WPF Tutorial : Beginning

By **Abhishek Sur** | 28 Dec 2010 | Article

C# .NET Dev WPF Intermediate

Licence	CPOL
First Posted	**28 Dec 2010**
Views	**86,223**
Downloads	**3,067**
Bookmarked	**190 times**

The article will guide you through the basics of WPF programing with in-depth knowledge about the architecture and the working principles of WPF programs. The article finally creates a sample "Hello World" application to step you into a new foundation.

4.77 (93 votes)

Download source - 24.97 KB

Table of Contents

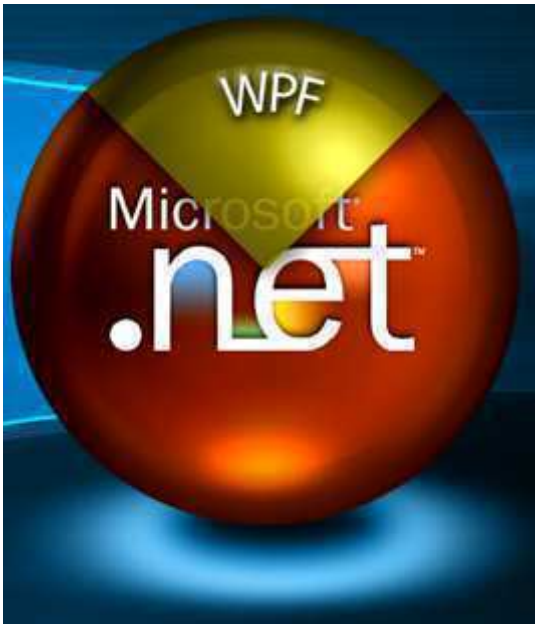
Introduction

After working for more than 6 months in WPF, it is time to start writing on the basics of WPF. I have already put up a few articles on this topic, which basically deal with some specific problems. Now it is time to go beyond that, and to let you understand how / why WPF came to us as a revolution to UI development.

As this is an article for beginners to intermediate level programmers, I will try to give as many basic examples as I can. I will also put a few reference links to each article of the series to make you navigate the other articles easily.

- WPF Tutorial : Beginning [^]
- WPF Tutorial : Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial : Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutorial - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

Windows Presentation Foundation



As the name says all, WPF is actually a new framework introduced with .NET framework 3.0 which actually puts forward a new set of classes and assemblies which allow us to write programs more efficiently and flexibly. It uses Direct3D rendering which employs graphics cards to render the output on the screen. Thus the drawing in the form will be smooth and also there is a chance to utilize the hardware capabilities installed in your machine. In case of traditional GDI forms application, it is not possible to use advanced graphics capabilities and hence Windows Forms application will always be inefficient in comparison to WPF. Another important thing that I must address in this regard, GDI Windows forms application uses Operating system controls to build its application. Thus it is basically very hard to customize them in your own application. WPF controls are actually drawn over the screen, and hence you can customize controls totally and modify their behavior when required.

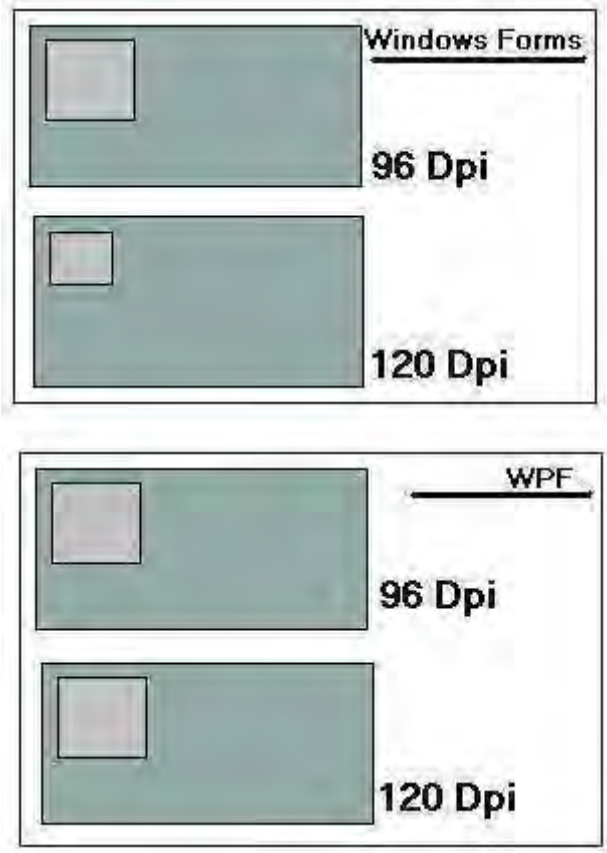
Features of WPF

WPF comes with lots of advantages. Let me introduce a few of its features:

- **Device Independent Pixel (DPI)**

  WPF introduces Device Independent DPI Settings for the applications built with it. For a window, it is very important to calculate how many Dots Per inch(DPI) the screen could draw. This is generally dependent on the hardware device and operating system in which the application runs and also how the DPI settings is applied on the Device. Any user can easily customize these settings and hence make the application look horrible. Windows forms application uses pixel based approach so with changing DPI settings, each control will change its size and look.

  WPF addresses this issue and makes it independent of DPI settings of the computer. Let's look at how it is possible:



Let's say you have drawn a box, just like the one in the figure, which is 1 inch long in 96 dpi screen. Now if you see the same application in 120 dpi settings, the box will appear smaller. This is because the things that we see on the screen are totally dependent on dpi settings.

In case of WPF, this is modified to density based approach. That means when the density of pixel is modified, the elements will adjust them accordingly and hence the pixel of WPF application is Device Independent Pixel. As you can see in the figure, the size of the control remains the same in case of WPF application and it takes more pixels in case of 120 DPI application to adjust the size properly.

- **Built-In Support for Graphics and Animation :**

  WPF applications as being rendered within DirectX environment, it has major support of graphics and animation capabilities. A separate set of classes are there which specifically deal with animation effects and graphics. The graphics that you draw over the screen is also Vector based and are object oriented. That means, when you draw a rectangle in WPF application, you can easily remove that from the screen as rectangle is actually an object which you always have a hold on. In a traditional Windows based application, once you draw a rectangle, you can't select that individually. Thus programming approach in case of WPF is completely different and more sophisticated than traditional graphics approach. We will discuss graphics and animation in more detail in a later section of the article.

- **Redefine Styles and Control Templates**

  In addition to graphics and animation capabilities, WPF also comes with a huge flexibility to define the styles and `ControlTemplates`. Style based technique as you might come across with CSS is a set of definitions which defines how the controls will look like when it is rendered on the screen. In case of traditional windows applications, styles are tightly coupled with each control, so that you need to define color, style, etc. for each individual control to make it look different. In case of WPF, Styles are completely separated from the `UIElement`. Once you define a style, you can change the look and feel of any control by just putting the style on the element.

  Most of the `UIElements` that we generally deal with are actually made using more than one individual elements. WPF introduces a new concept of `Templates`, which you might use to redefine the whole control itself. Say for instance, you have a `CheckBox`, which has a `Rectangle` in it and

a `ContentPresenter` (one where the caption of the `TextBox` appears). Thus you can redefine your `checkbox` and put a `ToggleButton` inside it, so that the check will appear on the `ToggleButton` rather than on the `Rectangle`. This is very interesting. We will look into more detail on `Styles` and `ControlTemplates` in later section of the article.

- **Resource based Approach for every control**

  Another important feature of WPF is Resource based approach. In case of traditional windows applications, defining styles is very hectic. So if you have 1000 buttons, and you want to apply Color to each `Buttons` to Gold, you need to create 1000 objects of Color and assign each to one individual elements. Thus it makes it very resource hungry.

  In WPF, you can store styles, controls, animations, and even any object as resource. Thus each resource will be declared once when the form loads itself, and you may associate them to the controls. You can maintain a full hierarchy of styles in a separate file called *`ResourceDictionary`*, from which styles for the whole application will be applied. Thus WPF application could be themed very easily.

- **New Property System & Binding Capabilities**

  In the next step, I must introduce the new property system introduced with WPF. Every element of WPF defines a large number of dependency properties. The dependency properties have strong capabilities than the normal properties. Thus when I define our new property, we can easily register my own property to any object I wish to. It will add up to the same observer that is associated to every object. As every element is derived from `DependencyObject` in its object hierarchy, each of them contains the Dependency Observer. Once you register a variable as Dependency property, it will create a room on the observer associated with that control and set the value there. We will discuss this in more detail in later sections of the series.

What is XAML?

According to the definition, XAML is an XML based declarative markup language for specifying and setting the characteristics of classes. In other words, XAML is a language used by WPF, Silverlight or any other application which can declare classes by itself. So, you can declare a variable, define the properties of any class and directly use it in your application. The XAML parser will automatically parse and create the actual object while it renders the application.

XAML is generally used to define layout of UI, its elements and objects for static and visual aspect. We cannot define flow of a program using XAML. So even though there are large capabilities of XAML, it is actually not a programming language, rather it is used to design UI of the application. Thus XAML employs other programming languages like C#, VB.NET, etc. to define the logic in code behind.

`ExpressionBuilder` is the best tool to generate XAML.

WPF Architecture

For every new technology, it is very essential to have a clear idea about its architecture. So before beginning your application, you must grab a few concepts. If you would not like to know WPF in detail, please skip this section. As mentioned earlier, WPF is actually a set of assemblies that build up the entire framework. These assemblies can be categorized as:

- Managed Layer
- UnManaged Layer
- Core API

Managed Layer: Managed layer of WPF is built using a number of assemblies. These assemblies build up the WPF framework, communicate with lower level unmanaged API to render its content. The few assemblies that comprise the WPF framework are:

1. ***PresentationFramework.dll***: Creates the top level elements like layout panels, controls, windows, styles, etc.
2. ***PresentationCore.dll***: It holds base types such as `UIElement`, Visual from which all shapes and

controls are Derived in *PresentationFramework.dll*.

3. ***WindowsBase.dll***: They hold even more basic elements which are capable of being used outside the WPF environment like `Dispatcher` object, `Dependency` Objects. I will discuss each of them later.

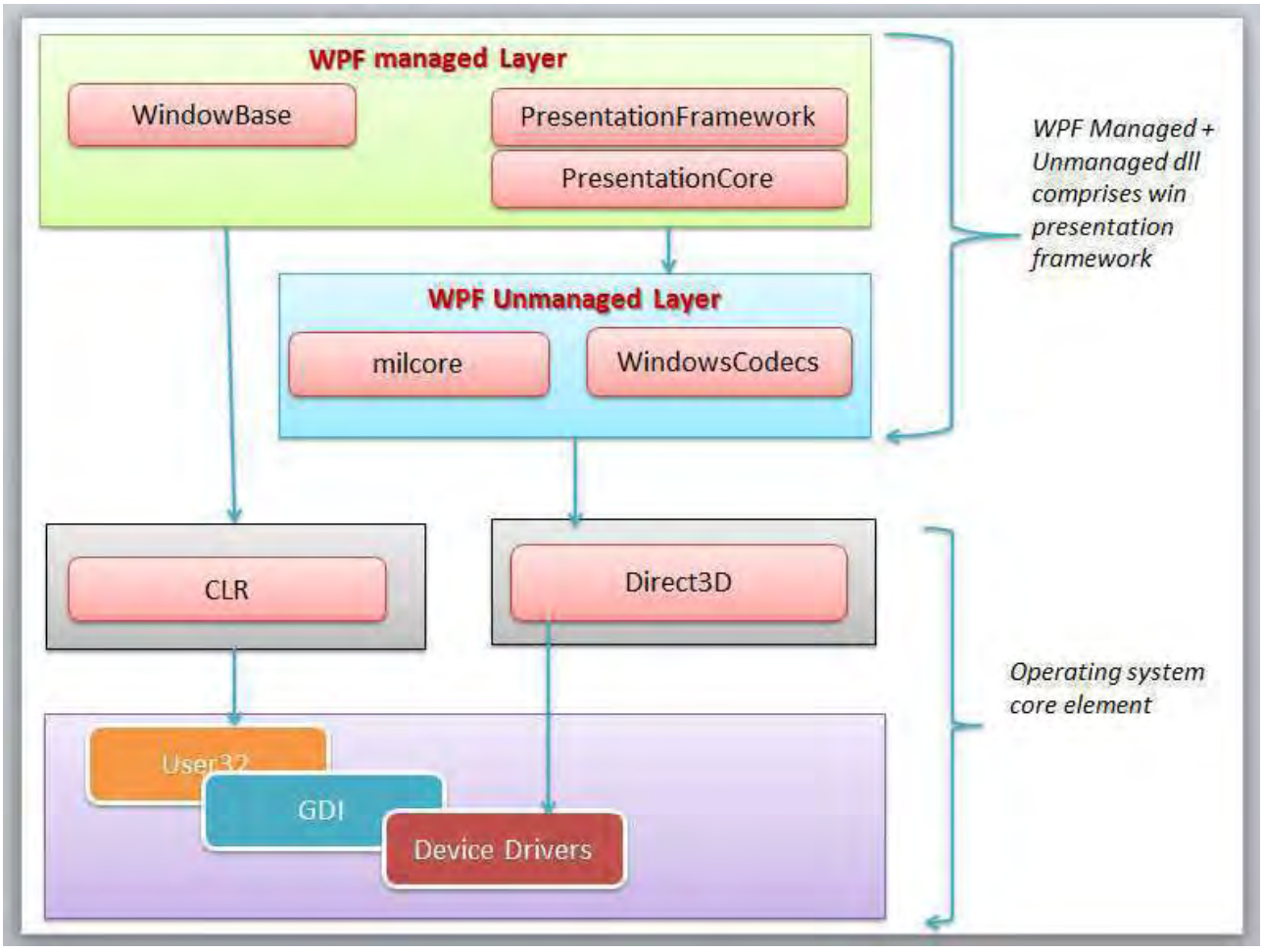
Unmanaged Layer (*milcore.dll*): The unmanaged layer of WPF is called milcore or Media Integration Library Core. It basically translates the WPF higher level objects like layout panels, buttons, animation, etc. into textures that `Direct3D` expects. It is the main rendering engine of WPF.

WindowsCodecs.dll: This is another low level API which is used for imaging support in WPF applications. *WindowsCodecs.dll* comprises a number of codecs which encode / decode images into vector graphics that would be rendered into WPF screen.

Direct3D: It is the low level API in which the graphics of WPF is rendered.

User32: It is the primary core API which every program uses. It actually manages memory and process separation.

GDI & Device Drivers: GDI and Device Drivers are specific to the operating system which is also used from the application to access low level APIs.



In the above figure, you can see how different framework elements communicate between one another that I have just discussed.

Few Things to Know Before You Proceed

There are quite a few things that you must know before starting with WPF applications.

What is Meant by Dispatcher & Thread Affinity?

When WPF application starts, it actually creates two threads automatically. One is Rendering Thread, which is hidden from the programmer, so you cannot use the rendering thread directly from your program;

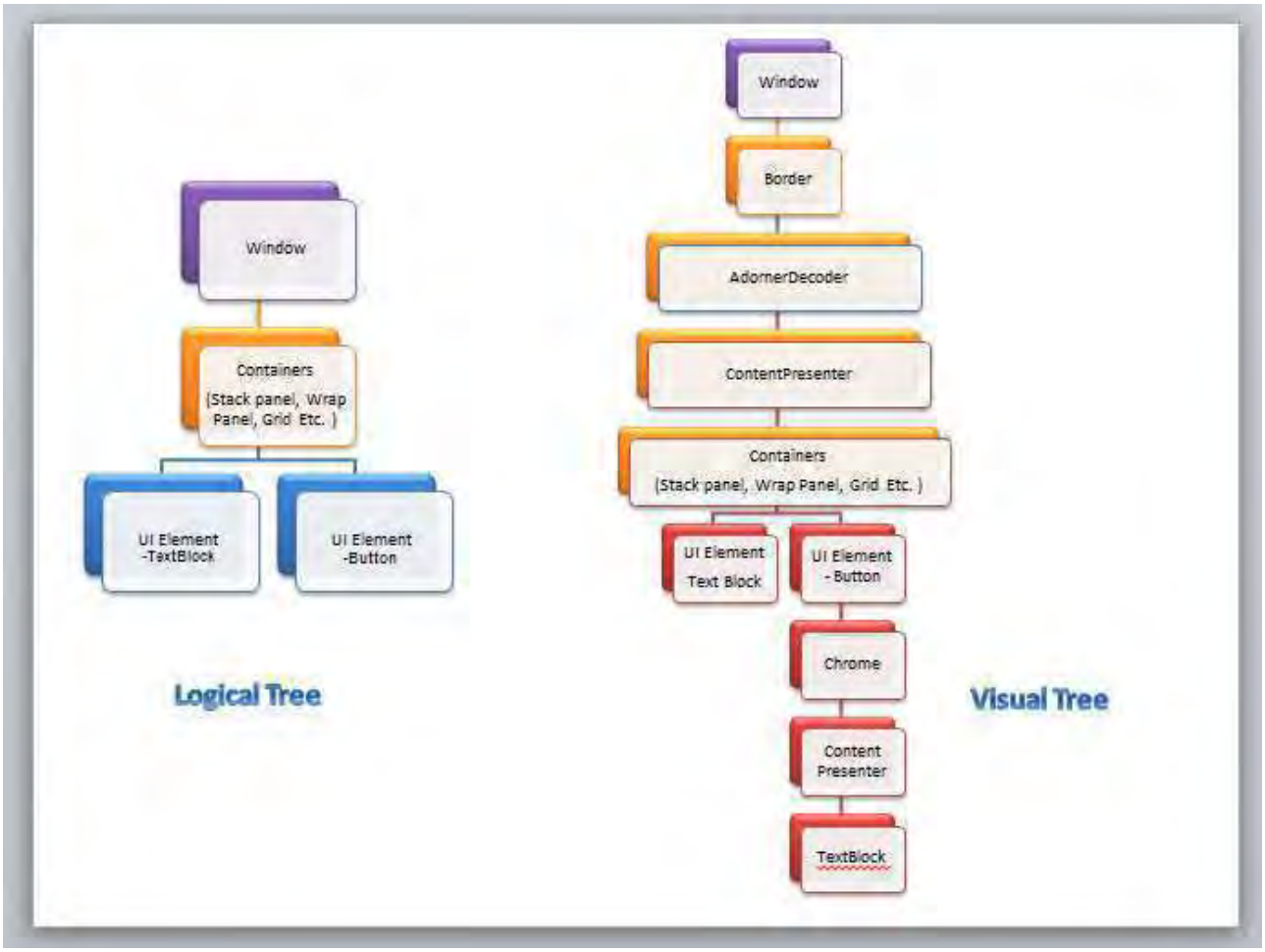
while the other is Dispatcher Thread, which actually holds all the UI elements. So in other words, you might say Dispatcher is actually the UI thread which ties all the elements created within the WPF application. Conversely, WPF requires all the UI elements to be tied with Dispatcher thread, this is called Thread Affinity. Thus you cannot change any element created on `Dispatcher` thread from any other threads, thus it follows the same Win32 based APIs. Thus it allows you to inter-operate any WPF component into `HWND` based API. For more, read this. [^]

`Dispatcher` is a class that handles thread affinity. It is actually a prioritized message loop through which all elements are channeled through. Every `UIElement` is derived from `DispatcherObject` which defines a property called `Dispatcher` which points to the UI thread. Thus from any other thread, if you want to invoke or access UI component, you need to Invoke using `Dispatcher` thread. `DispatcherObject` actually has two chief duties, to check and verify if the thread has access to the object.

