Dependency Properties

The heart of WPF

Dependency Properties are arguably the most important abstraction in the whole of WPF. There are not many parts of the framework that would work without dependency properties. For example the following technologies all use depency properties

1. Data Binding
2. Property value inheritance / Sparse storage
3. Styles – only dependency properties can be styled
4. Property Triggers

Change Notification

DependencyProperty change notification is not exactly as one might expect from a .Net API. There is no built in general purpose event based notification mechanism which which to register for notification of DependencyProperty change. When a dependency property takes a new value two mechanisms are notified

1. Data bindings mechanism is notified
2. Trigger mechanism notified

So if we want to respond to change in a dependency property we need to either bind to it (binding source) or use a trigger with it.

Where we register the Dependency property in our own code we can register a handler to be notified of changes. If we needed to we could then fire another event from handler to enable other clients to handle the change. The fact remains, however, that there is no default build in general purpose notification mechanism with DependencyProperties

The following XAML snippet shows the two build in ways to react to dependency property change

Listing 1DependencyProperty Change

```
<Window x:Class="Resolution.CodeSamples._1_Notification.ChangeNotify"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Height="300">

    <!-- If we want to react to a DependencyProperty change we have two choices. -->
    <!-- 1 - DataBind to it -->
    <TextBox Text="{Binding RelativeSource={RelativeSource Mode=Self},
Path=IsMouseOver,Mode=OneWay}">
        <TextBox.Style>
            <!-- 2. Define a trigger on it -->
            <Style TargetType="TextBox">
                <Style.Triggers>
                    <Trigger Property="IsMouseOver" Value="True">
                        <Setter Property="Background" Value="Red"></Setter>
                    </Trigger>
                </Style.Triggers>
            </Style>
        </TextBox.Style>
    </TextBox>
</Window>
```

Value Resolution

One of the most important features of DependencyProperties is that they evaluate their value dynamically. The base value of a DependencyProperty is resolved by evaluating a set of prioritised resoltuion items. The follow section shows the base value resolution items from lowest precedence to highest precedence.

1. ❶Default Value

The lowest precedence item is value specified in the dependency properties meta data provided at the point the property is registered

```
public static readonly DependencyProperty ValueProperty
   = DependencyProperty.RegisterAttached(
     "Value", typeof(string), typeof(StringRendererControl),
     new FrameworkPropertyMetadata("One",FrameworkPropertyMetadataOptions.Inherits));
```

2. ❷Inherited Value

A dependency property marked as `FrameworkPropertyMetadataOptions.Inherits` at the point it is registed can inherit its value from parent element in the visual tree.

```
    <controls03:StringRendererControl.Value>
        <system:String>Two</system:String>
    </controls03:StringRendererControl.Value>
```

3. ❸Default Theme Setter

A custom control’s default theme is the theme defined in Themes\Generic.xaml. Within the default theme values value lower precedence than triggers

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:basicScenario="clr-namespace:BasicScenario">
    <Style TargetType="basicScenario:StringRendererControl">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="basicScenario:StringRendererControl">
                    <Border Background="AliceBlue">
                        <TextBox Text="{TemplateBinding Value}"/>
                    </Border>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
        <Setter Property="Value" Value="Three"></Setter>
```

4. ❹Default Theme Trigger

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:basicScenario="clr-namespace:BasicScenario">
    <Style TargetType="basicScenario:StringRendererControl">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="basicScenario:StringRendererControl">
                    <Border Background="AliceBlue">
                        <TextBox Text="{TemplateBinding Value}"/>
                    </Border>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
        <Setter Property="Value" Value="Three"></Setter>
        <Style.Triggers>
            <Trigger Property="IsEnabled" Value="True">
                <Setter Property="Value" Value="Four"></Setter>
            </Trigger>
        </Style.Triggers>
    </Style>
```

5. ❺Style setters

A setter in a style other than the default theme style.

```
        <Window x:Class="BasicScenario.MainWindow"
                xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                xmlns:controls03="clr-namespace:BasicScenario"
                xmlns:system="clr-namespace:System;assembly=mscorlib"
                MouseEnter="MainWindow_OnMouseEnter"
                MouseLeave="MainWindow_OnMouseLeave"
                Title="MainWindow" SizeToContent="WidthAndHeight">

            <controls03:StringRendererControl.Value>
                <system:String>Two</system:String>
            </controls03:StringRendererControl.Value>

            <Window.Resources>
                <Style TargetType="controls03:StringRendererControl">
                    <Setter Property="Value" Value="Five"></Setter>
```

6.

