontrols.blogspot.com/>  
<http://msdn.microsoft.com/en-us/library/ms754130.aspx>  
<http://learnwpf.com/>  
<http://miteshsureja.blogspot.com/p/wpf-tutorial.html>  
http://www.codeproject.com/Articles/23301/WPF-A-Beginner-s-guide-Part-3-of-n  
WPF 4.5 Features - <http://msdn.microsoft.com/en-US/library/vstudio/bb613588>