What is Visual Tree and Logical Tree?

Every programming style contains some sort of `LogicalTree` which comprises the Overall Program. The `LogicalTree` comprises the elements as they are listed in XAML. Thus they will only include the controls that you have declared in you XAML.

`VisualTree` on the other hand, comprises the parts that make up the individual controls. You do not generally need to deal with `VisualTree` directly, but you should know how each control is comprised of, so it would be easier to build custom templates using this.



I personally always like to see the `VisualTree` before using it. `ExpressionBuilder` is the one tool that allows you to generate the actual control.

Why RoutedEvent?

`RoutedEvent` is very new to the C# language, but for those who are coming from JavaScript/web tech, you would have found it in your browser. Actually there are two types of `RoutedEvent`. One which

Bubbles through the Visual tree elements and another which Tunnels through the visual tree elements. There is also Direct `RoutedEvent` which does not Bubble or Tunnel.

When a `Routed` event which is registered, is invoked, it Bubbles / Tunnels through the Visual Elements and calls all the Registered `RoutedEventHandlers` associated within the Visual Tree one by one.

To discriminate between the two, WPF demarcated events with Preview*** as the events which are Tunneled and just *** for the events that Bubbles. For instance, `IsPreviewMouseDown` is the event that tunnels through the Visual Child elements while `MouseDown` Bubbles. Thus Mouse Down of the `Outermost` element is called first in case of `IsPreviewMouseDown` while Mouse Down for the innermost element will be called first in case of `MouseDown` event.

Why DependencyObject is Used?

Every WPF control is derived from `DependencyObject`. `DependencyObject` is a class that supports `DependencyProperty`, a property system that is newly built in WPF. Every object is derived from `DependencyObject` and hence it can associate itself in various inbuilt features of WPF like `EventTriggers`, `PropertyBindings`, `Animations`, etc.

Every `DependencyObject` actually has an `Observer` or a `List` and declares 3 methods called `ClearValue`, `SetValue` and `GetValue` which are used to add/edit/remove those properties. Thus the `DependencyProperty` will only create itself when you use `SetValue` to store something. Thus, it is resource saving as well. We will look at `DependencyProperty` in detail in other articles of the series.

What about Hardware Acceleration and Graphics Rendering Tiers in WPF?

Another important thing that you should know is how the WPF graphics is rendered. Actually WPF rendering automatically detects how much hardware acceleration is supported by the current system and adjusts itself accordingly. The graphics rendering detects the appropriate tier for rendering the output accordingly.

For hardware rendering, few things that have most of the impact are:

1. **Video RAM**: This determines the size and number of buffers that the application might use to render its output.
2. **Pixel Shader**: It is a graphics utility which calculates effects on per pixel basis.
3. **Vertex Shader**: It is a graphics processing utility that performs mathematical calculations on Vertex of the output. They are used to add special effects to objects in 3D environment.
4. **MultiTexture Blending**: This is a special function that allows you to apply two or more textures on the same object in 3D.

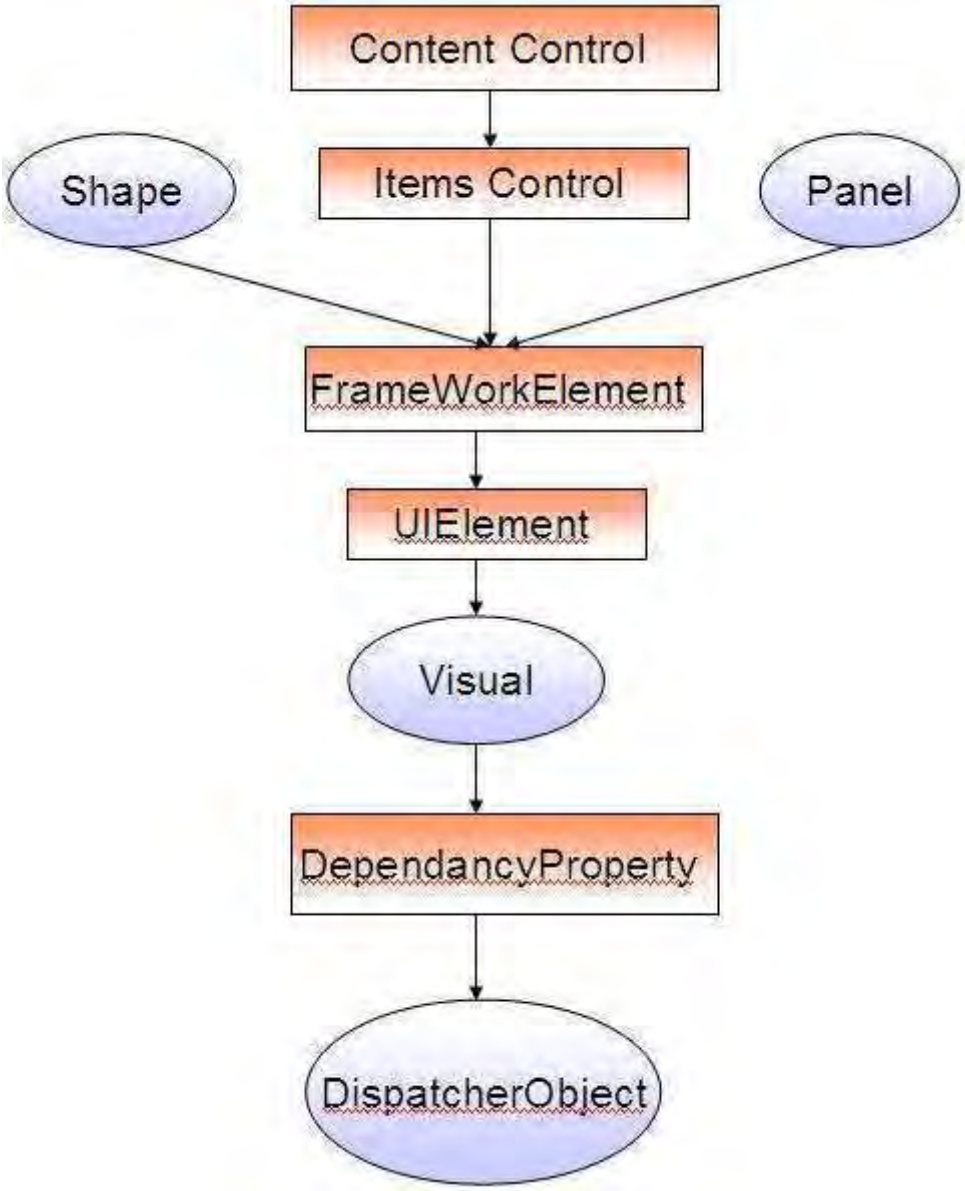
Now the rendering engine of WPF determines which tier is appropriate for the current application and applies the rendering tiers accordingly.

1. **TIER 0**: No graphics hardware acceleration takes place, rather everything is rendered using Software. Any version of DirectX 9 or less is capable of rendering this output.
2. **TIER 1**: Partial hardware and software rendering. You might use Directx9 or greater to use this tier.
3. **TIER 2**: Full hardware acceleration. Directx9 or above can render this output.

To know more about it, refer here [^] .

Object Hierarchy

There are quite a few objects in any WPF control. Let's discuss one by one as in the figure. (The `abstract` class is marked in ellipse while concrete class in Rectangles) :



- `DispatcherObject`: Mother of all WPF controls which takes care of UI thread
- `DependencyObject`: Builds the Observer for Dependency Properties
- `Visual`: Links between managed libraries and milcore
- `UIElement`: Adds supports for WPF features like layout, input, events, etc.
- `FrameworkElement`: Implementation of `UIElement`
- `Shape`: Base class of all the Basic Shapes
- `Control`: The UI elements that interact with the user. They can be Templated to change look.
- `ContentControl`: Baseclass of all controls that have single content
- `ItemsControl`: Baseclass for all controls that show a collection
- `Panel`: Baseclass of all panels which show one or more controls within it

Building Your First WPF Application

Now the time has come to build your first WPF Application. To do this, let's open Visual Studio 2008 / 2010. For this example, I used Visual Studio 2008. Create a new Project. You will see a new window. The XAML will look like:

```
<Window x:Class="FirstWindowsApplication.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Name="Window1"
  Title="Window1" Height="300" Width="300">
   <Grid>
   </Grid>
</Window>
```

Here the blank window is produced. `Height` / `Width` represents the Size of the Window. Title determines the text which is displayed in the `TitleBar` of the window. Every control in XAML could be named with `x:Name` attribute. This will be used to reference the `Window` object in your XAML. `x:Class` attribute represents the class which should be associated with current `Window`. As I already told you, that XAML is not self sufficient, so to define logic you need a class in C# or VB.NET.

`Grid` is the primary layout for WPF application. `Grid` can take multiple `Child` elements. So let us put some controls into the `grid`.

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition MinWidth="50" />
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <TextBlock Text="Enter Name :" Grid.Row="0" Grid.Column="0" />
        <TextBox x:Name="txtName" Grid.Row="0" Grid.Column="1" MinWidth="50"/>
        <Button Content="Click Me" Grid.Row="0" Grid.Column="2" Click="Button_Click"/>
    </Grid>
```

You can see that I have defined `RowDefination` and `ColumnDefination`. This allows you to divide the grid into cells so that you can place your control exactly where you want to. For each `RowDefination` and `ColumnDefination`, you can use `Height` and `Width` of it. You can see I have used `50`, `Auto`, and `*` as `width`. `Auto` represents the size which we would define for the control during control definition. `*` indicates the rest of the space which it can take. Thus, you can see the button is spread the rest of the area of the column it finds.

Now in the Behind, I put a `Messagebox` to show the content of `TextBox`.

```
private void Button_Click(object sender, RoutedEventArgs e)
 {
     MessageBox.Show(string.Format("Hi {0}", this.txtName.Text));
 }
```

Therefore you can see your name will be prompted to you. Isn't it funny. :)

If you have noticed the XAML minutely, you might wonder how I can define the property of `Grid` within other controls. Like I defined `Grid.Row=0` in each of the control. This is actually possible due to the use of Dependency properties. It's a new feature introduced with WPF. We will discuss in detail later on.

Conclusion

This is the first part of the series of WPF. This comprises the initial discussion of the WPF application. In the next articles, I would delve more into other aspects of WPF programs and how you can work with WPF controls and guide you to build a solution. Hope you like reading this post.

License

About the Author

Abhishek Sur

Web Developer
Buildfusion Inc
India

Member

Follow on Twitter

Did you like his post?

Oh, lets go a bit further to know him better.
Visit his Website : www.abhisheksur.com to know more about Abhishek.

Basically he is from India, who loves to explore the .NET world. He loves to code and in his leisure you always find him talking about technical stuffs.

Presently he is working in WPF, a new foundation to UI development, but mostly he likes to work on architecture and business classes. ASP.NET is one of his strength as well.
Have any problem? Write to him in his Forum.

You can also mail him directly to abhi2434@yahoo.com

Want a Coder like him for your project?
Drop him a mail to contact@abhisheksur.com

Visit His Blog

Dotnet Tricks and Tips

Dont forget to vote or share your comments about his Writing

Comments and Discussions

40 messages have been posted for this article Visit **http://www.codeproject.com/Articles/140611/WPF-Tutorial-Beginning** to post and view comments on this article, or click **here** to get a print view with messages.

WPF Tutorial : Layout-Panels-Containers & Layout Transformation

Licence CPOL
First Posted 28 Dec 2010
Views 41,715
Downloads 1,857
Bookmarked 91 times

By **Abhishek Sur** | 22 Aug 2011 | Article

C# .NET Dev WPF Intermediate

This article describes the basics of WPF application, how you can deal with layout, placements of controls and position.

4.92 (77 votes)

Download demo - 48.68 KB

Table of Contents

Introduction

In my previous article, I discussed few basics of WPF applications, its architecture and internal structure to start with WPF. In this article of the series, I am going to discuss about the very basics of writing your first WPF application and how you can place controls in your window. This is very basic to anybody who wants to start with WPF. I will discuss most of them which are commonly used.

- WPF Tutorial: Beginning [^]
- WPF Tutorial: Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial: Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]

- WPF Tutorial - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

A Window

While building your application, the first thing you notice is a Window. `Window` is the main class that interacts with the user and produces the lifetime of windows and dialog boxes. Like in normal windows application, it produces the object windows using the normal API. A window has two sections:

1. **Non-Client Area**: which displays the outer boundary of the window, that we normally see with any windows. The main parts of them are Icon, System Menu, a title Bar and Border.
2. **Client part**: This is the main part where the WPF controls will appear. You can customize this area using WPF.

Type of Window

WPF window is of 3 types:

1. `Window`: This is basically a normal windowed application, where every control is placed within the same window. The window appears normally as I told you earlier. The Client area is fully customizable using XAML.
2. `NavigationWindow`: This is a special type of window which is inherited from Windows, but with a Navigation panel top of it. So if you want to create an application that makes sense when used as Wizards, you might better go with `NavigationWindow`. You can also customize the navigation panel yourself so that it goes with your own look and feel.
3. `Page`: Almost similar to `NavigationWindow`, the main difference is that, `Page` can be opened in Web Browser as XBAP applications.

In the above image, you can see how Normal Window differs from `NavigationWindow`. `NavigationWindow` is very uncommon in general case, but might come in handy when you need special treatment for your application.

Let me discuss a bit on how you can use Pages in your application.

Pages are created to be used as a Page for the Same `Window`. Navigating from one page to another is very simple. `Page` class exposes an object of `NavigationService` which you can use to navigate between pages. `NavigationService` has few events like `Navigating`, `NavigationFailed`, `NavigationProgress`, `NavigationStopped`, etc. which you can use to show progressbar while the page is navigating. Methods like `GoBack`, `GoForward` and `Navigate` are the best way to navigate from one page to another.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

```
    this.NavigationService.Navigate(new Uri("Page2.xaml", UriKind.Relative));
}
```

Thus you can see, rather than calling a new window for *Page2.xaml*, I just used `NavigationService` to navigate from one page to another.

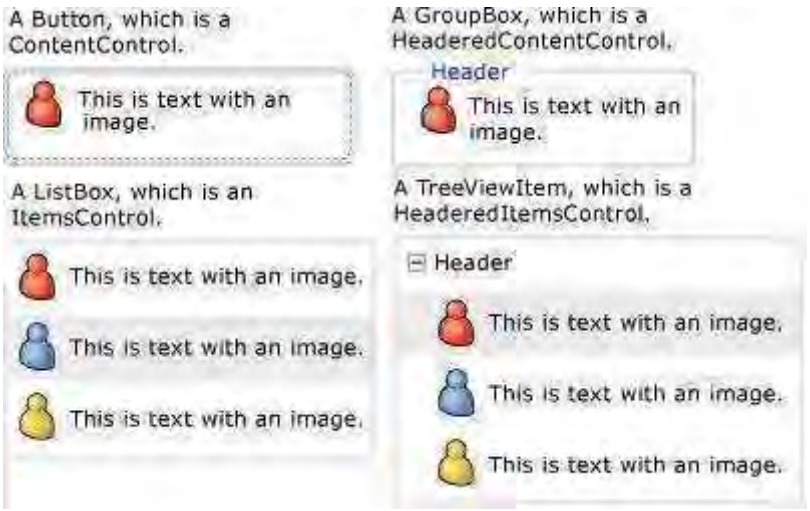
For further reference, you can use MSDN article:

- OverView of Navigation [^]

Types of Containers

WPF Window is derived from `ContentControl`. Basically while working with Controls, you might come across a few types of controls which form the basis for any WPF control. A `ContentControl` holds any arbitrary content. It may be a `string`, an `object` of any type or even a `UIElement` like `Button`, `TextBox`, etc. In other words, A `Content` is an arbitrary element that might be placed inside a container. Let's take a look at them one by one:

1. **ContentControl**: A `ContentControl` holds a single child Content. As Window is derived from `ContentControl`, every window can have only a single Child. For Example: `Window, Button`, etc.
2. **HeaderedContentControl**: It is basically the same as `ContentControl`, but additionally there is a header part which shows the Header for the `Content`. For instance, a `GroupBox, Expander` are `HeaderedContentControl`.
3. **ItemsControl**: The content of `ItemsControl` is multiple. Therefore, you can place many arbitrary elements inside an `ItemsControl`. For instance : `ListBox, ListView` are examples of `ItemsControl`.
4. **HeaderedItemsControl**: Here, each `Collection` has a specific header content of it. A `HeaderedItemsControl` is a complex element which holds each content with a specific header. `TreeView` is an example of `HeaderedItemsControl`.



The above picture shows the distinction between different `ContentControls`. Each `contentControl` contains a `Content` property which stores the inner content. In your XAML, you can specify using `Content` attribute, or you can directly write the `Content` inside the `Tag`. Thus,

```
<Button Content="This is a Button" />
```

is the same as:

```
<Button>This is a Button</Button>
```

XAML parser parses the element written inside the XAML `ContentControl` Tag as `Content`.

Alignment-Margin-Padding

`Alignment`, `Margin` and `padding` are the 3 most important properties that you should always consider for every `UIElement`. Before going further with the containers, you need to know about them.

Alignment: `Alignment` determines how the child elements will be placed within the allocated space of the Parent Element. In other words, it determines the position on the space it was provided. There are two types of `Alignment`:

1. **HorizontalAlignment**: It has 4 possible values - `Left, Right, Center and Stretch. Stretch` is the default value of any `HorizontalAlignment`.
2. **VerticalAlignment**: It has 4 possible Values - `Top, Center, Bottom and Stretch. Stretch` is the default value of any `VerticalAlignment`.

Margin: It determines the distance between one control to the boundary of the cell where it is placed. It can be uniform when specified as a integer value, or you can use `TypeConverter` to specify the value of all its sides. For instance:

`Margin = "20"` means `Left=20`, `Top=20`, `Right=20` and `Bottom=20`.
You can also specify as
`Margin="20,10,0,10"` which means `Left =20`, `Top=10`, `Right=0`, and `Bottom=10`.

```
<Button Margin="0,10,0,10">Button 1</Button>
<Button Margin="0,10,0,10">Button 2</Button>
<Button Margin="0,10,0,10">Button 3</Button>
```

Padding: Padding is present in few of the controls that help in enlarging the size of the control by its value. So it is almost similar, but Margin places space outside the boundary of the control whereas padding places it inside the boundary of the control.

```
<Button Padding="0,10,0,10">Button 1</Button>
<Button Padding="0,10,0,10">Button 2</Button>
<Button Padding="0,10,0,10">Button 3</Button>
```

Each of the `Margin` and `padding` takes an object of `Thickness`.