7. ❻Template Triggers

Any trigger in a templates Triggers collection. The template can be either applied from a style or directly set on an element. Here we show one applied in a default style

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:basicScenario="clr-namespace:BasicScenario">
    <Style TargetType="basicScenario:StringRendererControl">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="basicScenario:StringRendererControl">
                    <Border Background="AliceBlue">
                        <TextBox Text="{TemplateBinding Value}"/>
                    </Border>
                     <ControlTemplate.Triggers>
                         <Trigger Property="IsEnabled" Value="True">
                             <Setter Property="Value" Value="Six"></Setter>
                         </Trigger>
```

8. ❼Style Triggers (Non default style)

This precdence applies to triggers in styles other than the default style (which has a lower precdence)

```
<Window x:Class="BasicScenario.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:controls03="clr-namespace:BasicScenario"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        MouseEnter="MainWindow_OnMouseEnter"
        MouseLeave="MainWindow_OnMouseLeave"
        Title="MainWindow" SizeToContent="WidthAndHeight">

    <controls03:StringRendererControl.Value>
        <system:String>Two</system:String>
    </controls03:StringRendererControl.Value>

    <Window.Resources>
        <Style TargetType="controls03:StringRendererControl">
            <Setter Property="Value" Value="Five"></Setter>
             <Style.Triggers>
                 <Trigger Property="IsEnabled" Value="True">
                     <Setter Property="Value" Value="Seven"></Setter>
                 </Trigger>
             </Style.Triggers>
```

9. ❽Implicit Style

Only applies to depdency properties whose type is a Style itself

10. ❾Template Parent Property Set

Only applies when the dependency property is being evaluated against an elemnt created as part of a teampate.

Where a control is created inside a template any property sets on that control within the template XML are not the same as set on the final instantiated control. This is because same piece of template XML can be used to instantiate multiple controls and each will end up with a differnent runtime property which can be further changed

```
<Window.Template>
    <ControlTemplate TargetType="controls03:MainWindow">
        <controls03:StringRendererControl Value="Nine" x:Name="MyIntegerControl"/>
    </ControlTemplate>
</Window.Template>
```

11. ❿Template Parent Trigger

Only applies when the dependency property is being evaluated against an elemnt created as part of a teampate.

```
<Window.Template>
    <ControlTemplate TargetType="controls03:MainWindow">
        <controls03:StringRendererControl Value="Nine" x:Name="MyIntegerControl"/>
        <ControlTemplate.Triggers>
            <Trigger Property="IsEnabled" Value="True">
                <Setter TargetName="MyIntegerControl"
                        Property="Value" Value="Ten"></Setter>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
</Window.Template>
```

12. ❶❶Local Value

Where the dependency property is being evaluated against an element that exists in the logical tree, i.e. one that is no created as part of another elements template, the local value is often just set via an attribute setter in the XAML If, however, the element in question was created as part of a template we have to do a bit more work. The following source code shows the work we have to do.

```
private void MainWindow_OnMouseEnter(object sender, MouseEventArgs e)
{
    StringRendererControl integerControl =
        (StringRendererControl)Template.FindName("MyIntegerControl", this);
    StringRendererControl.SetValue(integerControl, "Three");
}

private void MainWindow_OnMouseLeave(object sender, MouseEventArgs e)
{
    StringRendererControl integerControl =
        (StringRendererControl)Template.FindName("MyIntegerControl", this);
    integerControl.ClearValue(StringRendererControl.ValueProperty);
}
```

The following listings show the full source code for the above resolution sections

Listing 2Custom Control

```
public class StringRendererControl : Control
{
    public static readonly DependencyProperty ValueProperty
        = DependencyProperty.RegisterAttached(
        "Value", typeof(string), typeof(StringRendererControl),
        // 1. Default Value
        new FrameworkPropertyMetadata("One",FrameworkPropertyMetadataOptions.Inherits));

    public static void SetValue(DependencyObject el, string v) =>el.SetValue(ValueProperty, v);
    public static string GetValue(DependencyObject el) => (string) el.GetValue(ValueProperty);

    static StringRendererControl()
    {
        DefaultStyleKeyProperty.OverrideMetadata(
            typeof(StringRendererControl),
            new FrameworkPropertyMetadata(typeof(StringRendererControl)));
    }
}
```

Listing 3Themes/Generic.xaml

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:basicScenario="clr-namespace:BasicScenario">
    <Style TargetType="basicScenario:StringRendererControl">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="basicScenario:StringRendererControl">
                    <Border Background="AliceBlue">
                        <TextBox Text="{TemplateBinding Value}"/>
                    </Border>
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsEnabled" Value="True">
                            <Setter Property="Value" Value="Six"></Setter>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
        <Setter Property="Value" Value="Three"></Setter>
        <Style.Triggers>
            <Trigger Property="IsEnabled" Value="True">
                <Setter Property="Value" Value="Four"></Setter>
            </Trigger>
        </Style.Triggers>
    </Style>
</ResourceDictionary>
```

Listing 4MainWindow.xaml

```
<Window x:Class="BasicScenario.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:controls03="clr-namespace:BasicScenario"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        MouseEnter="MainWindow_OnMouseEnter"
        MouseLeave="MainWindow_OnMouseLeave"
        Title="MainWindow" SizeToContent="WidthAndHeight">

    <controls03:StringRendererControl.Value>
        <system:String>Two</system:String>
    </controls03:StringRendererControl.Value>

    <Window.Resources>
        <Style TargetType="controls03:StringRendererControl">
            <Setter Property="Value" Value="Five"></Setter>
            <Style.Triggers>
                <Trigger Property="IsEnabled" Value="True">
                    <Setter Property="Value" Value="Seven"></Setter>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

    <Window.Template>
        <ControlTemplate TargetType="controls03:MainWindow">
            <controls03:StringRendererControl Value="Nine" x:Name="MyIntegerControl"/>
            <ControlTemplate.Triggers>
                <Trigger Property="IsEnabled" Value="True">
                    <Setter TargetName="MyIntegerControl"
                            Property="Value" Value="Ten"></Setter>
                </Trigger>
            </ControlTemplate.Triggers>
        </ControlTemplate>
    </Window.Template>
</Window>
```