```
Button bb = new Button();
bb.Margin = new Thickness(20);
bb.Padding = new Thickness(10, 20, 30, 10);
this.MyGrid.Children.Add(bb);
```

Layout Containers

Another important part of any WPF application is to define the Layout of the screen. WPF introduces a number of Panels each are derived from `abstract` class `Panel`. You can also derive `Panel` to define your custom `Panel` if you wish. We will look into how you can define your own `CustomPanel` later. Now let's discuss about all the basic `Panel`s supported by WPF.

PANEL

`Panel` is the `abstract` class from which each and every `panel` is derived from. So each layout element that we will talk about is derived from the `Panel` class and has few properties which I should discuss before talking about `Concrete` objects.

1. `Z-Index`: It determines the index of the `UIElement` which overlapped with another element. `ZIndex` is an Attached property which determines the index in layered elements. One with higher `Zindex` will show above the other.
2. `InternalChildren`: This is the basic `UIElementCollection` element which is exposed using `Children` property. When defining your custom `Panel`, you can use it to get the elements.
3. `Background`: This is also present for any `panel` element, which specifies the `Background` Color for the `Panel`.

Custom Panel

To create a custom `Panel` yourself, you need to override two methods:

MeasureOverride: This method is called whenever an element is added on the `Panel`. It takes the `availableSize` as input and returns the `DesiredSize` for the element passed. You need to calculate the `Size` so that it could be placed accordingly in desired size.

ArrangeOverride: This method is called to determine the arrangement of the `Element`. It will be called once for the whole panel, when `Layout` is created and the final desired size for the panel is returned from it. It will again be called when `Layout` is updated.

You can try the MSDN sample for more details of creating Custom `Panel`:

- http://go.microsoft.com/fwlink/?LinkID=159979 [^]

GRID

`Grid` is the most basic layout panel which forms a graph in the whole frame. `Grid` forms a `Table` which you can address using Row and Column. You can specify the `RowDefination` and `ColumnDefination` for Rows and columns of the `Grid`. You can specify the height of a Row and Width of a Column easily using `RowDefinations` and `ColumnDefinations`.

Sizing of Rows and Columns

As I already said, Height and Width of each Cell in a Grid can be specified using `RowDefinations` and `ColumnDefinations`, the sizing can be specified in more than one way. The Sizing can be:

- **Auto**: This is the default Sizing which is determined by the element you place within the Control.
- *** (Star)**: When you use star, it means the measurement will be done using ratio. 2* means double of

1*. So if you want to make two columns in 2:1 ratio, you mention the width as 2* and 1*.
- **Absolute**: You can also specify the absolute value of the height and width. Means if you specify 100 as height, it will take it accordingly.

From my own experience, it is better practice to use `MinHeight` and `MaxWidth` instead of Width and Height when you want your layout to be strict and doesn't depend on the child elements.

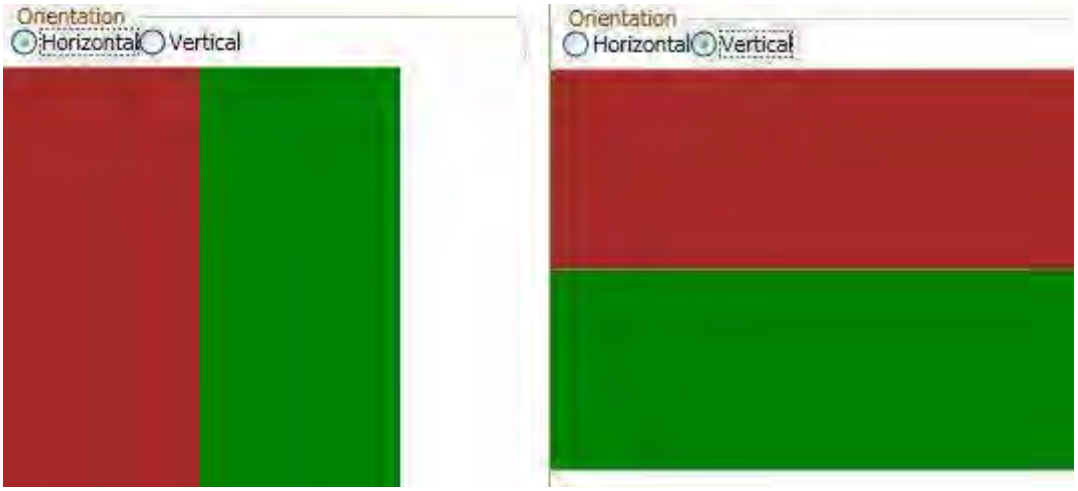
In the sample application, I have created a `Grid` with 3X3 matrix. You can use `TextBoxes` specified on the top to position the Box in `Row` and `Columns` Dynamically.

```
<Grid Grid.Row="1">
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Border Background="BurlyWood" x:Name="brdElement">
                <TextBlock x:Name="tbElement" Text="Row 0, Column 0"
                   HorizontalAlignment="Center" VerticalAlignment="Center"/>
            </Border>
</Grid>
```

This will place the Border in 0,0 column of the 3X3 Table.

STACKPANEL

The very next control that I must start with is a `StackPanel`. `StackPanel` is a container where all the child elements are placed in stacks, that means one after another, so that no one overlaps on one another.



`Stackpanel` places controls based on `PositiveInfinity`, which means the size that it can take in positive direction. The main property of `StackPanel` is its `Orientation`. There are two `Orientations` supported.

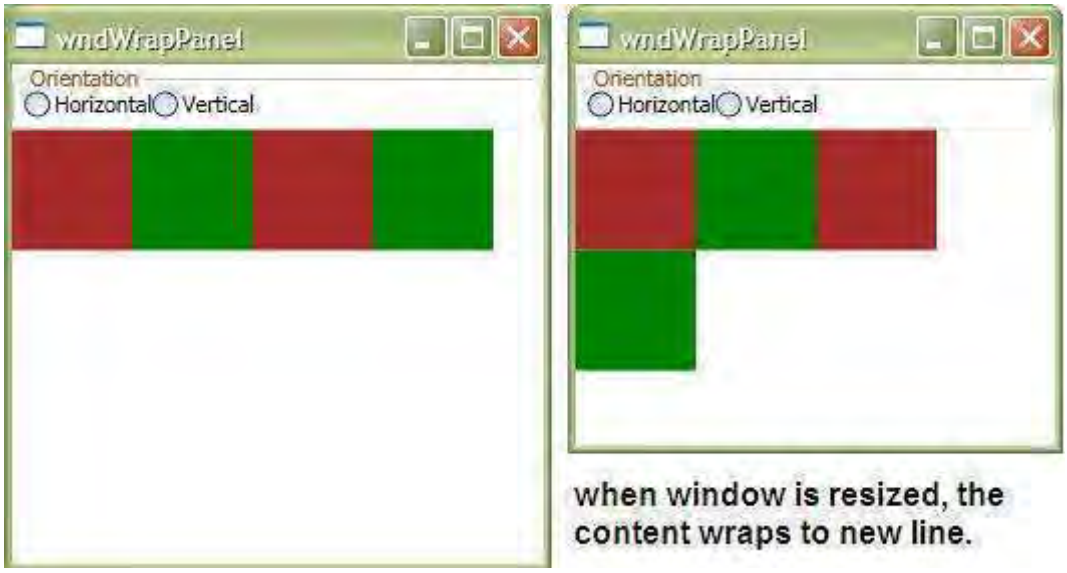
Vertical: This is the default orientation for `StackPanel` where the child items are placed vertically one after another from top to bottom.

Horizontal: Here the items are placed from left to Right one after another.

```
<StackPanel x:Name="spMain" Orientation="Horizontal">
            <Border Background="Brown" Padding="50"></Border>
            <Border Background="Green" Padding="50" />
</StackPanel>
```

WRAPPANEL

`WrapPanel` is almost similar to `StackPanel`, but it produces a `newLine` when it reaches the edge of the panel. Thus `WrapPanel` has additional flexibility to wrap elements when space matters. Another difference is, `WrapPanel` always determines the size based on the size of the content rather than `PositiveInfinity` as of `StackPanel`.



So if you resized the window, the content will be automatically wrapped to the new line. `WrapPanel` also exposes `Orientation` Property as `StackPanel`.

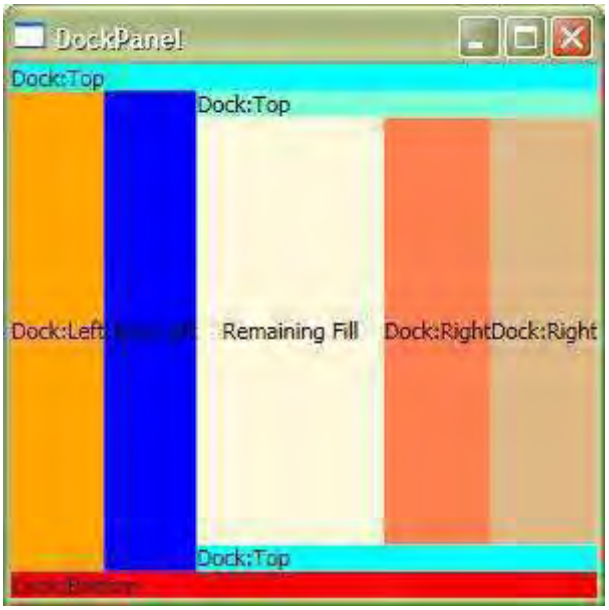
```
<WrapPanel x:Name="wpMain" Grid.Row="1">
            <Border Background="Brown" Padding="30"/>
```

```
            <Border Background="Green" Padding="30" />
            <Border Background="Brown" Padding="30" />
            <Border Background="Green" Padding="30" />
</WrapPanel>
```

DOCKPANEL

`DockPanel` is the most widely used control to determine the layout of an application. It uses `DockPanel.Dock` attached property to determine the position of the element. The `Dock` element when Top or Bottom, will make the element appear Top or Bottom of each other, and when its Left and right, it is left and right of each other.

In case of `DockPanel`, if the height and width of the element placed within it is not specified, it takes the size of the area it is provided with.




```
<DockPanel>
            <Border Background="Aqua" DockPanel.Dock="Top">
                <TextBlock Text="Dock:Top" />
            </Border>
            <Border Background="Red" DockPanel.Dock="Bottom">
                <TextBlock Text="Dock:Bottom" />
            </Border>
            <Border Background="Orange" DockPanel.Dock="Left">
                <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                  Text="Dock:Left" />
            </Border>
            <Border Background="Blue" DockPanel.Dock="Left">
                <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                          Text="Dock:Left" />
            </Border>
            <Border Background="Aqua" DockPanel.Dock="Bottom">
                <TextBlock Text="Dock:Top" />
            </Border>
            <Border Background="Aquamarine" DockPanel.Dock="Top">
                <TextBlock Text="Dock:Top" />
            </Border>
            <Border Background="BurlyWood" DockPanel.Dock="Right">
                <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                  Text="Dock:Right" />
            </Border>
            <Border Background="Coral" DockPanel.Dock="Right">
                <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                     Text="Dock:Right" />
```

```
            </Border>
            <Border Background="Cornsilk" >
                <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                     Text="Remaining Fill" />
            </Border>
</DockPanel>
```

So, you can see that you need to explicitly mention `Dock` property of each individual element to make it appear accordingly. The sequence of declaration also plays a vital role in case of `DockPanel`s. If you mention two elements in a row with `DockPanel.Dock=Top`, that means the two individual elements will appear as Vertically Oriented `StackPanel`.

A property called `LastChildFill` makes the remaining space to be filled with undocked element. You can see in the figure the last element is filled with the entire space left. You can make it `false` if you don't need it.

If you want to do this with code, you need to use `DockPanel.SetDock`.

VirtualizingStackPanel

WPF introduces a special type of panel which `Virtualize`s its content when the content is bound to Data elements. The word virtualize means the content will only be produced when the element is visible to the screen. Thus the performance will be improved a lot.

```
<ListBox x:Name="lstElements" VirtualizingStackPanel.IsVirtualizing="True"
    VirtualizingStackPanel.VirtualizationMode="Recycling" ItemsSource="{Binding}"/>
```

Now from code if you write:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
      ObservableCollection<int> obs = new ObservableCollection<int>();
      Random rnd = new Random(1000);
      for (int i = 0; i < 100000; i++)
          obs.Add(rnd.Next());
      this.lstElements.DataContext = obs;
}</int></int>
```

This will produce 100000 elements to be added over the `ListBox`. If you use `VirtualizingStackPanel.IsVirtualizing=True`, the content will appear instantly, as it doesn't require to produce all the `ListBoxItem` elements from the first time. The application will hang if you make `IsVirtualizing=false` as creating 100000 `ListboxItem` elements takes a lot of time.

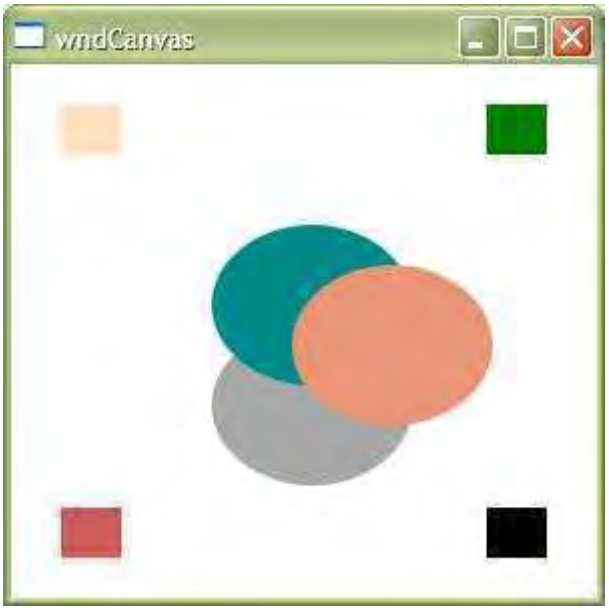
`VirtualizationMode` can be of two types:

1. **Standard**: It means the Item will be created when the `ScrollViewer` is scrolled.
2. **Recycling**: It means the item will be replaced with Data when `ScrollViewer` is scrolled.

CANVAS

`Canvas` is a special `Layout` panel which positions elements with absolute position means using x and y co-ordinates. When used within a `Canvas`, elements are not restricted to anything. It can be overlapped when the position intersects with other controls. Each element is drawn based on the sequence of its declaration. You can easily use `Panel.ZIndex` to make this declaration unspecific.

`Canvas` doesn't employ any restriction to its elements. So the width and height of individual elements is very necessary to be specified. You can use `Canvas.Left, Canvas.Right, Canvas.Top` and `Canvas.Bottom` to specify the co-ordinates. The only thing you need to remember is that `Canvas.Left` and `Canvas.Right` are the same, but it determines the start point of co-ordinate system from the extreme left or extreme right.




```
<Canvas>
     <Border Canvas.Top="20" Canvas.Left="25" Background="Bisque" Width="30"
          Height="25" />
     <Border Canvas.Top="20" Canvas.Right="25" Background="Green" Width="30"
        Height="25" />
     <Border Canvas.Bottom="20" Canvas.Right="25"
           Background="Black" Width="30" Height="25" />
     <Border Canvas.Bottom="20" Canvas.Left="25" Background="IndianRed"
           Width="30" Height="25" />
     <Ellipse Fill="DarkGray" Canvas.Left="100" Canvas.Top="130" Width="100"
          Height="80"></Ellipse>
     <Ellipse Fill="DarkCyan" Canvas.Left="100" Canvas.Top="80" Width="100"
           Height="80"></Ellipse>
     <Ellipse Fill="DarkSalmon" Canvas.Left="140" Canvas.Top="100" Width="100"
         Height="80" />
</Canvas>
```

Here you can see the `Border` elements even though placed in the same area, the `Canvas` properties change the Co-ordinate system and thus place on the four sides of the window.

The ellipses are Overlapped between one another in the same sequence as it is specified.

UNIFORMGRID

`UniformGrid` is a special control which adjusts its elements uniformly. If you want your Rows and Columns to be uniform in your grid, you can use `UniformGrid` instead of Normal Grid declaration.

In case of `UniformGrid`, the child elements will always be of the same size.


```
<UniformGrid Columns="2" Rows="3">
        <Border Background="Red" />
        <Border Background="Green" />
        <Border Background="Blue" />
        <Border Background="Yellow" />
        <Border Background="DarkGoldenrod" />
        <Border Background="DarkKhaki" />
    </UniformGrid>
```

So its 3X2 Grid and all elements are placed according to the sequence it is specified.

ScrollViewer

It is often happens that the elements go outside of the Display area. In that case, `ScrollViewer` places an Automatic `Scrollbars` which can be used to view the area outside the bounds. `ScrollViewer` encapsulates the `ScrollBar`s within it and displays it whenever it is required. As the `ScrollViewer` implements `IScrollInfo` in the main scrolling area inside the `scrollviewer`. `ScrollViewer` also responds to mouse and keyboard commands.


```
<ScrollViewer HorizontalScrollBarVisibility="Auto">
            <StackPanel VerticalAlignment="Top" HorizontalAlignment="Left">
```

```
            <TextBlock TextWrapping="Wrap" Margin="0,0,0,20">Scrolling is
                enabled when it is necessary. Resize the window, making it larger
                  and smaller.</TextBlock>
            <Rectangle Fill="Honeydew" Width="500" Height="500"></Rectangle>
        </StackPanel>
    </ScrollViewer>
```

The `CanContentScroll` property of a `ScrollViewer` determines whether the elements would be scrollable or not. `HorizontallScrollBarVisibility` and `VerticalScrollBarVisibility` make the scrollbar appear accordingly. The default behaviour is `Auto` which means the `scrollbar` will appear only when it is required.

GroupBox

`GroupBox` allows to group content with a custom header in it. This is the same as `GroupBox` we know in windows. The `Header` property takes the `text` element which is placed as `header` of the `GroupBox`. As `GroupBox` is a `ContentControl`, it can have only a single element in the body. So you need to use `Panel`s to add children within it.


```
<GroupBox Header="This goes to Header" Margin="50">
        <StackPanel>
            <Button Content="First Element"/>
            <Button Content="Second Element" />
        </StackPanel>
    </GroupBox>
```

Expander

`Expander` is same as `groupbox` but has extra facility to expand content. It is also derived form `HeaderedContentControl` and thus has one single content within it. The `IsExpanded` property determines if the panel is expanded or not.

`ExpandDirection` is used to make the content expanded behaviour. It has four Directions, `Down, Up, Right and Left`. You can use them to change the Expanded direction of the content.

```
<Expander Header="This goes to Header" Margin="50" IsExpanded="True"
            ExpandDirection="Down">
            <StackPanel>
                <Button Content="First Element"/>
                <Button Content="Second Element" />
            </StackPanel>
        </Expander>
```

ViewBox

`ViewBox` is a special WPF control which stretches or scales the contents of the elements. This comes in very handy to allow anchoring of the elements, as in case of `ViewBox` the controls will never change its position, rather the whole content will be stretched or shrunk.

The `Stretch` property of `ViewBox` can have four properties:

1. `Fill`: Fills the content and also makes the Aspect Ratio intact.
2. `None`: Stretch behaviour will not be set.
3. `UniformToFill`: Fills the element uniformly, and changes the Aspect Ratio.
4. `Uniform`: Uniformly enlarges the content.

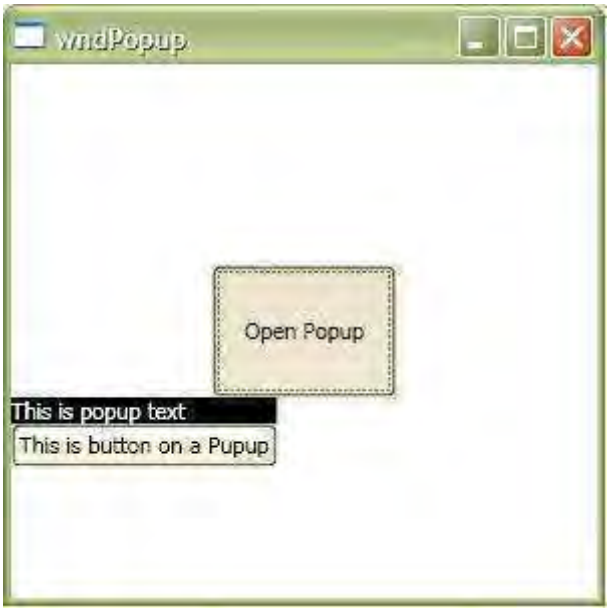
The `stretchDirection` can be specified as `Both, DownOnly and UpOnly`.


```
<Viewbox Stretch="None" StretchDirection="Both" >
        <Grid>
            <TextBox Text="This is a content" FontWeight="Bold" FontSize="30" />
        </Grid>
    </Viewbox>
```

Popup

`Popup` is a special control that is used to create floating window over the actual window. `Popup` is a control that is rendered always on Top of the window. `Popup` is used to display quick elements whenever it is needed without altering the whole window.