Listing 5MainWindow.xaml.cs

```
public partial class MainWindow
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void MainWindow_OnMouseEnter(object sender, MouseEventArgs e)
    {
        StringRendererControl integerControl =
            (StringRendererControl)Template.FindName("MyIntegerControl", this);
        StringRendererControl.SetValue(integerControl, "Three");
    }

    private void MainWindow_OnMouseLeave(object sender, MouseEventArgs e)
    {
        StringRendererControl integerControl =
            (StringRendererControl)Template.FindName("MyIntegerControl", this);
        integerControl.ClearValue(StringRendererControl.ValueProperty);
    }
}
```

Value Inheritance

In the following piece of code the FontStyle dependency property is set on the Window but because of property value inhertiance if flows all the way down the tree of FrameworkElements until it impacts the font used in the TextBoxes. Note that not all nodes in this tree are logical tree nodes but as they are all FrameworkElement objects they inherit values from other FrameworkElement parent ancestors.

Figure 1Value Inheritance

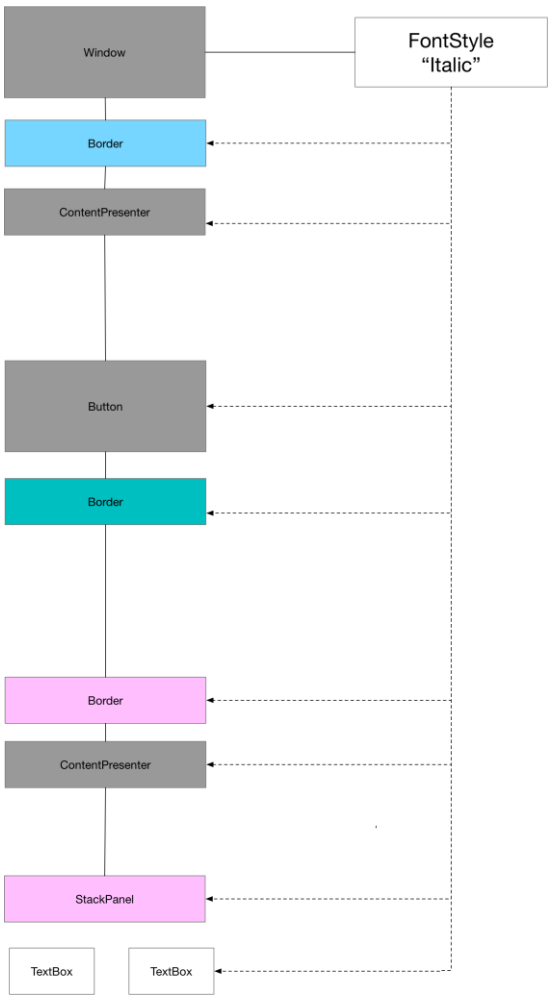
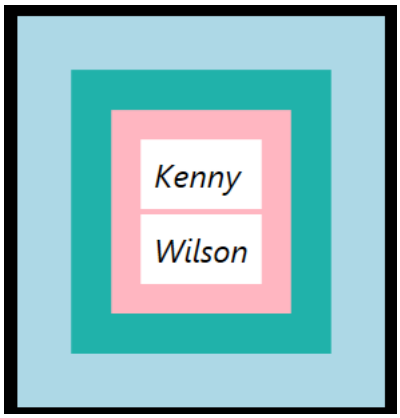


Figure 2Rendered Property Value Inheritance



The source code is as follows

Listing 6Value Inheritance Example 1

```
<Window x:Class="Controls_03.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:controls03="clr-namespace:Controls_03"
        xmlns:viewModels="clr-namespace:SharedResources.ViewModels;assembly=SharedResources"
        Background="LightBlue"
        Padding="10"
        FontStyle="Italic"
        Title="MainWindow" SizeToContent="WidthAndHeight">
    <Window.Resources>

        <DataTemplate DataType="{x:Type viewModels:Person}">
            <StackPanel>
                <TextBlock Background="White" Margin="1" Padding="5" Text="{Binding
FirstName}"></TextBlock>
                <TextBlock Background="White" Margin="1" Padding="5" Text="{Binding
SecondName}"></TextBlock>
            </StackPanel>
        </DataTemplate>

        <Style TargetType="Button">
            <Setter Property="Margin" Value="5"/>
            <Setter Property="Padding" Value="5"/>
            <Setter Property="Template" >
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Border Background="LightSeaGreen" Padding="5">
                            <ContentPresenter/>
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>

    <Window.Template>
        <ControlTemplate TargetType="{x:Type controls03:MainWindow}">
            <Border Background="{TemplateBinding Background}">
                <ContentPresenter></ContentPresenter>
            </Border>
        </ControlTemplate>
    </Window.Template>

    <Button Content="{viewModels:PersonMarkup FirstName=Kenny, SecondName=Wilson}">
        <Button.ContentTemplate>
            <DataTemplate>
                <Border Padding="5" Margin="5" Background="LightPink">
                    <ContentPresenter Content="{Binding}"></ContentPresenter>
                </Border>
            </DataTemplate>
        </Button.ContentTemplate>
    </Button>
</Window>
<Window x:Class="Controls_03.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:controls03="clr-namespace:Controls_03"
        xmlns:viewModels="clr-namespace:SharedResources.ViewModels;assembly=SharedResources"
        Background="LightBlue"
        Padding="10"
        FontStyle="Italic"
        Title="MainWindow" SizeToContent="WidthAndHeight">
    <Window.Resources>

        <DataTemplate DataType="{x:Type viewModels:Person}">
            <StackPanel>
                <TextBlock Background="White" Margin="1" Padding="5" Text="{Binding
FirstName}"></TextBlock>
```

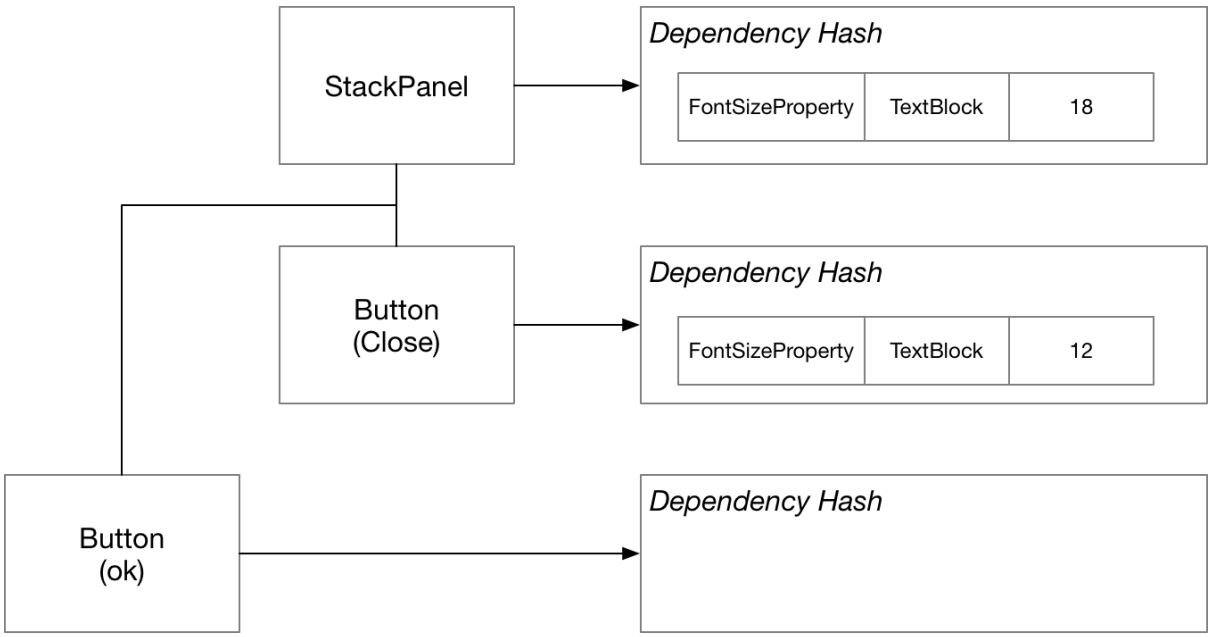
How Value Inheritance works

We can show how value inheritance works with another example. Consider the following XAML

```
<Window x:Class="ValueInheritance.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel TextBlock.FontSize="18">
        <Button>Ok</Button>
        <Button TextBlock.FontSize="12">Close</Button>
    </StackPanel>
</Window>
```

Logically our visual tree is as follows. The panel has two child button elements. If the dynamic resolution of the property traverses as follows. The call for the OK button will first check the OK buttons dependency hash and notice there is no value for FontSize. At this stage it will walk back up the element tree to the parent stackpanel. The stackpanel dependency hash has an entry and so the OK button's text is rendered with a FontSize of 18

Contrast this with the call for the close buttons fontsize. As the close buttons dependency hash has a value for the fontsize then no further traversal of the tree of hashes is necessary



Listing 7Value Inhertiance XAML

```
<Window x:Class="Resolution.CodeSamples._2_ValueResolution.ValueResolutionWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:views1="clr-
namespace:RiskAndPricingSolutions.WPF.SharedResources.Views;assembly=SharedResources"
        WindowStyle="None"
        ResizeMode="NoResize"
        mc:Ignorable="d"
        Title="ValueResolutionWindow" Height="300" Width="300">
    <StackPanel Margin="10" Orientation="Vertical">
        <!-- 1. Default value -->
        <views1:SimpleRectangle/>
        <StackPanel Orientation="Horizontal" views1:SimpleRectangle.Fill="Blue">
            <!-- 2. Inherited Value -->
            <views1:SimpleRectangle Margin="10"/>

            <views1:SimpleRectangle Margin="10">
                <views1:SimpleRectangle.Style>
                    <Style>
                        <!-- 3. Style Setter -->
                        <Setter Property="views1:SimpleRectangle.Fill" Value="Red"/>
                        <!-- 4. Trigger Setter -->
                        <Style.Triggers>
                            <Trigger Property="views1:SimpleRectangle.IsMouseOver" Value="True">
                                <Setter Property="views1:SimpleRectangle.Fill" Value="Black"/>
                            </Trigger>
                        </Style.Triggers>
                    </Style>
                </views1:SimpleRectangle.Style>
             </views1:SimpleRectangle>

            <!-- 5. Local Value -->
        </StackPanel>
        <views1:SimpleRectangle Margin="10"/>
    </StackPanel>
</Window>
```