A `Popup` control can be positioned using properties called `PlacementTarget`, `PlacementRectangle`, `Placement`, `HorizontalOffset`, `VerticalOffset`, etc. A `popup` is a window outside the bounds of the existing WPF window, and thus it can be moved outside the whole content area of the XAML. WPF `popup` control supports few animation like `Fade`, `Scroll`, `Slide`, etc. which you can apply to it using `PopupAnimation` property. A WPF `Popup` supports transparency when `AllowsTransparency` is set to `true`.




```
<ToggleButton IsChecked="{Binding ElementName=pup, Path=IsOpen}"
              Content="Open Popup" Margin="100" />
        <Popup Placement="Bottom" AllowsTransparency="True"
                    PopupAnimation="Fade" x:Name="pup" VerticalOffset="-100">
            <StackPanel>
                <TextBlock Name="McTextBlock" Background="Black" Foreground="White" >
                    This is popup text
                </TextBlock>
                <Button Content="This is button on a Popup" />
            </StackPanel>
        </Popup>
```

Here, the `Popup` will be displayed when the `ToggleButton` is clicked as `IsChecked` is bound to `IsOpen` of `Popup`. When `IsOpen` is `true`, the `popup` will be displayed.

InkCanvas

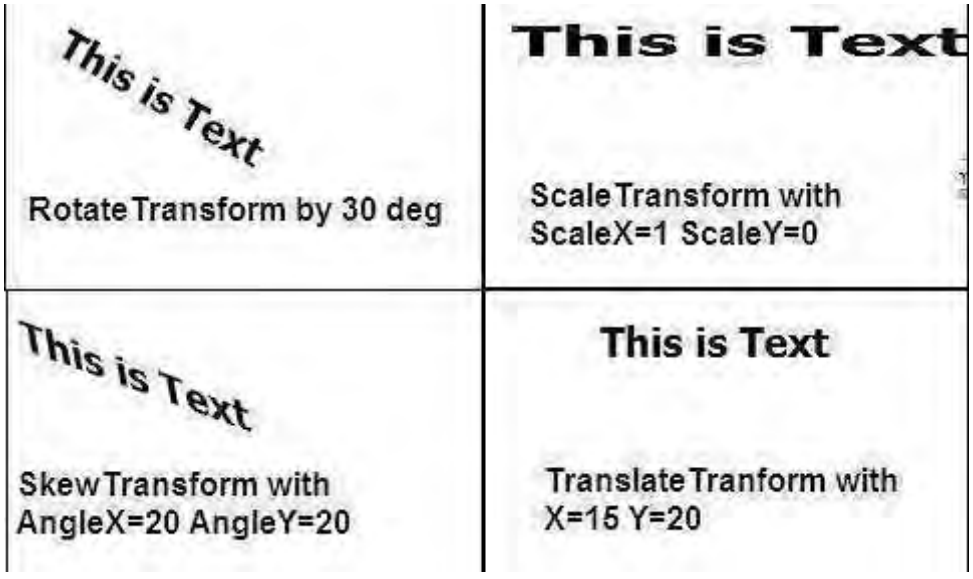
Another most powerful control which is introduced with WPF is `InkCanvas`. This control allows you to draw over the `Canvas` and ultimately get the image bytes saved. It is very powerful as you can easily get the strokes drawn over the canvas as `Objects`.

Just place an `InkCanvas` on the WPF Window and you will find that you can draw over the screen. The `EditingMode` gives you few editing mode for the `InkCanvas`. We will discuss with this control later in another article.


```
<StackPanel>
        <InkCanvas Height="200" x:Name="icBox">
        </InkCanvas>
        <RadioButton GroupName="mode" Checked="Pen_Checked" Content="Pen"/>
        <RadioButton GroupName="mode" Checked="Erase_Checked"
              Content="Eraser By Point" />
        <RadioButton GroupName="mode" Checked="EraseByStroke_Checked"
                Content="Eraser By Stroke" />
    </StackPanel>
```

Transformation

Transformation is one of the important features that is introduced with WPF. Transformation allows to map element from one co-ordinate space to another co-ordinate space. The transformation is mapped using Transformation Matrix in 2D space. By manipulating the matrix values, you can Transform elements to `Rotate, Scale, Skew and Translate`.



Transformation is of 4 basic types:

1. **RotateTranform**: Rotates an element by a specified `Angle`. You can specify the Angle of Rotation and the element will be rotated in 2D space.
2. **ScaleTransform**: `ScaleTransform` allows you to scale the element means increase/decrease the

size of the element in the 2D space.

3. **SkewTransform**: It skews the element by specified angle. Skew stretches elements in a `NonUniform` manner and thus the element will be transformed so as in 3D space.
4. **TranslateTransform**: This transformation will make the element move by a specified X and Y co-ordinates.

There is also a provision to apply more than one Transformation using `TransformGroup` or `MatrixTransform`. `TransformGroup` allows you to specify more than one `Transformation` to be applied on the single element and thus gives you a hybrid `Transformation` for your control.


```
<TextBlock FontWeight="Bold" FontSize="20" Text="This is Text" Margin="20">
            <TextBlock.RenderTransform>
                <TransformGroup>
                        <RotateTransform Angle="20" />
                        <SkewTransform AngleX="10" AngleY="10" />
                        <TranslateTransform X="15" Y="19"/>
                        <ScaleTransform ScaleX="2" ScaleY="1" />
                </TransformGroup>
            </TextBlock.RenderTransform>
        </TextBlock>
```

Conclusion

This is the second part of the series. I hope you like it. Please don't forget to write your feedback. I have skipped few things intentionally (like details of `InkCanvas`) as it would make the article very long. I will discuss that in a separate article.

Thanks for reading.

History

- 28th December, 2010: Initial post

License

About the Author

Abhishek Sur Did you like his post?

WPF Tutorial : Fun with Border & Brush

Licence CPOL
First Posted 28 Dec 2010
Views 21,486
Downloads 964
Bookmarked 27 times

By **Abhishek Sur** | 28 Dec 2010 | Article

C# .NET Dev WPF Intermediate

In this article, you will find most of the interesting things with Border and Brush elements.

4.84 (29 votes)

Download source - 32.04 KB

Table of Contents

Introduction

Border is the primary building block of any application in WPF. In my current application, I have been using lots of borders to adorn the User Interface. Starting from placing borders directly to the Window to putting borders in `ControlTemplate` of `ListBoxItem`, borders generally play a very important role in creating a better look and feel for the application. In this application, you will see how you can use Borders and most of the properties with ease.

- WPF Tutorial : Beginning [^]
- WPF Tutorial : Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial : Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutorial - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

Everyone knows what exactly the **Border** is. It is a rectangular area used to decorate the UI elements. The main difference between a `Rectangle` and a `Border` is that `Border` allows you to add one single child element inside it. `Border.Child` allows you to include a child `DependancyObject` inside a Border. Let us see a sample `Border`:

```
<Border Width="50" Height="50" x:Name="brdElement">
  <Border.Background>
     <SolidColorBrush Color="Bisque"></SolidColorBrush>
  </Border.Background>
  <Border.Effect>
    <DropShadowEffect BlurRadius="10" Color="Red" Direction="235" Opacity=".5"
           RenderingBias="Quality" ShadowDepth="10" />
  </Border.Effect>
</Border>
```

If you place this in your code, you will see something like the above. Let us look into detail what exactly I have done.

First of all, Width / Height determines the dimension of the `Border` element. `Border.Background` determines what will be the color of the `Brush` which will draw the inside of the `Border`. You can see the color is `Bisque`. You can define any type of `Brush` here. `SolidColorBrush` takes one `Color` element (which is here defined as `Bisque`) and fills the `Border` Background with that color. There are other properties too like `CornerRadius`, used to create `RoundedCorner Border`, etc. I will discuss them later in the article.

`Border Effect` can also be applied to a `Border`. Here I have added a `DropShadowEffect`. It allows you to put a shadow rendered outside the `Border`. The dependency properties that you need to take note of are:

1. **Color**: Defines the `Color` of the `Shadow`.
2. **Opacity**: Fades out the `Color`. You can see the `Red` color is faded out here to `.5`; Opacity ranges between 0 - 1.
3. **BlurRadius**: It defines the extent of shadow radius. Thus if you increase the size of `BlurRadius`, it will increase the `Shadow`.
4. **Direction**: It is the Light Direction in degrees. 235 degree implies where the shadow will focus, thus you can see 360 -235 is the angle where light is placed. Value ranges from 0 to 360.
5. **ShadowDepth**: It defines the depth of the `Shadow`. It means, how much the object is raised from the `Shadow`. If you increase the value of `ShadowDepth`, you will see, the being raised.

Now with these, let's create a few more:

```
<Border Width="50" Height="50" x:Name="brdElement">
                <Border.Background>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                        <LinearGradientBrush.GradientStops>
                            <GradientStop Color="Red" Offset="0"/>
                            <GradientStop Color="Pink" Offset=".5"/>
                            <GradientStop Color="Azure" Offset="1"/>
                        </LinearGradientBrush.GradientStops>
                    </LinearGradientBrush>
                </Border.Background>
                <Border.Effect>
                    <DropShadowEffect BlurRadius="10" Color="Red"
            Direction="45" Opacity=".4" RenderingBias="Performance" ShadowDepth="30" />
                </Border.Effect>
            </Border>
```

In the first sample, I have modified the `SolidColorBrush` to `LinearGradientBrush` with 3 `GradientStops`. It takes `StartPoint` and `EndPoint`. `StartPoint` defines where the `Gradient` will start. So 0,0 means starts from the `TopLeft` corner. First 0 represents the X axis Offset color, and second defines Y - axis Offset color.

Here I have used `Gradient` from `TopLeft` to `BottomRight`, so the `Gradient` will be straight. `GradientStops` defines the different colors on the `Gradient`. Here I have defined all the colors from 0 to 1. Thus the `Gradient` will start from 0,0 means Red to 1,1 as `Azure`. If I start as 0,1 to 1,0 it would

have been a Diagonal Gradient.

```
<Border Width="50" Height="50" x:Name="brdElement" BorderBrush="Goldenrod"
             BorderThickness="2">
                <Border.Background>
                    <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
                        <LinearGradientBrush.GradientStops>
                            <GradientStop Color="BurlyWood" Offset="0"/>
                            <GradientStop Color="MediumBlue" Offset=".5"/>
                            <GradientStop Color="SlateGray" Offset="1"/>
                        </LinearGradientBrush.GradientStops>
                    </LinearGradientBrush>
                </Border.Background>
                <Border.Effect>
                    <DropShadowEffect BlurRadius="10" Color="CadetBlue" Direction="0"
            Opacity=".4" RenderingBias="Performance" ShadowDepth="15" />
                </Border.Effect>
            </Border>
```

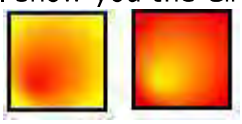
In this version, I have modified the `Colors` of the `Gradient`. You can see the `DropShadow` `Color`, `ShadowDepth` and `Direction` is changed as well to demonstrate to you how it modifies.

The `BorderBrush` and `BorderThickness` defines the border element of the Border. It means it draws the outer boundary of the `Border`.

More of Brushes

As I have already discussed, the most common `Brush` viz `LinearGradientBrush`, `SolidColorBrush`; Let us look into other brushes that are available to us.

1. **`RadialGradientBrush`**: It produces a circular gradient. Thus if I place a `RadialGradientBrush` instead a `LinearGradientBrush`, it will show you the Circular gradient.

   In the above, `RadialGradientBrush` is used to produce these borders. Let's look at the code:

   ```
   <Border Width="50" Height="50" BorderBrush="Black" BorderThickness="2">
                   <Border.Background>
                       <RadialGradientBrush GradientOrigin=".25,.75" RadiusX=".6"
                        RadiusY=".6">
                           <RadialGradientBrush.GradientStops>
                               <GradientStop Color="Red" Offset="0"></GradientStop>
                               <GradientStop Color="Yellow" Offset="1"></GradientStop>
                           </RadialGradientBrush.GradientStops>
                       </RadialGradientBrush>
                   </Border.Background>
               </Border>
   ```

   The `GradientOrigin` determines where the Origin is when we take the whole area to be 1. So If you place a value more than 1, Origin will lie outside the border. I have placed .25,.75 in this case.

   The `RadiusX` and `RadiusY` determines the `Radius` of the Gradient. Finally the `GradientStops` determines the actual gradient colors. Just interchanging the Offsets will produce the 2nd image.

2. **`ImageBrush`**: It allows you to draw using `Image`. You need to specify the `ImageSource` to determine what Image is to be drawn.

Here an `ImageBrush` is specified with my image. I have also added a `BitmapEffect` to the `Border` with some noise to distort the image a little.

```
<Border Width="100" Height="100" >
                <Border.Background>
                    <ImageBrush ImageSource="logo.jpg" Opacity=".7">
                        <!--<ImageBrush.Transform>
                            <SkewTransform AngleX="10" AngleY="10" />
                        </ImageBrush.Transform>-->
                    </ImageBrush>
                </Border.Background>
                <Border.BitmapEffect>
                    <OuterGlowBitmapEffect GlowColor="Brown" GlowSize="20" Noise="3"/>
                </Border.BitmapEffect>
            </Border>
```

The `Opacity` specifies the opacity of the image drawn inside the `Border`.

In addition to this, I have added one `BitmapEffect` with `OuterGlowEffect`. `OuterGlow` allows you to glow the outer section of the `Border`. I used Brown glow with `GlowSize` = 20 and `Noise`=3. `Noise` is used to distort the image, just seen in the image.

3. **`VisualBrush`**: This allows you to draw using an already visual element. It is very simple to use. Just see:

In the first Image, I have used `VisualBrush` to draw the `Image` on the right side which draws itself as the left side. I have modified the `OuterGlowBitmapEffect` to `BevelBitmapEffect` in the next version, to have a bevel effect to the image. The `VisualBrush` is also flipped XY so it seems upside down. See how the code looks like:

```
<Border Width="100" Height="100" x:Name="brdElement" CornerRadius="5" >
                <Border.Background>
                    <ImageBrush ImageSource="logo.jpg" Opacity=".7">
                    </ImageBrush>
                </Border.Background>
                <Border.BitmapEffect>
                    <BevelBitmapEffect BevelWidth="5" EdgeProfile="BulgedUp"
                   LightAngle="90" Smoothness=".5" Relief=".7"/>
                </Border.BitmapEffect>
            </Border>
            <Border Width="100" Height="100" Margin="20,0,0,0">
                <Border.Background>
                    <VisualBrush TileMode="FlipXY" Viewport="1,1,1,1"
```

```
        Stretch="UniformToFill" Visual="{Binding ElementName=brdElement}">
               </VisualBrush>
               </Border.Background>
            </Border>
```

The `VisualBrush` is bound to `brdElement`, which represents the Visual element placed in the window. `TileMode` indicates the `Flip` direction of the actual visual. Thus if there is a button or any other visual placed other than the border, it would look flipped as well. So `VisualBrush` comes in very handy at times.

Other than that, there are lots of `Brush`es as well like `DrawingBrush`, used to draw `Geometry` objects, etc.

Border Effects

As I have already used some `Border` Effects previously, let's see what are the main effects that you might use in real life.

`Border` element or any element inherited from `Border` supports two types of Effects.

1. **`Effect`**: Effects are applied to the whole `Border`. Thus any control inside the `Border` will also be affected with the effect.

Thus you can see using `BlurEffect` in the `Border` actually affects the `Text` written in the `TextBlock` inside the `Border`. You will get a clear idea by looking at the code below:

```
<Border Background="AliceBlue" Width="100" Height="100" CornerRadius="5"
       BorderBrush="Black" BorderThickness="2">
                <Border.Effect>
                    <BlurEffect Radius="3" RenderingBias="Quality" />
                </Border.Effect>
                <TextBlock HorizontalAlignment="Center"
         VerticalAlignment="Center" Text="This is inside Blured Border"
            TextWrapping="Wrap" TextTrimming="WordEllipsis"/>
            </Border>
```

2. **`DropShadowEffect`**: It is used to place a shadow outside the `Border`. I have already discussed it in the previous section of the article, so there is no need to elaborate this anymore.

Border BitmapEffects

`Border` also defines `BitmapEffect`. I have already discussed about `OuterGlowBitmapEffect` and `BevelBitmapEffect`, so let's discuss about the rest.

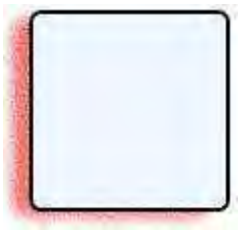
1. **`EmbossBitmapEffect`**: This effect will emboss the whole border. The `LightAngle` specifies the angular direction of light placed on the border. So if you write something inside the `Border`, it would have a shadow effect automatically just like the below:

If you see the code, it looks like:

```
<Border Background="AliceBlue" Width="100" Height="100" CornerRadius="5"
             BorderBrush="Black" BorderThickness="2">
                <Border.BitmapEffect>
                    <EmbossBitmapEffect LightAngle="270" Relief=".4" />
                </Border.BitmapEffect>
                <TextBlock HorizontalAlignment="Center" Foreground="Gold"
      FontSize="20" VerticalAlignment="Center" Text="This is Embossed"
       TextWrapping="Wrap" TextTrimming="WordEllipsis"/>
            </Border>
```

The `Text` inside the `textblock` is Embossed with the whole border itself.

2. **`DropShadowBitmapEffect`**: You can also specify `dropshadow` using `BitmapEffect`. It allows to add `Noise` to it as well.



The code will look like:

```
<DropShadowBitmapEffect Color="Red" Direction="200" Noise=".6"
       ShadowDepth="10" Opacity=".6"/>
```

This is the same as normal `DropShadowEffect` but a bit of enhancements. For `BitmapEffects`, you can also add all the effects at a time using `BitmapEffectGroup`.

Points to Remember

While working, I have found both `Effect` and `BitmapEffect` of `Border`, even though it looks great for an application, it may often deteriorate performance of the application. So it is always better to avoid `Effect`s. If you still like to use them, always use the `effect`s to small elements. Do not put say `BitmapEffect` to a border that spans the whole window. This will seriously slow down the WPF application.

Conclusion

Borders and brushes are the most commonly used objects in XAML. So I hope you like the article and that it has helped you a lot. Thanks for reading.

License

About the Author

Abhishek Sur

Did you like his post?

Oh, lets go a bit further to know him better.
Visit his Website : www.abhisheksur.com to know more about Abhishek.

Basically he is from India, who loves to explore the .NET world. He loves to code and in his leisure you always find him talking about technical stuffs.

WPF Tutorial - TypeConverter & Markup Extension

Licence CPOL
First Posted 28 Dec 2010
Views 17,770
Downloads 876
Bookmarked 27 times

By **Abhishek Sur** | 28 Dec 2010 | Article

C# .NET Dev WPF Intermediate

XAML as an Extensible Markup Language has great flexibilities to create objects in XAML itself and do functions like automatic binding, Conversion of Data, etc. Markup Extension allows you to truly extend your markup upto a certain extent to elevate you to write less code and design your application

4.78 (20 votes)

Download WPFSampleDemo - 68.26 KB

Table of Contents

Introduction

In my previous article I have discussed about basic architecture of WPF, and then gradually started learning with Layout panels, Transformation, introduced different Controls, containers, UI Transformation etc. In this article I will discuss the most important part of XAML which every developer must learn before starting with any XAML applications.

Markup Extensions are extensions to XAML which you might use to assign custom rules on your XAML based Applications. Thus any custom behavior which you would like to impose on your application in your designer, you should always use Markup Extensions. Here we will discuss how you can use Markup Extensions to generate your custom behaviors to XAML.

XAML or Extensible Application Markup Language is actually an XML format which has special schema defined. Now you might always wonder, how extensible the Markup is. What type of capabilities that are there within XAML which widely differs the XAML with XML. Yes, it is because of XAML parser which has

huge number of capabilities to make the normal XML to a very rich UI designs.

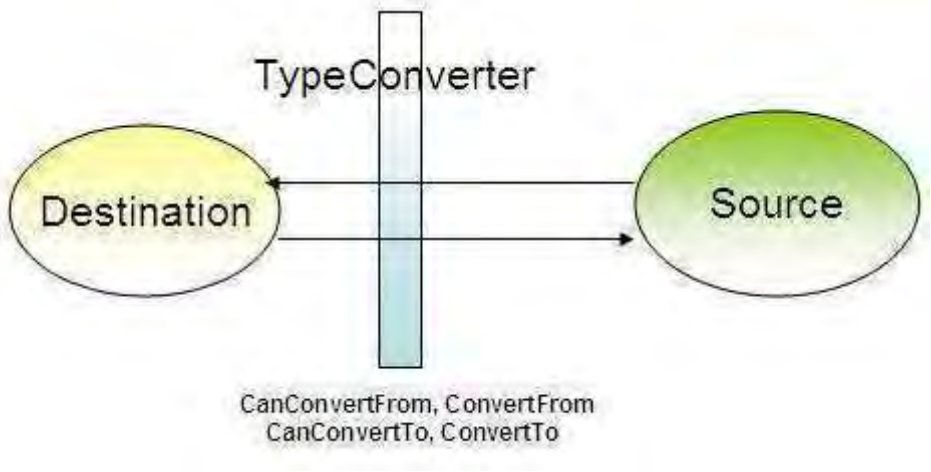
You all know XAML is actually a text format. The tags are very same as with any XML where the attributes takes everything as String. Even though you want to assign a object to a string, you cannot do so because the object can only take a string. Markup Extension allows you to handle these kind of situations. So you can say a Markup extension is actually the way to extend a normal XML to a complete Extensible Markup as XAML.

- WPF Tutorial : Beginning [^]
- WPF Tutorial : Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial : Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutoriall - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

As XAML takes everything as string, sometimes we need to convert those data into valid values. For instance, when we use `Margin`, we need to specify each values of margin element. In such situation where the conversion is very simple and straight forward, we can use Type Converters to do this job rather than going for Markup extensions. Lets discuss about Type Converters first before we move to Markup Extensions.

Type Converter

As I already told you, markup as an extension of XML cannot impose restriction over a data element. That means we can only specify string data for attributes of any object in XAML. But XAML provides a flexibility to create your Type converter which allows you to impose restriction on the data. Thus even primitives like Single or Double could not have restrictions while you describe in your XAML. Type Converters plays a vital role to put this restriction to XAML parser.



XAML parser while parsing any value of an attribute needs two pieces of information.

1. Value Type : This determines the Type to which the string data should be converted to.
2. Actual Value

Well, when parser finds a data within an attribute, it first looks at the type. If the type is primitive, the parser tries a direct conversion. On the other hand, if it is an Enumerable, it tries to convert it to a particular value of an Enumerable. If neither of them satisfies the data, it finally tries to find appropriate Type Converters class, and converts it to an appropriate type. There are lots of typeconverters already defined in XAML, like `Margin`. **Margin = 10,20,0,30** means margin :left,top,right,bottom is defined in sequence. Therefore system defines a typeconverter that converts this data into `Thickness` object.

Custom TypeConverter

To create a TypeConverter we need to decorate the Type with `TypeConverterAttribute` and define a custom class which converts the data into the actual type. And the actual converter class, a class which

implements from `TypeConverter`.

Let us make it clear using an Example :

TypeConverter to convert a GeoPoint:

To create a `TypeConverter` as I have already told you, you need to create a class for which the `TypeConverter` to be applied. In my example, I have created a class which has two properties called `Latitude` and `Longitude` and creates an implementation of a Geographic `Point`. Lets see how the class looks like :

```
[global::System.ComponentModel.TypeConverter(typeof(GeoPointConverter))]
    public class GeoPointItem
    {
        public double Latitude { get; set; }
        public double Longitude { get; set; }

        public GeoPointItem()
        {
        }

        public GeoPointItem(double lat, double lon)
        {
            this.Latitude = lat;
            this.Longitude = lon;
        }

        public static GeoPointItem Parse(string data)
        {
            if (string.IsNullOrEmpty(data)) return new GeoPointItem();

            string[] items = data.Split(',');
            if (items.Count() != 2)
                throw new FormatException("GeoPoint should have both latitude
                and longitude");

            double lat, lon;
            try
            {
                lat = Convert.ToDouble(items[0]);
            }
            catch (Exception ex) {
                throw new FormatException("Latitude value cannot be converted", ex);
            }

            try
            {
                lon = Convert.ToDouble(items[1]);
            }
            catch (Exception ex) {
                throw new FormatException("Longitude value cannot be converted", ex);
            }

            return new GeoPointItem(lat, lon);
        }

        public override string ToString()
        {
            return string.Format("{0},{1}", this.Latitude, this.Longitude);
        }
    }
```

In the above code you can see that I have created a quite normal class, which defines a Geographic point

on Earth. The type has two parameters, Latitude and Longitude and both of which are `Double` values. I have also overridden the `ToString()` method, which is actually very important in this situation to get the string resultant of the object. The Parse method is used to parse a string format to `GeoPointItem`.

After implementing this, the first thing that you need to do is to decorate the class with `TypeConverter` Attribute. This attribute makes sure that the item is Converted easily using the `TypeConverter` `GeoPointConverter` passed as argument to the attribute. Thus while XAML parser parses the string, it will call the `GeoPointConverter` automatically to convert back the value appropriately.

After we are done with this, we need to create the actual converter. Lets look into it :

```
public class GeoPointConverter : global::System.ComponentModel.TypeConverter
    {

        //should return true if sourcetype is string
        public override bool CanConvertFrom(
         System.ComponentModel.ITypeDescriptorContext context, Type sourceType)
        {
            if (sourceType is string)
                return true;
            return base.CanConvertFrom(context, sourceType);
        }
        //should return true when destinationtype if GeopointItem
        public override bool CanConvertTo(
             System.ComponentModel.ITypeDescriptorContext context, Type destinationType)
        {
            if (destinationType is string)
                return true;

            return base.CanConvertTo(context, destinationType);
        }
        //Actual convertion from string to GeoPointItem
        public override object ConvertFrom(
     System.ComponentModel.ITypeDescriptorContext context,
         System.Globalization.CultureInfo culture, object value)
        {
            if (value is string)
            {
                try
                {
                    return GeoPointItem.Parse(value as string);
                }
                catch (Exception ex)
                {
                    throw new Exception(string.Format(
      "Cannot convert '{0}' ({1}) because {2}", value, value.GetType(), ex.Message), ex);
                }
            }

            return base.ConvertFrom(context, culture, value);
        }

        //Actual convertion from GeoPointItem to string
        public override object ConvertTo(
         System.ComponentModel.ITypeDescriptorContext context,
          System.Globalization.CultureInfo culture, object value, Type destinationType)
        {
            if(destinationType == null)
                throw new ArgumentNullException("destinationType");

             GeoPointItem gpoint = value as GeoPointItem;

            if(gpoint != null)
```

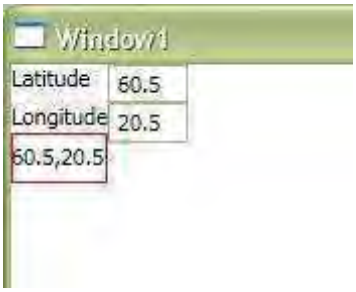
```
            if (this.CanConvertTo(context, destinationType))
                return gpoint.ToString();

            return base.ConvertTo(context, culture, value, destinationType);
        }
    }
```

In the above code we have implemented the converter class by deriving it from `TypeConverter`. After implementing it from TypeConverter class we need to override few methods which XAML parser calls and make appropriate modifications so that the XAML parser gets the actual value whenever required.

1. **CanConvertFrom** : This will be called when XAML parser tries to parse a string into a `GeopointItem` value. When it returns true, it calls `ConvertFrom` to do actual Conversion.
2. **CanConvertTo** : This will be called when XAML parser tries to parse a `GeoPointItem` variable to a string equivalent.When it returns true, it calls `ConvertTo` to do actual Conversion.
3. **ConvertFrom** :Does actual conversion and returns the `GeoPointItem` after successful conversion.
4. **ConvertTo** : Does actual conversion and returns the string equivalent of `GeoPointItem` passed in.

In the above example, you can see I have actually converter the string value to `GeoPointItem` and vice-versa using the `TypeConverter` class.



Now its time to use it. To do this I build a custom UserControl and put the property of that to have a `GeoPoint`. The XAML looks very simple :

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Latitude" Grid.Row="0" Grid.Column="0"></TextBlock>
        <TextBox x:Name="txtlat" MinWidth="40" Grid.Row="0" Grid.Column="1"
              TextChanged="txtlat_TextChanged"/>
        <TextBlock Text="Longitude" Grid.Row="1" Grid.Column="0"></TextBlock>
        <TextBox x:Name="txtlon" MinWidth="40" Grid.Row="1" Grid.Column="1"
             TextChanged="txtlon_TextChanged"/>
    </Grid>
```

It has 2 Textboxes which displays value of Latutude and Longitude individually. And when the value of these textboxes are modified, the actual value of the `GeopointItem` is modified.

```
public partial class GeoPoint : UserControl
{
        public static readonly DependencyProperty GeoPointValueProperty =
       DependencyProperty.Register("GeoPointValue", typeof(GeoPointItem),
             typeof(GeoPoint), new PropertyMetadata(new GeoPointItem(0.0, 0.0)));
        public GeoPoint()
        {
            InitializeComponent();
```

```
        }

        public GeoPointItem GeoPointValue
        {
            get
            {
                return this.GetValue(GeoPointValueProperty) as GeoPointItem;
            }
            set
            {
                this.SetValue(GeoPointValueProperty, value);
            }
        }

        private void txtlat_TextChanged(object sender, TextChangedEventArgs e)
        {
            GeoPointItem item = this.GeoPointValue;

            item.Latitude = Convert.ToDouble(txtlat.Text);
            this.GeoPointValue = item;
        }

        private void txtlon_TextChanged(object sender, TextChangedEventArgs e)
        {
            GeoPointItem item = this.GeoPointValue;

            item.Longitude = Convert.ToDouble(txtlon.Text);
            this.GeoPointValue = item;
        }

        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
            GeoPointItem item = this.GeoPointValue;
            this.txtlat.Text = item.Latitude.ToString();
            this.txtlon.Text = item.Longitude.ToString();

        }
    }
```

Here when the `UserControl` Loads, it first loads values passed in to the `TextBoxes`. The `TextChanged` operation is handled to ensure the actual object is modified whenever the value of the textbox is modified.

From the window, we need to create an object of the `UserControl` and pass the value as under:

```
<converter:GeoPoint x:Name="cGeoPoint" GeoPointValue="60.5,20.5" />
```

The converter points to the namespace. Hence you can see the value is shown correctly in the TextBoxes.

Markup Extension

Now as you know about `TypeConverter`, let us take `MarkupExtension` into account. Markup Extensions gives the flexibility to create custom objects into your XAML attributes. Every markup extension is enclosed within a {} braces. Thus anything written within the Curl braces will be taken as Markup Extension. Thus XAML parser treats anything within the curl braces not as literal string and rather it tries to find the actual MarkupExtension corresponding to the name specified within the Curl braces.

Exception

If you want to put braces within a string you need to put {} at the beginning to make it an exception. You can read "How to Escape {} in XAML" [^] for more information.

Example of Markup Extension :

```
<TextBox Text={x:Null} />
```

Probably the simplest of all, it actually a MarkupExtension which returns Null to the Text string.

There are already few Markup Extensions defined with XAML defined within System.Windows.Markup namespace which helps every time to produce functional features within your XAML. Let us discuss few of them :

NullExtension

This is the most simple `MarkupExtension` we ever have. It actually returns Null when placed against a value.

```
Content = "{x:Null}"
```

ArrayExtension

This is used to create an `ArrayList` of items. The x:Array actually returns an object of Array of type specified.

```
Values = {x:Array Type=sys:String}
```

StaticExtension

Another simple Extension which returns Static field or Property references.

```
Text="{x:Static Member=local:MyClass.StaticProperty}"
```

Thus when you define a static property StaticProperty for `MyClass` , you will automatically set the value to Text property using this.

TypeExtension

Type extension is used to get Types from objects. When the attribute of a control takes the Type object you can use it.

```
TargetType="{x:Type Button}"
```

So `TargetType` receives a Type object of `Button`.

Reference

It is used to Refer an object declared somewhere in XAML by its name. It is introduced with .NET 4.0

```
Text="{x:Reference Name=Myobject}"
```

StaticResourceExtension

Resources are objects that are defined within the XAML. `StaticResource` substitutes the key assigned to the object and replaces its reference to the Resource element declared.

```
<Grid.Resources>

<Color x:Key="rKeyBlack">Black</Color>
<SolidColorBrush Color="{StaticResource rKeyBlack}" x:Key="rKeyBlackBrush"/>


</Grid.Resources>

<TextBlock Background="{StaticResource ResourceKey=rKeyBlackBrush}" />
```

`StaticResource` errors out if the key is not there during compile time.

DynamicResourceExtension

This is same as `StaticResource`, but it defers the value to be a runtime reference. Thus you can declare any key for a DynamicResource and it should be present during runtime when the actual object is created.

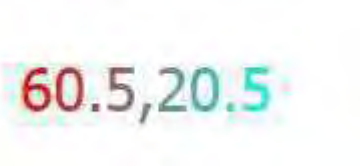
```
<TextBlock Background="{DynamicResource ResourceKey=rKeyBlackBrush}" />
```

Now the rKeyBlackBrush if not declared as Grid.Resources will not error out. You can add that during Runtime when Window Loads.

What are Resources ?

Resources are objects that are created the object is rendered by XAML parser. Every `FrameworkElement` object has a `ResourceDictionary` object placed within it. You can add Resources within this `ResourceDictionary` and can reuse those component within the scope.

Resources are reusable component which one can use for several times to produce its output. So if you need an object that you want to reuse more than once in your application, and you don't want the object to be changed within the scope, Resources are then the best option for you.




```
<Grid.Resources>
            <Color x:Key="rKeyRed">Red</Color>
            <Color x:Key="rKeyCyan">Cyan</Color>
            <LinearGradientBrush x:Key="rKeyforegroundGradient">
                <GradientStop Color="{StaticResource rKeyRed}" Offset="0"></GradientStop>
                <GradientStop Color="{StaticResource rKeyCyan}" Offset="1"></GradientStop>
            </LinearGradientBrush>
        </Grid.Resources>
<TextBlock Text="{Binding ElementName=cGeoPoint, Path=GeoPointValue}"
      FontSize="30" Margin="50" Grid.Row="1" Grid.ColumnSpan="2"
      Foreground="{StaticResource rKeyforegroundGradient}" />
```

So you can see we have defined a LinearGradientBrush and used as Foreground of the TextBlock. This object can be reused any time when required.

Every `FrameworkElement` has a property called `Resource` which takes an object of `ResourceDictionary`. You can assign various resources to this Collection which you would use in different object created within the scope of the object. In XAML, resources are declared using x:Key attribute, and later this key can be used to refer to the resource in ResourceDictionary using StaticResource or `DynamicResource`.

Difference between StaticResource and DynamicResource

StaticResource finds the key defined within the ResourceDictionary under its scope during the Loading of the application. Hence the Compiler will throw error during compilation if not found in resources. On the other hand, DynamicResource Markup Extension defers the resource assignment to the actual runtime of the application. So the expression remains unevaluated until the object being created.

Note : If the resource is derived from Freezable (immutable), any change to the object will change the UI regardless of whether it is a StaticResource or DynamicResource. Example : Brush, Animation, Geometry objects are Freezable.

Choice between StaticResource and DynamicResource

- **`StaticResource`** requires less CPU during runtime, so it is faster.
- `StaticResource` are created when the application loads. So making everything as StaticResource means slowing down the Application Load process.
- When the resources are unknown during compilation, you can use DynamicResource. `DynamicResource` are used when user interaction changes the look and feel of an object.
- `DynamicResource` is best when you want your resource to be pluggable. You can read how to create pluggable resources from : Pluggable Styles and Resources in WPF with Language Converter Tool [^]

Binding

Binding is the most important and complex Markup extension which might be used to bind one object. It provides Databound object when the data object is assigned to the DataContext of the object.

Thus say you have an object Obj which has several properties like Name, Age etc. Then you might write

```
Text = "{Binding Name}"
```

which means the `DataContext` object will automatically be evaluated during runtime and the actual value of Name property of the object will be shown on the Text property.

Binding has lots of flexibilities, thus you can specify expressions on the databound objects and hence made it one of the most complex Markup extension.


```
<StackPanel Orientation="Vertical">
        <TextBox x:Name="txtObj"></TextBox>
        <TextBox x:Name="txtCustom" Text="{Binding FallbackValue=10,
        ElementName=txtObj, Path=Text,StringFormat='Text Entered : {0:N2}'}, Mode=TwoWay">
        </TextBox>
    </StackPanel>
```

As both of them are bound any change to the property automatically reflects the changes to the other textbox. There are few modes of these binding, `OneTime, OneWayToSource, OneWay and TwoWay`. `StringFormat` creates formatting of the string. We can have Converters associated with a binding to enable us to return appropriate value based on the value we feed in.

You can read how to create Converter for Binding from :

How to deal with Converter in Binding [^]

In case of binding, you must also make sure that the object which is bound to have implemented INotifyPropertyChanged. Otherwise every binding will work as OneTime.

You can read more on how you can implement Binding with INotifyPropertyChanged from : Change Notification for objects and Collection [^]

We will discuss Binding in greater detail in next article.

RelativeSource

`RelativeSource` is a `MarkupExtension` which you have to use within Binding. Binding has a property called RelativeSource, where you can specify a RelativeSource MarkupExtension. This Markup Extension allows you to give Relative reference to the element rather than absolutely specifying the value. RelativeSource comes handy when absolute Reference is unavailable.

RelativeSource has few properties which one can use :

1. **AncestorType** : It defines the Type of the Ancestor element where the property to be found. So if you defined a button within a Grid, and you want to refer to that Grid, you might go for RelativeSource.
2. **Mode** : Determines how the `RelativeSource` to be found. The enumeration has few values :
    - **Self** : This means the Binding will take place within the object itself. So if you want the Background and Foreground of the same object, `RelativeSource` Mode =Self will come handy.
    - **FindAncestor** : Finds elements parent to it. Thus it will look through all the visual element into the Visual tree and try to find the control for which the AncestorType is provided and it goes on until the object is found in the Ancestor elements to it.
    - **TemplatedParent** : TemplatedParent allows you to find the value from the object for which the Template is defined. Templates are definition of the control itself which helps to redefine the Data and Control Elements altogether. TemplatedParent allows you to find the Templated object directly.
    - **PreviousData** : This allows you to track the Previous Data element when the object is bound to a CollectionView. Thus this is actually RelativeSource to previous DataElement.
3. **AncestorLevel** : Numeric value that determines how many levels that the RelativeSource should search before determining the result. If you specify 1 as AncestorLevel, it will go through only one level.

You can use `RelativeSource` for any binding.