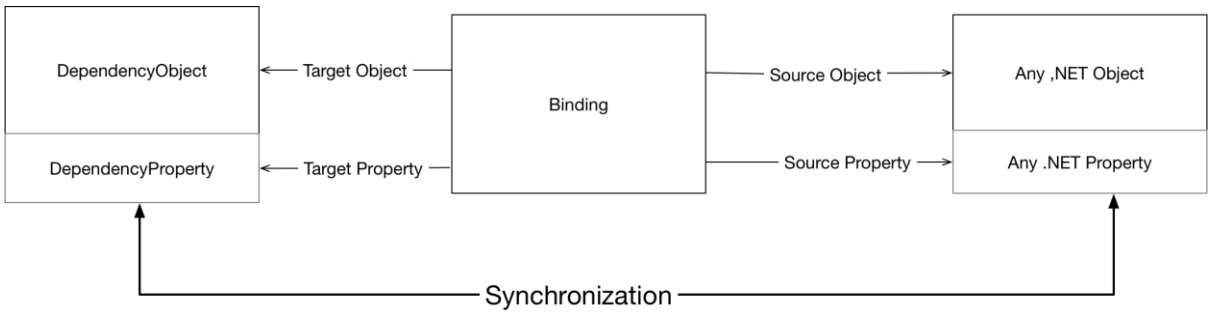
AttachedProperty

In the previous section we saw how dependency property value inheritance is implemented as a hierarchy of hashes. We can attach property values to dependency objects whose actual type does not support that property. This then enables us to to specify values on parents that will filter down to child controls where appropriate. This mechanism is known as attached properties. Attached properties allow one to add property values to hashmaps of root elements which don't understand what to do with that value only so that leaf nodes can share a common price of state.

Data Binding

A Binding keeps source and target properties in sync. The target property must be a dependency property and hence the target object must be a dependency object. The source property can be any .net property, however if we want the target to respond to changes in the source then the source object's type should implement `INotifyPropertyChanged`

Figure 3Binding Object



BINDING

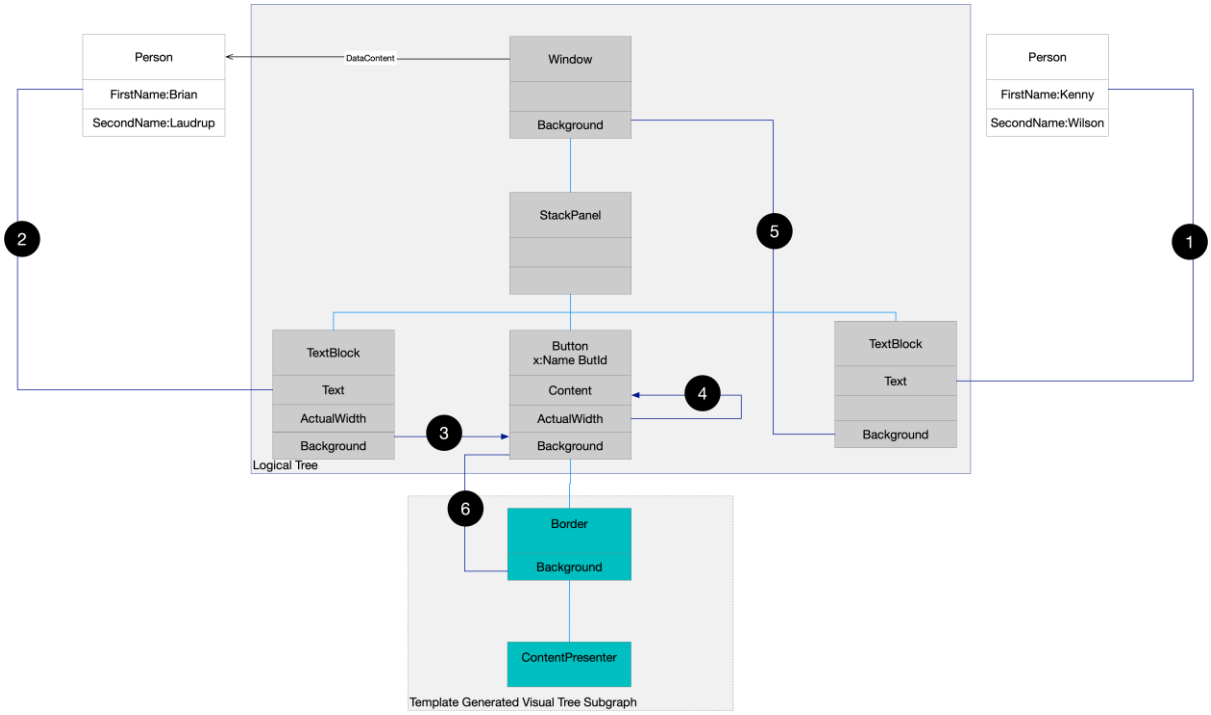
❶ **Target Object**	Must be a DependencyObject
❷ **Target Property**	Must be DependencyProperty
❸ **Source Object**	Any .NET object but must implement INotifyPropertyChanged if we want change propagation
❹ **Source Property**	Any .NET property but again must raise INotifyPropertyChanged if we want change notification. Must also be a property and not just a field.