```
<Grid Name="grd">

       <TextBox x:Name="txtCustom" Text="{Binding Name,
RelativeSource={RelativeSource AncestorType={x:Type Grid},
Mode=FindAncestor,AncestorLevel=2}}" />

</Grid>
```

This allows you to find the Grid. Practically speaking, this example doesnt gives you a sense when to use `RelativeSource` as in this case you can easily get the actual reference to the Grid. RelativeSource comes very useful when the Grid is somewhat inaccessible from it, say the TextBox is within the `DataTemplate` of a Grid.

TemplateBinding

TemplateBinding allows you to bind the value of a property of an object within the Template of a control to the object TemplatedParent to it.

```
<RadioButton Foreground="Red">
            <RadioButton.Template>
                <ControlTemplate>
                    <ToggleButton Content="{TemplateBinding Foreground}" />
                </ControlTemplate>
            </RadioButton.Template>
        </RadioButton>
```

In this case the ToggleButton Caption will be shown as #FFFF0000 which is the equivalent color for RadioButton.

MultiBinding

MultiBinding allows you to create Binding based on more than one Binding. You can create a class from `IMultiValueConverter` which will allow you to covert multipleBinding statements into one single output. The only difference between a normal converter and `MultiValueConverter` is a normal `IValueConverter` takes one value as argument whereas a `MultiValueConverter` takes an Array of values from all the Binding elements. We will discuss more on MultiBinding later in the series.

```
<Button x:Name="NextImageButton" >

    <Button.IsEnabled>

        <MultiBinding Converter="{StaticResource SelectedItemIndexIsNotLastToBoolean}">

            <Binding Mode="OneWay" ElementName="ImageListBox" Path="SelectedIndex" />

            <Binding Mode="OneWay" ElementName="ImageListBox" Path="Items.Count" />

        </MultiBinding>

    </Button.IsEnabled>

    <Image Source="Icons/navigate_right.png"/>

</Button>
```

Custom Markup Extension

In the final section, I will build my custom Markup Extension and use it. For simplicity we use Reflection to Get fields, Methods, and Properties and bind them into a ListBox. To create a custom Markup Extension,

you need to create a class and inherit it from MarkupExtension. This class has an abstract method called ProvideValue which you need to override to make the MarkupExtension work. So actually the XAML parser calls this ProvideValue to get the output from MarkupExtension. So the actual implementation looks like :

```
public class ReflectionExtension : global::System.Windows.Markup.MarkupExtension
    {
        public Type CurrentType { get; set; }
        public bool IncludeMethods { get; set; }
        public bool IncludeFields { get; set; }
        public bool IncludeEvents { get; set; }

        public ReflectionExtension(Type currentType)
        {
            this.CurrentType = currentType;
        }

        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            if (this.CurrentType == null)
                throw new ArgumentException("Type argument is not specified");

            ObservableCollection<string> collection = new ObservableCollection<string>();
            foreach(PropertyInfo p in this.CurrentType.GetProperties())
                collection.Add(string.Format("Property : {0}", p.Name));

            if(this.IncludeMethods)
                foreach(MethodInfo m in this.CurrentType.GetMethods())
                    collection.Add(string.Format("Method : {0} with {1}
                      argument(s)", m.Name, m.GetParameters().Count()));
            if(this.IncludeFields)
                foreach(FieldInfo f in this.CurrentType.GetFields())
                    collection.Add(string.Format("Field : {0}", f.Name));
            if(this.IncludeEvents)
                foreach(EventInfo e in this.CurrentType.GetEvents())
                    collection.Add(string.Format("Events : {0}", e.Name));

            return collection;
        }

    }
```

You can see the constructor for this class takes one argument of Type. Now to use it we refer it in XAML and use it in similar fashion as we do with other MarkupExtensions.

```
<ListBox ItemsSource="{local:Reflection {x:Type Grid},
       IncludeMethods=true, IncludeFields=true, IncludeEvents=true}"
         MaxHeight="200" Grid.Row="3" Grid.ColumnSpan="2" />
```

So here the Constructor gets argument from `{x:Type Grid}`. The first argument for any `MarkupExtension` is treated as Constructor argument. The other properties are defined using comma separated strings.

Conclusion

So this article deals with the basics of Markup Extension and Type Converters of basic XAML applications. We have left the Binding markup extension behind, and we will discuss on it on our next topic. I hope this article comes helpful to all of you. Thanks for reading.

License

WPF Tutorial - Dependency Property

By **Abhishek Sur** | 28 Dec 2010 | Article

C# | Dev | WPF | Intermediate

Licence CPOL
First Posted **28 Dec 2010**
Views **29,249**
Downloads **1,472**
Bookmarked **48 times**

WPF introduces a new property system which is enhanced by Dependency property. There are many improvements of Dependency Property over CLR properties. In this article, I have discussed how you could create your own Dependency Property and to work with various features of it.

4.87 (28 votes)

Download source - 28.25 KB

Table of Contents

Introduction

WPF comes with a lot of new features and alternatives that the normal Windows applications do not have. As we have already discussed some of the features of WPF, it's time to go a bit further to introduce other new features. After reading the previous articles of this tutorial, I hope you are more or less familiar with WPF architecture, borders, Effects, Transformation, Markup extensions, etc. If you aren't, please go through the series of articles as labelled:

- WPF Tutorial: Beginning [^]
- WPF Tutorial: Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial: Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutorial - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

So in this article, I am going to introduce you to a new property system that underlays the WPF property system. We will go further to introduce how easily you can use these properties to produce custom callbacks, create attached properties, apply animations and styles, etc. using the all new Dependency properties. Finally, we will discuss about Binding alternatives that were left behind in the previous article to finish it totally. I hope you will like this article as well as you did for all the tutorials that I provided.

A New Property System

Well, you must have been surprised to see this title. Yes, WPF comes with a completely new technique of defining a property of a control. The unit of the new property system is a Dependency property and the wrapper class which can create a Dependency property is called a `DependencyObject`. We use it to register a Dependency property into the property system to ensure that the object contains the property

in it and we can easily get or set the value of those properties whenever we like. We even use normal CLR property to wrap around a dependency property and use `GetValue` and `SetValue` to get and set values passed within it.

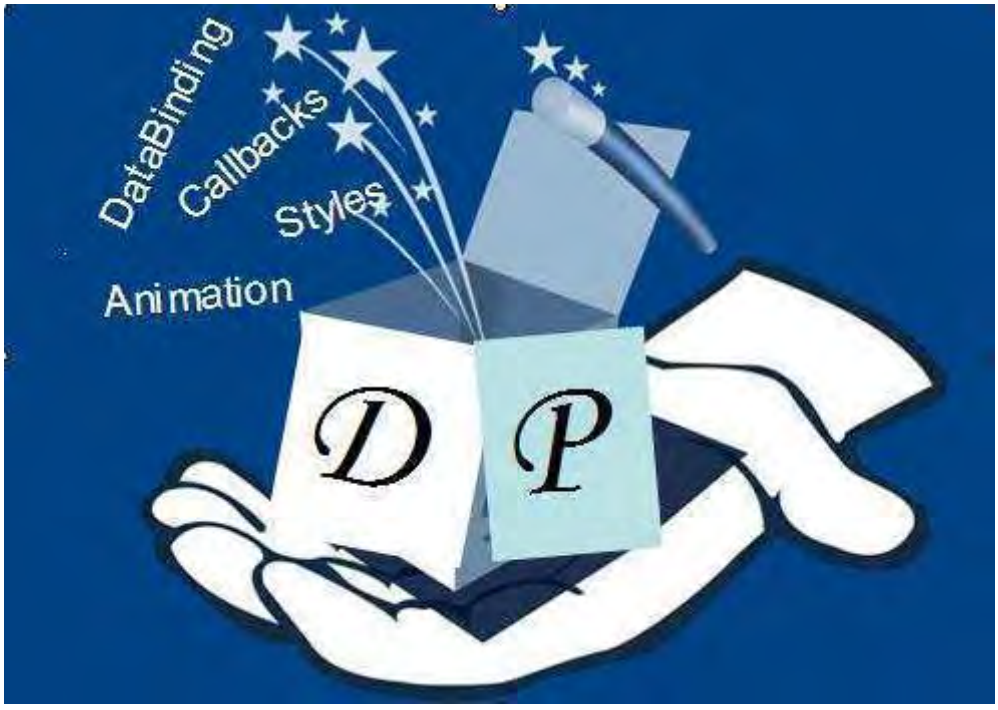
This is almost the same as CLR property system. So, what are the advantages of the new property system. Let's see the difference between Dependency property and CLR property.

To work with dependency property, you must derive the class from `DependencyObject` as the whole observer which holds the new Property System is defined within the `DependencyObject`.

Difference between Dependency Property and CLR Property

CLR property is just a wrapper around `private` variables. It uses `Get` / `Set` methods to retrieve and store value of a variable into it. So to be frank with you, a CLR property gives you only one block in which you can write code to invoke whenever a property is get or set. Hence CLR property system is fairly straightforward.

On the other hand, the capabilities of Dependency property system is huge. The idea of Dependency property is to compute the value of the property based on the value of other external inputs. The external inputs might be styles, themes, system properties, animations, etc. So, you can say a dependency property works with most of the WPF inbuilt features that were introduced.



Advantages of Dependency Property

As a matter of fact, a Dependency Property has lots of advantages over the normal CLR properties. Let's discuss the advantages a bit before we create our own dependency property:

1. **Property Value Inheritance**: By Property Value Inheritance, we mean that value of a Dependency property can be overridden in the hierarchy in such a way that the value with highest precedence will be set ultimately.
2. **Data Validation**: We can impose Data Validation to be triggered automatically whenever the property value is modified.
3. **Participation in Animation**: Dependency property can animate. WPF animation has lots of capabilities to change value at an interval. Defining a dependency property, you can eventually support Animation for that property.
4. **Participation in Styles**: Styles are elements that define the control. We can use `Style Setters` on Dependency property.
5. **Participation in Templates**: Templates are elements that define the overall structure of the element. By defining Dependency property, we can use it in templates.

6. **DataBinding**: As each of the Dependency properties itself invoke `INotifyPropertyChanged` whenever the value of the property is modified, `DataBinding` is supported internally. To read more about `INotifyPropertyChanged`, please read [^]
7. **CallBacks**: You can have callbacks to a dependency property, so that whenever a property is changed, a callback is raised.
8. **Resources**: A Dependency property can take a Resource. So in XAML, you can define a Resource for the definition of a Dependency property.
9. **Metadata overrides**: You can define certain behavior of a dependency property using `PropertyMetaData`. Thus, overriding a metadata form a derived property will not require you to redefine or re implementing the whole property definition.
10. **Designer Support**: A dependency property gets support from Visual Studio Designer. You can see all the dependency properties of a control listed in the Property Window of the Designer.

In these, some of the features are only supported by Dependency Property. Animation, Styles, `Templates`, Property value Inheritance, etc. could only be participated using Dependency property. If you use CLR property instead in such cases, the compiler will generate an error.

How To Define a Dependency Property

Now coming to the actual code, let's see how you could define a Dependency Property.

```
public static readonly DependencyProperty MyCustomProperty =
DependencyProperty.Register("MyCustom", typeof(string), typeof(Window1));
public string MyCustom
{
    get
    {
        return this.GetValue(MyCustomProperty) as string;
    }
    set
    {
        this.SetValue(MyCustomProperty, value);
    }
}
```

In the above code, I have simply defined a Dependency property. You must have been surprised why a dependency property is to be declared as `static`. Yes, like you, even I was surprised when I first saw that. But later on, after reading about Dependency property, I came to know that a dependency property is maintained in class level, so you may say Class A to have a property B. So property B will be maintained to all the objects that class A have individually. The Dependency property thus creates an observer for all those properties maintained by class A and stores it there. Thus it is important to note that a Dependency property should be maintained as `static`.

The naming convention of a dependency property states that it should have the same wrapper property which is passed as the first argument. Thus in our case, the name of the Wrapper `"MyCustom"` which we will use in our program should be passed as the first argument of the `Register` method, and also the name of the Dependency property should always be suffixed with `Property` to the original Wrapper key. So in our case, the name of the Dependency Property is `MyCustomProperty`. If you don't follow this, some of the functionality will behave abnormally in your program.

It should also be noted that you should not write your logic inside the `Wrapper` property as it will not be called every time the property is called for. It will internally call the `GetValue` and `SetValue` itself. So if you want to write your own logic when the dependency property is fetched, there are callbacks to do them.

Defining Metadata for Properties

After defining the most simplest Dependency property ever, let's make it a little enhanced. To Add metadata for a `DependencyProperty`, we use the object of the class `PropertyMetaData`. If you are inside a `FrameworkElement` as I am inside a `UserControl` or a `Window`, you can use `FrameworkMetaData` rather than `PropertyMetaData`. Let's see how to code:

```
static FrameworkPropertyMetadata propertymetadata =
new FrameworkPropertyMetadata("Comes as Default",
 FrameworkPropertyMetadataOptions.BindsTwoWayByDefault |
FrameworkPropertyMetadataOptions.Journal,new
PropertyChangedCallback(MyCustom_PropertyChanged),
new CoerceValueCallback(MyCustom_CoerceValue),
false, UpdateSourceTrigger.PropertyChanged);

public static readonly DependencyProperty MyCustomProperty =
DependencyProperty.Register("MyCustom", typeof(string), typeof(Window1),
propertymetadata, new ValidateValueCallback(MyCustom_Validate));

private static void MyCustom_PropertyChanged(DependencyObject dobj,
DependencyPropertyChangedEventArgs e)
{
  //To be called whenever the DP is changed.
  MessageBox.Show(string.Format(
     "Property changed is fired : OldValue {0} NewValue : {1}", e.OldValue, e.NewValue));
}

private static object MyCustom_CoerceValue(DependencyObject dobj, object Value)
{
//called whenever dependency property value is reevaluated. The return value is the
//latest value set to the dependency property
MessageBox.Show(string.Format("CoerceValue is fired : Value {0}", Value));
return Value;
}

private static bool MyCustom_Validate(object Value)
{
//Custom validation block which takes in the value of DP
//Returns true / false based on success / failure of the validation
MessageBox.Show(string.Format("DataValidation is Fired : Value {0}", Value));
return true;
}

public string MyCustom
{
get
{
return this.GetValue(MyCustomProperty) as string;
}
set
{
this.SetValue(MyCustomProperty, value);
}
}
```

So this is a little more elaborate. We define a `FrameworkMetaData`, where we have specified the `DefaultValue` for the `Dependency` property as "Comes as Default", so if we don't reset the value of the `DependencyProperty`, the object will have this value as default. The `FrameworkPropertyMetaDataOption` gives you a chance to evaluate the various metadata options for the dependency property. Let's see the various options for the enumeration.

- `AffectsMeasure`: Invokes `AffectsMeasure` for the `Layout` element where the object is placed.
- `AffectsArrange`: Invokes `AffectsArrange` for the layout element.
- `AffectsParentMeasure`: Invokes `AffectsMeasure` for the parent.
- `AffectsParentArrange`: Invokes `AffectsArrange` for the parent control.
- `AffectsRender`: Rerenders the control when the value is modified.
- `NotDataBindable`: `Databinding` could be disabled.
- `BindsTwoWayByDefault`: By default, databinding will be `OneWay`. If you want your property to have a `TwoWay` default binding, you can use this.

- `Inherits`: It ensures that the child control to inherit value from its base.

You can use more than one option using | separation as we do for flags.

The `PropertyChangedCallback` is called when the property value is changed. So it will be called after the actual value is modified already. The `CoerceValue` is called before the actual value is modified. That means after the `CoerceValue` is called, the value that we return from it will be assigned to the property. The validation block will be called before `CoerceValue`, so this event ensures if the value passed in to the property is valid or not. Depending on the validity, you need to return `true` or `false`. If the value is `false`, the runtime generates an error.

So in the above application after you run the code, the `MessageBox` comes for the following:

- **`ValidateCallback`**: You need to put logic to validate the incoming data as Value argument. `True` makes it to take the value, `false` will throw the error.
- **`CoerceValue`**: Can modify or change the value depending on the value passed as argument. It also receives `DependencyObject` as argument. You can invoke `CoerceValueCallback` using `CoerceValue` method associated with `DependencyProperty`.
- **`PropertyChanged`**: This is the final `Messagebox` that you see, which will be called after the value is fully modified. You can get the `OldValue` and `NewValue` from the `DependencyPropertyChangedEventArgs`.

Note on CollectionType Dependency Property

`CollectionType` dependency property is when you want to hold a collection of `DependencyObject` into a collection. We often require this in our project. In general, when you create a dependency object and pass the default value into it, the value will not be the default value for each instance of the object you create. Rather, it will be the initial value for the `Dependency` property for the type it is registered with. Thus, if you want to create a collection of `Dependency` object and want your object to have its own default value, you need to assign this value to each individual item of the dependency collection rather than defining using `Metadata` definition. For instance:

```
public static readonly DependencyPropertyKey ObserverPropertyKey =
DependencyProperty.RegisterReadOnly("Observer", typeof(ObservableCollection<Button>),
typeof(MyCustomUC),new FrameworkPropertyMetadata(new ObservableCollection<Button>()));
public static readonly DependencyProperty ObserverProperty =
     ObserverPropertyKey.DependencyProperty;
public ObservableCollection<Button> Observer
{
get
{
return (ObservableCollection<Button>)GetValue(ObserverProperty);
}
}
```

In the above code, you can see we declare a `DependencyPropertyKey` using the `RegisterReadonly` method. The `ObservableCollection` is actually a collection of `Button` which is eventually a `DependencyObject`.

Now if you use this collection, you will see that when you create object of the `Usercontrol`, each of them refers to the same `Dependency` Property rather than having its own dependency property. As by definition, each `dependencyproperty` allocates memory using its type, so if object 2 creates a new instance of `DependencyProperty`, it will overwrite the Object 1 collection. Hence the object will act as a `Singleton` class. To overcome with this situation, you need to reset the collection with a new instance whenever a new object of the class is created. As the property is `readonly`, you need to use `SetValue` to crate the new Instance using `DependencyPropertyKey`.

```
public MyCustomUC()
{
            InitializeComponent();
```

```
            SetValue(ObserverPropertyKey, new ObservableCollection<Button>());
}
```

So for each instance, the collection will reset and hence you will see a unique collection for each `UserControl` created.

PropertyValue Inheritance

`DependencyProperty` supports `Property` Value inheritance. By the definition, after you create a `DependencyObject` for you, you can easily inherit a `DependencyProperty` to all its child controls using `AddOwner` method associated to the `DependencyProperty`.

Each of `DependencyProperty` has `AddOwner` method, which creates a link to another `DependencyProperty` that is already defined. Say you have a `DependencyObject` A, which has a property called `Width`. You want the value of the `DependencyObject` B to inherit the value of A.

```
public class A :DependencyObject
{
public static readonly DependencyProperty HeightProperty =
   DependencyProperty.Register("Height", typeof(int), typeof(A),
new FrameworkPropertyMetadata(0,
FrameworkPropertyMetadataOptions.Inherits));

public int Height
{
get
{
return (int)GetValue(HeightProperty);
}
set
{
SetValue(HeightProperty, value);
}
}

public B BObject { get; set; }
}

public class B : DependencyObject
{
public static readonly DependencyProperty HeightProperty;

static B()
{
HeightProperty = A.HeightProperty.AddOwner(typeof(B),
new FrameworkPropertyMetadata(0, FrameworkPropertyMetadataOptions.Inherits));
}

public int Height
{
get
{
return (int)GetValue(HeightProperty);
}
set
{
SetValue(HeightProperty, value);
}
}
}
```

In the above code, you can see, the class `B` inherits the `DependencyProperty Height` using `AddOwner` without re declaring the same in the class. Thus when `A` is declared, if you specify the height

of A, it will automatically be transferred to the inherited child object B.

This is same as normal objects. When you specify the Foreground of a Window, it will automatically inherit to all the child elements, hence the Foreground of each control will behave the same.

Update

Even though PropertyValueInhertence is there for any dependency property, it will actually work for AttachedProperties. I came to know that Property Value Inheritance only works when the property is taken as attached. If you set the Default Value for the Attached property and also set FrameworkMetaData.Inherits, the property value will be inherited from Parent to Child automatically and also gives the chance for the Children to modify the content. Check MSDN [^] for more details. So, the example that I put above is not proper for Property Value Inheritance, but you can easily create one to see after you read the next section.

Attached Properties

Attached property is another interesting concept. Attached property enables you to attach a property to an object that is outside the object altogether making it define value for it using that object. Seems a little confused, huh? Yes, let's see an example.

Say you have declared a DockPanel, inside which you want your controls to appear. Now DockPanel registers an AttachedProperty.

```
public static readonly DependencyProperty DockProperty =
DependencyProperty.RegisterAttached("Dock", typeof(Dock), typeof(DockPanel),
 new FrameworkPropertyMetadata(Dock.Left,
new PropertyChangedCallback(DockPanel.OnDockChanged)),
 new ValidateValueCallback(DockPanel.IsValidDock));
```

You can see, the DockProperty is defined inside the DockPanel as Attached. We use RegisterAttached method to register attached DependencyProperty. Thus any UIElement children to the DockPanel will get the Dock property attached to it and hence it can define its value which automatically propagates to the DockPanel.

Let's declare an Attached DependencyProperty:

```
public static readonly DependencyProperty IsValuePassedProperty =
DependencyProperty.RegisterAttached("IsValuePassed", typeof(bool), typeof(Window1),
new FrameworkPropertyMetadata(new PropertyChangedCallback(IsValuePassed_Changed)));

public static void SetIsValuePassed(DependencyObject obj, bool value)
{
obj.SetValue(IsValuePassedProperty, value);
}

public static bool GetIsValuePassed(DependencyObject obj)
{
return (bool)obj.GetValue(IsValuePassedProperty);
}
```

Here I have declared a DependencyObject that holds a value IsValuePassed. The object is bound to Window1, so you can pass a value to Window1 from any UIElement.

So in my code, the UserControl can pass the value of the property to the Window.

```
<local:MyCustomUC x:Name="ucust" Grid.Row="0" local:Window1.IsValuePassed="true"/>
```

You can see above the IsValuePassed can be set from an external UserControl, and the same will be passed to the actual window. As you can see, I have added two static methods to individually Set

or `Get` values from the object. This would be used from the code to ensure we pass the value from code from appropriate objects. Say, you add a button and want to pass values from code, in such cases the `static` method will help.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
     Window1.SetIsValuePassed(this, !(bool)this.GetValue(IsValuePassedProperty));
}
```

In the similar way, `DockPanel` defines `SetDock` method.

Conclusion

To conclude, `DependencyProperty` is one of the most important and interesting concepts that you must know before you work with WPF. There are certain situations, where you would want to declare a `DependencyProperty`. From this article, I have basically visited each section of the `DependencyProperty` that you can work. I hope this one has helped you. Thank you for reading. Looking forward to getting your feedback.

License

About the Author

Abhishek Sur

Web Developer
Buildfusion Inc
India

Member

Follow on Twitter

Did you like his post?

Oh, lets go a bit further to know him better.
Visit his Website : www.abhisheksur.com to know more about Abhishek.

Basically he is from India, who loves to explore the .NET world. He loves to code and in his leisure you always find him talking about technical stuffs.

Presently he is working in WPF, a new foundation to UI development, but mostly he likes to work on architecture and business classes. ASP.NET is one of his strength as well.
Have any problem? Write to him in his Forum.

You can also mail him directly to abhi2434@yahoo.com

Want a Coder like him for your project?
Drop him a mail to contact@abhisheksur.com

Visit His Blog

Dotnet Tricks and Tips

Dont forget to vote or share your comments about his Writing

WPF Tutorial - Concept Binding

By **Abhishek Sur** | 31 Dec 2010 | Article

C# .NET Dev WPF Intermediate MVVM

Licence CPOL
First Posted **28 Dec 2010**
Views **23,379**
Downloads **1,730**
Bookmarked **41 times**

Binding is the most important topic of WPF programming. In this article, I have demonstrated how you could employ DataBinding to ensure that the Presentation logic is separated from the View and also give a simple demonstration on how the DataBinding concept works.

4.78 (19 votes)

Download source - 265.27 KB

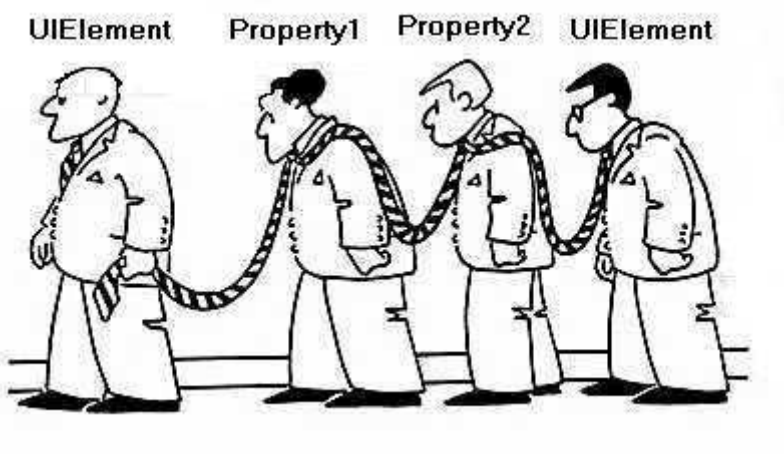
Table of Contents

Introduction

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

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

- WPF Tutorial : Beginning [^]
- WPF Tutorial : Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial : Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutoriall - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]



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

Binding in WPF

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

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

Binding can be classified into few Types.

DataBinding / Object Binding

The most important and primary binding is Databinding. WPF introduces objects like ObjectDataProvider and XMLDataProvider to be declared into XAML to enhance the capability of object binding. DataBinding can be achieved by several ways. As shown by Adnan in his blog [^], we can make use of Binding capabilities by employing either XAML, XAML and C#, and C# itself. So WPF is flexible enough to handle any situation.

```
<TextBox x:Name="txtName" />
<TextBlock Text="{Binding ElementName=txtName, Path=Text.Length}" />
```

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

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

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

```
<StackPanel Orientation="Vertical">
    <StackPanel.Resources>
      <ObjectDataProvider ObjectType="{x:Type m:StringData}"
         x:Key="objStrings" MethodName="GetStrings"/>
    </StackPanel.Resources>
    <ListBox Name="lstStrings" Width="200" Height="300"
           ItemsSource="{Binding Source={StaticResource objStrings}}" />
```

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

```
public class StringData
 {
  ObservableCollection<String> lst= new ObservableCollection<String>();

        public StringData()
        {
            lst.Add("Abhishek");
            lst.Add("Abhijit");
            lst.Add("Kunal");
            lst.Add("Sheo");
        }
        public ObservableCollection<String> GetStrings()
        {
             return lst;
        }
    }
```

So you can see the list has been populated with the `string`s.

Why ObservableCollection , the INotifyPropertyChanged, INotifyCollectionChanged?

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

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

I have already discussed the two in detail in an article : Change Notification for Objects and Collection [^].

On the contrary, Sacha has a good point of getting rid of `INotifyPropertyChanged` interface using Aspect Examples (INotifyPropertyChanged via aspects).

XML Binding

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

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

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

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

If you want to read more about XML binding, check: XML Binding in WPF [^].

Importance of DataContext

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

```
<Grid DataContext="{StaticResource dtItem}">
<TextBox Text="{Binding MyProperty}" />
</Grid>
```

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

Binding Members

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

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

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

     By this, the source gets updated.
6. **Converter**: Converter gives you an interface to put an object which will be invoked whenever the Binding objects get updated. Any object that implements IValueConverter can be used in place of Converter. You can read more about it from Converter in Binding [^].
7. **ConverterParameter**: It is used in addition to Converter to send parameters to Converter.
8. **FallbackValue**: Defines the value which will be placed whenever the Binding cannot return any value. By default, it is blank.
9. **StringFormat**: A formatting string that indicates the Format to which the data will follow.

10. **ValidatesOnDataErrors**: When specified, the `DataErrors` will be validated. You can use `IDataErrorInfo` to run your custom Validation block when Data object is updated. You can read more about `IDataErrorInfo` from : Validate your application using IDataErrorInfo [^].

Binding in Code-behind

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

```
Binding myBinding = new Binding("DataObject");
 myBinding.Source = myDataObject;
 myTextBlock.SetBinding(TextBlock.TextProperty, myBinding);
```

You can also specify the `Binding` properties in this way.

Command Binding

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

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

```
<Window.InputBindings>
        <KeyBinding Command="{Binding CreateNewStudent}" Key="N" Modifiers="Ctrl" />
        <MouseBinding Command="{Binding CreateNewStudent}"
                MouseAction="LeftDoubleClick" />
    </Window.InputBindings>
```

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

Note: In VS 2008, the `InputBindings` only take `Static` Command objects. There is a bug report for this [^] , and it will be fixed in later releases.

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

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

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

```
public class CommandBase : ICommand
    {
        private Func<object, bool> _canExecute;
        private Action<object> _executeAction;
        private bool canExecuteCache;

        public CommandBase(Action<object>executeAction, Func<object, bool> canExecute)
        {
            this._executeAction = executeAction;
            this._canExecute = canExecute;
        }

        #region ICommand Members

        public bool CanExecute(object parameter)
        {
```

```
            bool tempCanExecute = _canExecute(parameter);
            canExecuteCache = tempCanExecute;
            return canExecuteCache;
        }
        private event EventHandler _canExecuteChanged;
        public event EventHandler CanExecuteChanged
        {
            add { this._canExecuteChanged += value; }
            remove { this._canExecuteChanged -= value; }
        }
        protected virtual void OnCanExecuteChanged()
        {
            if (this._canExecuteChanged != null)
                this._canExecuteChanged(this, EventArgs.Empty);
        }
        public void Execute(object parameter)
        {
            _executeAction(parameter);
        }

        #endregion
    }
```

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

```
private CommandBase createNewstudent;
        public CommandBase CreateNewStudent
        {
            get
            {

                this.createNewstudent = this.createNewstudent ??
               new CommandBase(param => this.CreateStudent(), param => this.CanCreateStudent);
                return this.createNewstudent;
            }
        }

        private object CreateStudent()
        {
            this.CurrentStudent = new StudentItem();
            return this.CurrentStudent;
        }

        public bool CanCreateStudent
        {
            get { return true; }
        }
```

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

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

MultiBinding

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

```
<TextBlock DockPanel.Dock="Top" >
   <TextBlock.Text>
      <MultiBinding Converter="{StaticResource mbindingconv}">
        <Binding ElementName="lst" Path="Items.Count" />
        <Binding ElementName="txtName" Path="Text" />
        <Binding ElementName="txtAge" Path="Text" />
      </MultiBinding>
   </TextBlock.Text>
 </TextBlock>
```

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

```
public class MyMultiBindingConverter : IMultiValueConverter
    {
        #region IMultiValueConverter Members

        public object Convert(object[] values, Type targetType,
                object parameter, System.Globalization.CultureInfo culture)
        {
            string returnval = "Total no of Data {0}, NewData : ";

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

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

```
            for (int i = 1; i < values.Count(); i++)
                returnval += "- " + values[i];

            return returnval;
        }

        public object[] ConvertBack(object value, Type[]
            targetTypes, object parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }

        #endregion
    }
```

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

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

Conclusion

I think you must be enjoying the series. Also feel free to write your comments. Thanks for reading.

License

About the Author

Abhishek Sur

Web Developer
Buildfusion Inc
India

Member

Follow on Twitter

Did you like his post?

Oh, lets go a bit further to know him better.
Visit his Website : www.abhisheksur.com to know more about Abhishek.

Basically he is from India, who loves to explore the .NET world. He loves to code and in his leisure you always find him talking about technical stuffs.

Presently he is working in WPF, a new foundation to UI development, but mostly he likes to work on architecture and business classes. ASP.NET is one of his strength as well.
Have any problem? Write to him in his Forum.

You can also mail him directly to abhi2434@yahoo.com

Want a Coder like him for your project?
Drop him a mail to contact@abhisheksur.com

Visit His Blog

Dotnet Tricks and Tips

Dont forget to vote or share your comments about his Writing

WPF Tutorial - Styles, Triggers & Animation

By **Abhishek Sur** | 28 Dec 2010 | Article

C# .NET Dev WPF Intermediate

Licence CPOL
First Posted **28 Dec 2010**
Views **26,896**
Downloads **1,662**
Bookmarked **45 times**

In this article, I have specified how you could use Style, Triggers and animation in your WPF application to make your application more attractive, interactive and user friendly.

4.91 (38 votes)

Download WPFAnimationStyles - 67.37 KB

Table of Contents

Introduction

Perhaps the most interesting and most important feature for any WPF application is Styling. Styling means defining styles for controls, and store in reusable `ResourceDictionaries` and hence forth, it could be used later on by calling its name. Styles in WPF could be well compared with CSS styles. They are both similar in most of the cases, while the former extends the feature allowing most of the features which WPF have. In this article I will discuss mostly how you could use `Style` in your WPF application to enhance the Rich experience of your UI.

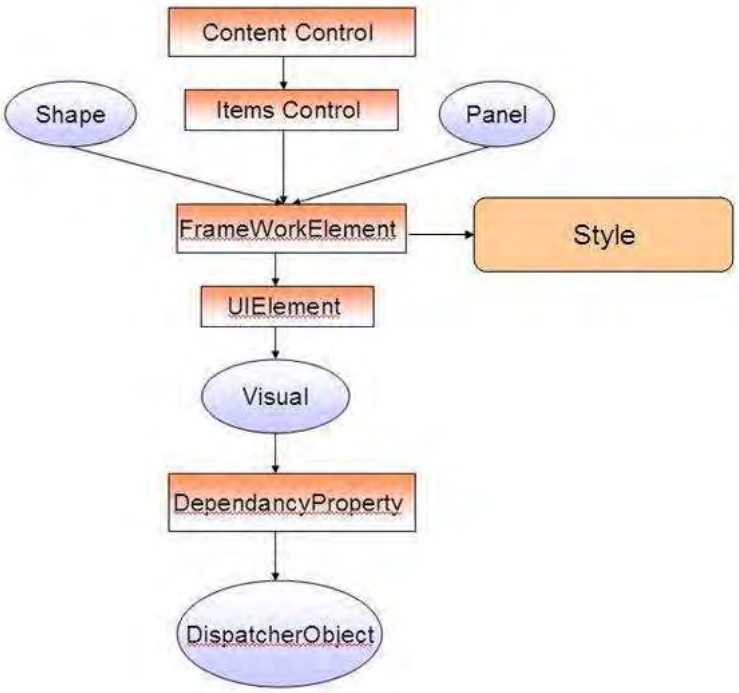
Before we go on further with styles, lets jot down what we have already discussed so far.

- WPF Tutorial : Beginning [^]
- WPF Tutorial : Layout-Panels-Containers & Layout Transformation [^]
- WPF Tutorial : Fun with Border & Brush [^]
- WPF Tutorial - TypeConverter & Markup Extension [^]
- WPF Tutorial - Dependency Property [^]
- WPF Tutoriall - Concept Binding [^]
- WPF Tutorial - Styles, Triggers & Animation [^]

So basically if you have already gone through the articles in the series, you must have already know most of the things on how you could apply your styles, how could you define style objects etc. In this article I will discuss how you could write better styles for your application.

Style

WPF exposes a property `Style` for every Control. If you look into the object Hierarchy, the Style is basically a property which exposes an object of `Style` in `FrameworkElement`. So each object can associate it and define custom setters to manipulate the basic look and feel of a control.



Clearly, the above diagram shows the association of `Style` in `FrameworkElement` and from the object hierarchy every control somehow inherits from `FrameworkElement` and hence style will be available to it. Style is also a WPF object which is inherited form `DispatcherObject` which helps in setting different properties of your UI Element.

How Style differs from Theme ?

Before we move further into Styles lets talk about Themes. Theme is totally different from Styles. Themes are defined at OS level, or more precisely a Theme can take part of delivering styles all over the Desktop while `Styles` are restricted to the contextual area of a WPF window. WPF are capable of retrieving the color scheme which is defined in OS level. Say for instance, if you do not define style for your application, the elements in the screen will automatically get styles from external environment. Say for instance, in XP if you change the theme to something else you would see that the buttons, `TextBox` on your WPF window will change its color instantly. You can even set the Theme which the application would use programmatically [^] from your code.

What about Templates ?

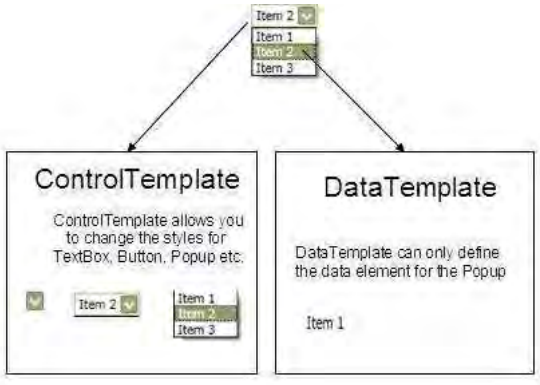
Every control defines a `ControlTemplate`. A `ControlTemplate` defines the overall structure of the control. As I have already told you, say for instance you have a `Button`. Button is a control that is made up of more than one control. It would have a `ContentPresenter` which writes the `Text` over the control, it would have a `Rectangle` which keeps the boundary of the `Button` etc. So Template is a special property associated with a Control which specifies how the control will look like structurally. We can easily define our `Template` and change the overall structure of a control.

Templates are basically of 2 types :

1. **ControlTemplate**
2. **DataTemplate**

`ControlTemplate` defines the structure of the `Control`. It means say for instance, you define the `ControlTemplate` for a `ComboBox`. So from `ControlTemplate` you can easily change the `ToggleButton` associated with the `ComboBox` which opens the `DropDown`, you can change the structure of the `TextBox`, the `Popup` etc. So `ControlTemplate` allows you to change the overall structure of the Control.

Each control is made up of Data. Say for instance a `ItemsControl` contains a number of Data Element which builds the items inside the Popup. The `DataTemplate` could be associated with `ItemsTemplate` and will build up the Data Block for the `ComboBox`.



So, you should always remember, `ControlTemplate` defines the whole Control while the `DataTemplate` defines each individual Data Element.

How to define Style?

Normally a style is an unique object which is used to style WPF controls. Each WPF element contains a number of Dependency Properties. A dependency property defines the basic behavior and look of the control in UI. Styles maintains a collection of `Setters` which enumerates a `DependencyProperty`

with its value.

Thus you can say a style is a collection of `DependencyProperty` settings which when applied on a Target will change the behavior of it.

Let us suppose you are going to style a `TextBox`.