The following code shows how two wire up source and target properties in XAML

```
<Window x:Class="DataBinding_01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:viewModels="clr-
namespace:SharedResources.ViewModels;assembly=SharedResources"
        Title="MainMyWindow" Height="70" Width="100">

    <Window.Resources>
        <viewModels:Person x:Key="Person" FirstName="Kenny" SecondName="Wilson"/>
    </Window.Resources>

    <StackPanel>
        <TextBox Text="{Binding Source={StaticResource Person}, Path=FirstName}"/>
    </StackPanel>
</Window>
```

Specifying the Binding Source

Figure 4Binding Source



KEY

❶ Explicit Source

```
{Binding  Source={StaticResource APerson},
Path=FirstName}
```

❷ DateContext (Implicit Source)

```
{Binding  Path=FirstName}
```

❸ Element Name

```
{Binding ElementName=ButId, Path=Background}
```

❹ RelativeSource Self

```
{Binding RelativeSource={RelativeSource Self},
Path=ActualWidth}
```

❺RelativeSource AncestorType

```
{Binding RelativeSource={RelativeSource
Mode=FindAncestor,AncestorType=Window},Path=Background}
```

❻RelativeSource TemplatedParent

```
{Binding RelativeSource={RelativeSource
TemplatedParent},Path=Background}
```

Listing 8DataBindng Source

```
    <Window.DataContext>
        <viewModels:Person FirstName="Brian" SecondName="Laudrup"/>
    </Window.DataContext>

    <Window.Resources>
        <viewModels:Person x:Key="APerson" FirstName="Kenny" SecondName="Wilson"/>
        <Style TargetType="Button">
            <Style.Setters>
                <Setter Property="Template">
                    <Setter.Value>
                        <ControlTemplate TargetType="Button">
                            <Border Background="{Binding RelativeSource={RelativeSource
TemplatedParent},Path=Background}">
                                <ContentPresenter></ContentPresenter>
                            </Border>
                        </ControlTemplate>
                    </Setter.Value>
                </Setter>
            </Style.Setters>
        </Style>
    </Window.Resources>

    <StackPanel Background="LightBlue" Margin="20">
        <!-- 1. Absolute Reference
             3. Element Name -->
        <TextBlock
            Text="{Binding Source={StaticResource APerson}, Path=FirstName}"
            Background="{Binding ElementName=ButId, Path=Background}"/>

        <!-- 4. RelativeSource Self -->
        <Button x:Name="ButId" Content="{Binding RelativeSource={RelativeSource Self},
Path=ActualWidth}"></Button>

        <!-- 2. Implicit Source (DataContext) -->
        <TextBlock Text="{Binding  Path=FirstName}"/>

        <!-- 5. RelativeSource AncestorType-->
        <TextBlock Background="{Binding RelativeSource={RelativeSource
Mode=FindAncestor,AncestorType=Window},Path=Background}"
                   Text="SomeText"/>
    </StackPanel>
```

In the previous section we explicitly set the binding source via a static reference to an object defined in the XAML resources. Rather than explicitly setting the binding source in XAML we can take advantage of what is known as the **DataContext.** Every FrameworkElement object has a DependencyProperty called DataContext which can be used to set an implicit source for data bindings. Once a DataContext has been set on a FrameworkElement then any bindings that target that element's dependency properties will pick use the DataContext as the binding source implicitly.The killer feature of the DataContext property is that it is inheritable. The binding infrastructure will traverse the tree of elements until it finds a DataContext. So we can set a single DataContext on an ancestor and all descendents who do not explicitly override it will have access to the same binding source.

Binding Details

Mode Types

OneWay	The target is updates to reflect changes in the source
OneWayToSource	The source is updated to reflect changes in the target
OneTime	Same as OneWay but the target is only updated once when the binding is initialized
Two	Changes from source reflected in target and vice versa

One way binding is the default for most dependency properties. TextBox.Text is the classic two way binding depenedency property. And of course cells in data grids will want to be two way bound

In order for two way and one way to source to work the source property must have an applicable public setter.

UpdateSourcetrigger

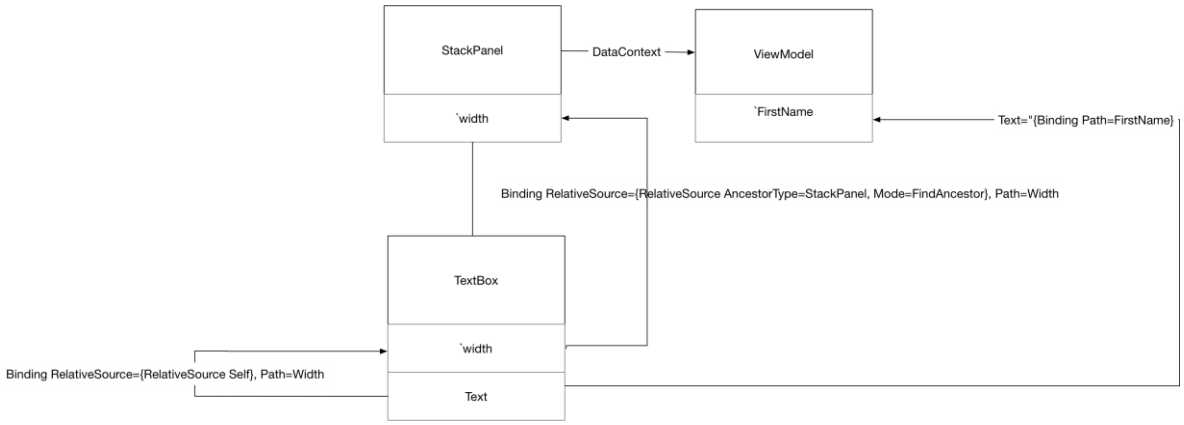
PropertyChanged	Source updated whenever the property changes
LostFocus	Source only updated when the target element loses focus
Explcit	Source updated when you call BindingExpression.UpdateSource

DataCoversion

If we are using a DataConverter the Convert method is using when updating the target to reflect changes in the source and ConvertBack is invoked when updating the source to relect changes in the target

Target Object as source

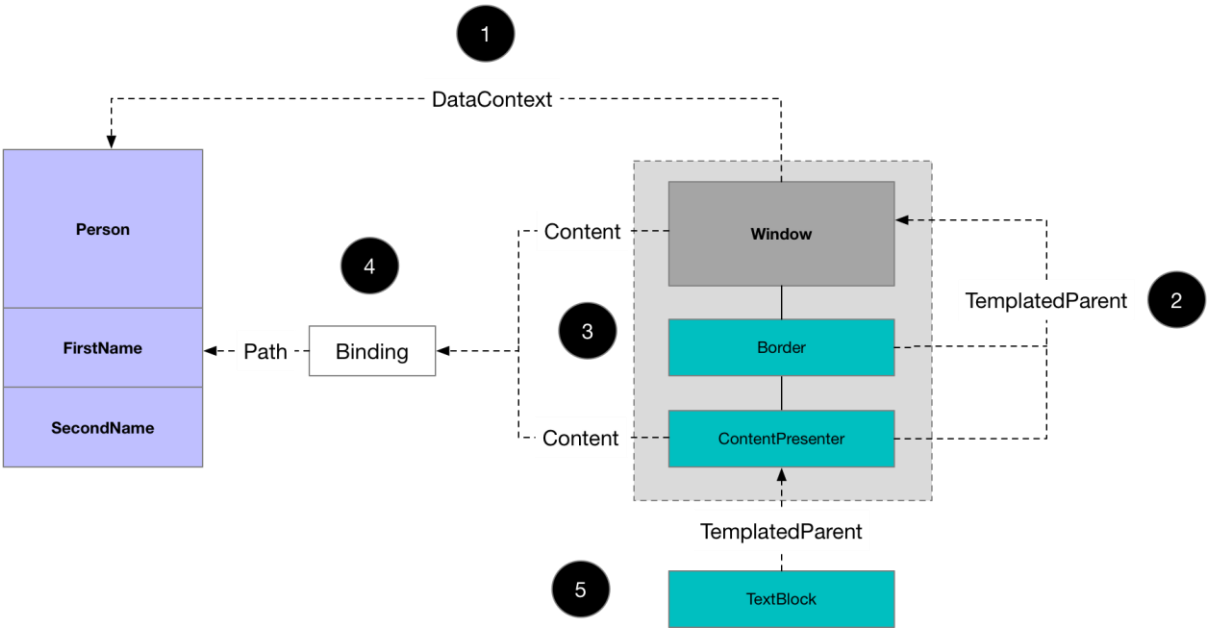
There are different ways of specifying the binding source as the following diagram and piece of XAML highlight



```
<Window x:Class="DataBinding_01.SpecifyingBindingSource"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:viewModels="clr-namespace:SharedResources.ViewModels;assembly=SharedResources"
        mc:Ignorable="d"
        Title="SpecifyingBindingSource" Height="300" Width="300">
    <StackPanel Width ="200" Height="200" Background="LightGreen">
        <StackPanel.DataContext>
            <viewModels:Person FirstName="Kenny"></viewModels:Person>
        </StackPanel.DataContext>
        <TextBox  Width="100" Text="{Binding Path=FirstName}"></TextBox>
        <TextBox Width="100" Text="{Binding RelativeSource={RelativeSource Self}, Path=Width}"/>
        <TextBox Width="100" Text="{Binding RelativeSource={RelativeSource AncestorType=StackPanel,
Mode=FindAncestor}, Path=Width}"/>
        <TextBox Width="100" Text="{Binding RelativeSource={RelativeSource AncestorType=Window,
Mode=FindAncestor}, Path=Width}"/>
    </StackPanel>
</Window>
```