```
<TextBox Text="This is a TextBox without Styles"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            CharacterCasing="Lower"
            FlowDirection="RightToLeft"
            FontSize="20"
            FontWeight="UltraBlack"
            Width="400"
            Height="40">
    <TextBox.Background>
        <LinearGradientBrush>
            <GradientStop Color="Cyan" Offset="0.0"/>
            <GradientStop Color="Yellow" Offset="0.5"/>
            <GradientStop Color="Red" Offset="1.0"/>
        </LinearGradientBrush>
    </TextBox.Background>
    <TextBox.Foreground>
        <SolidColorBrush Color="Black"/>
    </TextBox.Foreground>
    <TextBox.Effect>
        <DropShadowEffect BlurRadius="40" Color="Maroon"
           Direction="50" Opacity="0.5"/>
    </TextBox.Effect>
</TextBox>
```

So I have just designed a `TextBox` in the above code. The XAML looks straight forward, where I have configured different properties of the `TextBox` control to create my stylish `TextBox`. But looking at the code, you might wonder how difficult it would be if you need to redo the same thing again and again for every TextBox you define in your application. This is what the problem is. So WPF comes with an alternative with `style`. A `style` is an object that holds this behaviors into a collection of `Setters`. So lets redefine the same with `Styles`.


```
<TextBox>
    <TextBox.Style>
        <Style TargetType="{x:Type TextBox}">
            <Setter Property="Text" Value="This is a TextBox with Styles"/>
            <Setter Property="HorizontalAlignment" Value="Center"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="CharacterCasing" Value="Lower"/>
            <Setter Property="FlowDirection" Value="RightToLeft"/>
            <Setter Property="FontSize" Value="20"/>
            <Setter Property="FontWeight" Value="UltraBlack"/>
            <Setter Property="Width" Value="400"/>
            <Setter Property="Height" Value="40"/>
            <Setter Property="Background">
                <Setter.Value>
                        <LinearGradientBrush>
                        <GradientStop Color="Cyan" Offset="0.0"/>
                        <GradientStop Color="Yellow" Offset="0.5"/>
                        <GradientStop Color="Red" Offset="1.0"/>
                    </LinearGradientBrush>
                </Setter.Value>
            </Setter>
            <Setter Property="Foreground">
                <Setter.Value>
                    <SolidColorBrush Color="Black"/>
                </Setter.Value>
            </Setter>
            <Setter Property="Effect" >
                <Setter.Value>
                    <DropShadowEffect BlurRadius="40"
                     Color="Maroon" Direction="50" Opacity="0.5"/>
                </Setter.Value>
            </Setter>
        </Style>
    </TextBox.Style>
</TextBox>
```

So you can see, I have defined the `Style` inside the `TextBox` and the textbox looks almost the same. The Setters allows you to enumerate all the properties for the `TextBox` and produced a style inside it whose `TargetType` is set to `{x:Type Button}`

Now how this `style` can be made reusable for many controls ? Yes, this might be your first question that arose in your mind. Yes, if you have read my previous articles, you should already know the use of `ResourceDictionaries`. So in our case I will shift the style to Resource section for the Window and reuse the code just by calling the Resource key from the `Textbox`.

```
<Grid>
    <Grid.Resources>
```

```
        <ResourceDictionary>
            <Style TargetType="{x:Type TextBox}" x:Key="MyTextBoxStyle">
                <Setter Property="Text" Value="This is a TextBox with Styles"/>
                <Setter Property="HorizontalAlignment" Value="Center"/>
                <Setter Property="VerticalAlignment" Value="Center"/>
                <Setter Property="CharacterCasing" Value="Lower"/>
                <Setter Property="FlowDirection" Value="RightToLeft"/>
                <Setter Property="FontSize" Value="20"/>
                <Setter Property="FontWeight" Value="UltraBlack"/>
                <Setter Property="Width" Value="400"/>
                <Setter Property="Height" Value="40"/>
                <Setter Property="Margin" Value="0,20,0,10" />
                <Setter Property="Background">
                    <Setter.Value>
                        <LinearGradientBrush>
                            <GradientStop Color="Cyan" Offset="0.0"/>
                            <GradientStop Color="Yellow" Offset="0.5"/>
                            <GradientStop Color="Red" Offset="1.0"/>
                        </LinearGradientBrush>
                    </Setter.Value>
                </Setter>
                <Setter Property="Foreground">
                    <Setter.Value>
                        <SolidColorBrush Color="Black"/>
                    </Setter.Value>
                </Setter>
                <Setter Property="Effect" >
                    <Setter.Value>
                        <DropShadowEffect BlurRadius="40"
                             Color="Maroon" Direction="50" Opacity="0.5"/>
                    </Setter.Value>
                </Setter>
            </Style>
        </ResourceDictionary>
    </Grid.Resources>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <TextBox Style="{StaticResource MyTextBoxStyle}" Grid.Row="0" />
    <TextBox Style="{StaticResource MyTextBoxStyle}" Grid.Row="1"
             Text="The Style is modified here"
             FlowDirection="LeftToRight"/>
</Grid>
```

So here I have shifted the Style into Resource section and used `MyTextBoxStyle` key to refer for each `TextBox` i defined. Notably, the style of both the textboxes remains same, while you can see I have also overridden certain settings in the control itself and it works the same. I have modified the `Text` of the 2nd TextBox to **"The Style is modified here"** and also made the `FlowDirection` to `LeftToRight`.

Another important thing, that you should always keep into mind, that if you do not define the Key element for the Style in Resource section, it will automatically be applied to all the `TextBox` you define.

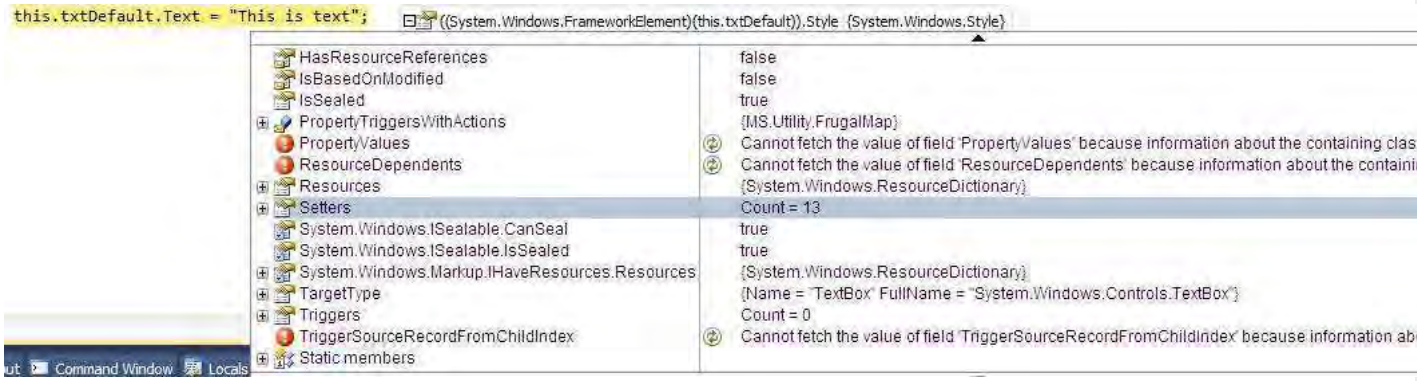
```
<Style TargetType="{x:Type TextBox}">
</Style>
```

Say the `style` you define does not contain any Key. So all the `TextBoxes` will automatically apply the style when appeared. You can eventually use

```
<TextBox Style="{x:Null}"/>
```

to revert the style.

Members of Style



The styling of WPF controls is made up with the help of a class called `Style`. The style object exposes few properties which help you to define various behavior. Lets look into the properties:

- **Resources** : It holds the reference for the `ResourceDictionary` where the Style is defined.
- **Setters** : It is a collection which holds all the `DependencyProperty` configuration for the whole control.
- **TargetType** : `TargetType` defines the type of the control for which the Style can be applied. So based on the `TargetType` the Style setters are defined to. So if you define a style for `TextBox` you cannot use Content as property `Setter`.
- **BasedOn** : This is used to allow `Style` inheritance. You can use an existing style key to inherit all the properties to a new `Style`.
- **Triggers** : A collection of Setters which would be applied based on certain conditions.

Using those properties you can define your own styles.

What about Explicit and Implicit Styles ?

WPF controls can have two type of styles associated with it. A control can have a style defined in the application and applied to its `Style` property. If your control is using a `Style` to define its look and feel or basically your control has set an object of Style into its Style property, then it is using an `Explicit` Style.

On the other hand, if your control takes the style from external environment (Theme) and the Style property is set to Null, then your control is using `Implicit` Style. Basically any WPF control automatically defines a DefaultStyle for it, so that you can set only the portion of the control which you need to change.

Say for instance, you have a `Button`. If you want to have its `Text` to be colored Red, you just need to change the Foreground of the `Button`. You need not to define the whole style. If there is no Default Style defined for Buttons, you need to define all the properties individually to make it appear. Thus the default color of the `Text` is Black if not defined otherwise.

Triggers

`Triggers` are a set of styles that work on a particular condition. You can think Trigger as a part of `Style` which will be set only when the Condition defined for the `Trigger` is met.

There are few types of Triggers :

1. **Property Trigger** : Will be set only when the `DependencyProperty` of a certain object has been set to a `Value`.
2. **Data Trigger** : Will work for any normal Properties using on `Binding`.
3. **Event Trigger** : Will work only when some event is triggered from the control.

Now to demonstrate let us look into the code below :

```
<Style TargetType="{x:Type TextBox}" x:Key="MyTextBoxStyle">
    <Setter Property="Text" Value="This is a TextBox with Styles"/>
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="CharacterCasing" Value="Lower"/>
    <Setter Property="FlowDirection" Value="RightToLeft"/>
    <Setter Property="FontSize" Value="20"/>
    <Setter Property="FontWeight" Value="UltraBlack"/>
    <Setter Property="Width" Value="400"/>
    <Setter Property="Height" Value="40"/>
    <Setter Property="Margin" Value="0,20,0,10" />
    <Setter Property="Background">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Color="Cyan" Offset="0.0"/>
                <GradientStop Color="Yellow" Offset="0.5"/>
                <GradientStop Color="Red" Offset="1.0"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="Foreground">
        <Setter.Value>
            <SolidColorBrush Color="Black"/>
        </Setter.Value>
    </Setter>
    <Setter Property="Effect" >
        <Setter.Value>
            <DropShadowEffect BlurRadius="40" Color="Maroon" Direction="50"
                              Opacity="0.5"/>
        </Setter.Value>
    </Setter>
    <Style.Triggers>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="Effect">
                <Setter.Value>
                    <DropShadowEffect BlurRadius="40" Color="Red"
                                      Direction="50" Opacity="0.9"/>
                </Setter.Value>
            </Setter>
        </Trigger>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property="IsFocused" Value="True"/>
                <Condition Property="IsMouseOver" Value="True"/>
            </MultiTrigger.Conditions>
            <Setter Property="Effect">
                <Setter.Value>
                    <DropShadowEffect BlurRadius="40" Color="Violet"
                                      Direction="50" Opacity="0.9"/>
                </Setter.Value>
            </Setter>
            <Setter Property="Foreground" Value="White" />
            <Setter Property="Background" Value="Maroon" />
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

Here you can see I have used Property `Trigger` to change the `DropShadowEffect` of TextBox when it is focussed. Every WPF control exposes few properties to work with Property Triggers, which will be set to true based on the control appearance changes. You can use these properties like `IsFocused, IsMouseDown` etc to work around with Property Triggers.

On the second occasion, I have defined a `MultiTrigger`. `MultiTrigger` allows you to mention Condition, so that when all the conditions of the `MultiTrigger` is met, the Property `Setters` for the object is applied.

So you can see when you hover your mouse over the `TextBox` and your textbox has its focus in it, only then you see the `TextBox` to appear in Maroon background and Violet `DropShadow` effect.

Animation Basics

Another interesting thing that you might think very interesting is the support of Animation for WPF. Basically, by the word Animation, we generally think of large Texture graphics in 3D space, which would probably be created in 3DS MAX studio or MAC etc. But believe me there is nothing to worry about this in case of WPF. WPF simplifies the concept Animation to be the change of a property over time.

Say for instance, say you want your `textbox` to change its color over time, you would write a simple color animation to do this or say you want to change the `Opacity` of a Border element during time, you need `DoubleAnimation` to do this. Animation is cool if you are clear about how it works.

Type of Animation

I must say, don't make yourself more confused by seeing the types of Animation. Animation is actually categorized in the same way as you categorize variables. Say for instance :

1. **DoubleAnimation** : This will animate a Double Value from one value to another. So if you want to change the Width of a `TextBox` over time you need `DoubleAnimation`.
2. **ColorAnimation** : Same as the above if the type of Changing element is Color, you need ColorAnimation.
3. `SingleAnimation, RectAnimation, PointAnimation, Int32Animaition, ThicknessAnimation` etc each of them bears the same meaning.

So basically the basis of Animation types is based on the type of the property for which you want your animation to work on.

Animation can also be categorized into two basic ways :

1. **Animation Without KeyFrames** : These are animation that only needs two values, From and To. It gives you a smooth animation based on the `Timeline.DesiredFramerate` property for the animation.
2. **Animation With KeyFrames** : Allows you to specify a `KeyFrame` collection which lets you define the KeyFrame value on a specified time. So that you can adjust your own animation based on specific time intervals.

Let us take a look at a few examples to make you understand animation feature of WPF:

```
<Window.Triggers>
        <EventTrigger RoutedEvent="Loaded">
            <BeginStoryboard>
                <Storyboard RepeatBehavior="Forever">
                    <DoubleAnimation Storyboard.TargetProperty="Width"
      From="300" To="200" AutoReverse="True" Duration="0:0:5" ></DoubleAnimation>
                    <DoubleAnimation Storyboard.TargetProperty="Height"
      From="300" To="200" AutoReverse="True" Duration="0:0:5"></DoubleAnimation>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Window.Triggers>
```

In the above code, I have defined an `EventTrigger` which lets you have a `DoubleAnimation`(as Width is double value) on `Width` of the `Window`. We use `Loaded` Event to start a `StoryBoard`.

What is a StoryBoard ?

A `StoryBoard` can be defined as a Container for `TimeLines` or a collection of animation timelines for which the object specified in `Target` will animate. We use `StoryBoard` to specify Animation within it.

Few important properties of `StoryBoard` :

- **RepeatBehaviour** : Specifies the number of times for which the `StoryBoard` repeat the animation.
- **Target** : Specifies the Target for which the `storyboard` will be applied to.
- **TargetName** : Defines the target and reference it by its name attribute.
- **TargetProperty** : Specifies the property for which the animation will be applied for.
- **AccelerationRatio / DecelerationRatio** : Defines the acceleration or deceleration for the animation.
- **AutoReverse** : Defines whether the `StoryBoard` will be reversed automatically. This is really cool concept, which allows you to get the reverse of the storyboard timeline automatically generated by the WPF.

Animation can also be applied from code.

```
DoubleAnimation myDoubleAnimation = new DoubleAnimation();
myDoubleAnimation.From = 1.0;
myDoubleAnimation.To = 0.0;
myDoubleAnimation.Duration = new Duration(TimeSpan.FromSeconds(5));
```

In the above code I have declared a `DoubleAnimation` which starts From 1.0 and moves to 0.0 in 5 seconds.

Animation with KeyFrames

Animation can be defined either using KeyFrames or without KeyFrames. KeyFrame allows you to define an intermediate frame so that the animation occurs for each individual frame intervals. There are three types of interpolation for an AnimationwithKeyFrames.

1. Linear
2. Discrete
3. Spline

Linear

Lets create an animation using KeyFrames :

```
<Border Background="Violet"
             HorizontalAlignment="Stretch"
             VerticalAlignment="Stretch" >
          <Border.Triggers>
              <EventTrigger RoutedEvent="Border.MouseLeftButtonDown">
                  <EventTrigger.Actions>
                      <BeginStoryboard>
                          <Storyboard>
                              <DoubleAnimationUsingKeyFrames
                                    Storyboard.TargetName="transformObj"
                                    Storyboard.TargetProperty="X"
                                    Duration="0:0:15">
                                  <LinearDoubleKeyFrame Value="500"
              KeyTime="0:0:3" />
                                  <LinearDoubleKeyFrame Value="50"
              KeyTime="0:0:7" />
                                  <LinearDoubleKeyFrame Value="300"
              KeyTime="0:0:13" />
                              </DoubleAnimationUsingKeyFrames>
                          </Storyboard>
                      </BeginStoryboard>
                  </EventTrigger.Actions>
              </EventTrigger>
          </Border.Triggers>
          <Border.RenderTransform>
              <TranslateTransform x:Name="transformObj" X="0" Y="0" />
          </Border.RenderTransform>
      </Border>
```

Here the animation is applied as LinearDoubleKeyFrame, which means the animation would be smooth while we define each KeyFrame value based on KeyTime. Here we change the Translation of the Border based on different KeyTime specified such that on 3rd second, the Rectangle will move to 500, at 7th second it will be at 50 and at 13th second it will be at 300. The animation is LinearDouble so the animation is smooth and steady.

Discrete

If I change the animation to DiscreteAnimation it will place the object only at the KeyTime specified

```
<DoubleAnimationUsingKeyFrames
        Storyboard.TargetName="transformObj"
        Storyboard.TargetProperty="X"
        Duration="0:0:15">
    <DiscreteDoubleKeyFrame Value="500" KeyTime="0:0:3" />
    <DiscreteDoubleKeyFrame Value="50" KeyTime="0:0:7" />
    <DiscreteDoubleKeyFrame Value="300" KeyTime="0:0:13" />
</DoubleAnimationUsingKeyFrames>
```

Thus changing the LinearDouble with DiscreteDouble makes it change its position all of a sudden based on the KeyTime specified for the animation.

Spline

SplineAnimation is used to define more realistic animation behavior for your control. It lets you control acceleration and deceleration of the animation. With KeySpline you can define the the cubic bazier curve using Spline Key frame. Lets look at the example

```
<DoubleAnimationUsingKeyFrames
        Storyboard.TargetName="transformObj"
        Storyboard.TargetProperty="X"
        Duration="0:0:15">
    <SplineDoubleKeyFrame Value="500" KeyTime="0:0:3" KeySpline="0.0,0.1 0.1,0.1" />
    <SplineDoubleKeyFrame Value="50" KeyTime="0:0:7" KeySpline="0.0,0.1 0.1,0.1"/>
    <SplineDoubleKeyFrame Value="300" KeyTime="0:0:13" KeySpline="0.0,0.1 0.1,0.1"/>
</DoubleAnimationUsingKeyFrames>
```

Thus you can see KeySpline allows you to define a smooth animation that starts with acceleration and with highest speed in the middle and ultimately decelerates back again.

You can also use EasingFunction to specify the custom formulas for the animation. For further information try : KeyFrameAnimation [^]

Conclusion

I hope you like my article. I am also happy that I have covered almost most of the topics that you need to know in WPF. There are still a few things left behind, which I would cover in my next article. Please put your valuable feedback for the article.

Thank you for reading.

License

About the Author

Abhishek Sur Did you like his post?