Dependency Properties and Control Templates (1)

```
<Window x:Class="Controls_04.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:viewModels="clr-namespace:SharedResources.ViewModels;assembly=SharedResources"
        xmlns:controls04="clr-namespace:Controls_04"
        Background="LightBlue"
        Padding="10"
        FontSize="28"
        Title="MainWindow" SizeToContent="WidthAndHeight">

    <!-- Set the DataContent on the Window -->
    <Window.DataContext>
        <viewModels:Person FirstName="Kenny" SecondName="Wilson"></viewModels:Person>
    </Window.DataContext>

    <!-- Define a ControlTemplate-->
    <Window.Template>
        <ControlTemplate TargetType="{x:Type controls04:MainWindow}">
            <!-- Background does not flow down the Logical/Visual tree so we need explicitly bind to
it-->
            <Border Background="{Binding RelativeSource={RelativeSource TemplatedParent},
Path=Background}">
                <ContentPresenter  />
            </Border>
        </ControlTemplate>
    </Window.Template>

    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Border Background="LightGreen" >
                            <ContentPresenter Margin="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Padding}" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>

    <Window.Content>
        <Binding Path="FirstName"></Binding>
    </Window.Content>

</Window>
```

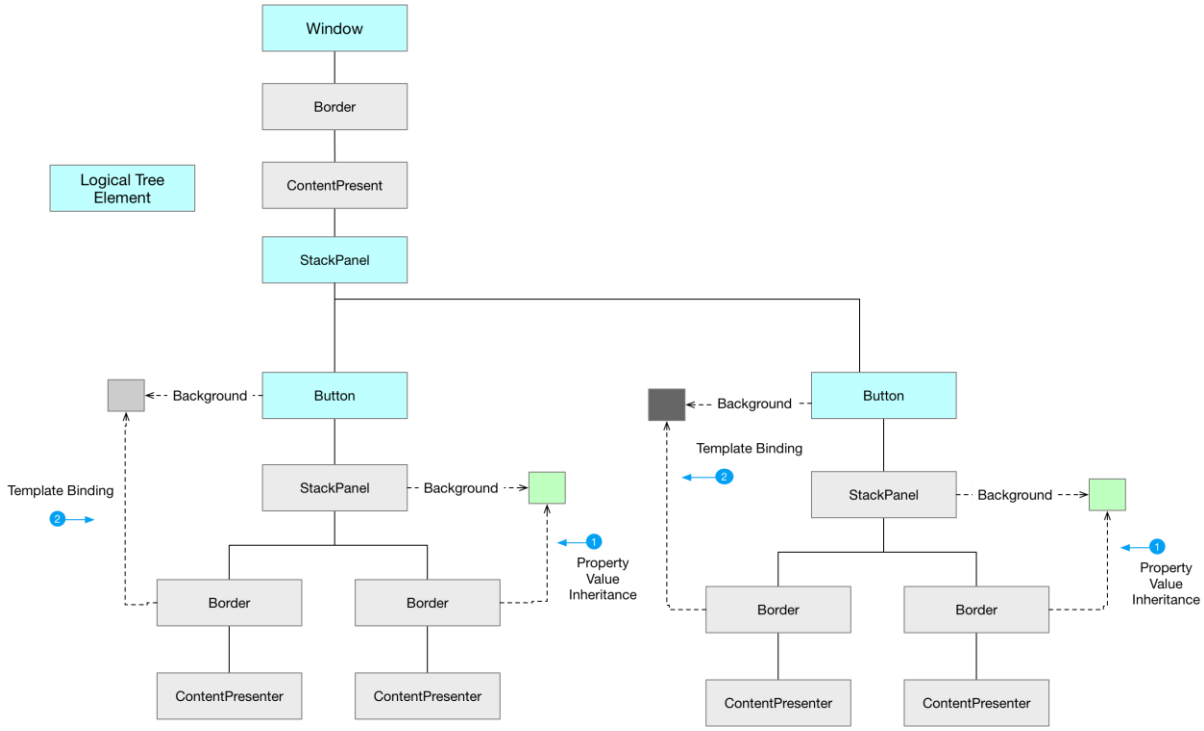


1. The window sets its DataContextx to an instance of type Person, a non visual element. As DataContext is an inheritable DependencyProperty it is inherited by the windows decedents in the visual tree.
2. The elements generated by the Window's Template can reference their containing Control. This reference is known as the TemplatedParent
3. A ContentPresenter, instantiated as part of a Control's ControlTemplate, implicitly uses the TemplatedParent Content property as its own Content.
4. The windows Content property value is set to a Binding. The binding only sets the Path property thereby implicitly using the DataSource as the Binding source
5. The TextBlock is generated from a template that is used to render strings. As such its TemplatedParent is the ContentPresenter

Dependency Properties and Control Templates (2)

The following code shows how dependency properties use a sparse storage system whereby values are inherited fro parent objects in the visual tree. It also shows how a visual element in a templarte can access the value of its template parent rather than its actual parent in the template.

```
<Window x:Class="AndControlTemplates.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Title="MainWindow" SizeToContent="WidthAndHeight">

    <Window.Template>
        <ControlTemplate TargetType="Window">
            <Border>
                <ContentPresenter></ContentPresenter>
            </Border>
        </ControlTemplate>
    </Window.Template>

    <Window.Resources>
        <system:String x:Key="String">Kenny</system:String>
        <ControlTemplate x:Key="ButtonControlTemplate" TargetType="Button">

            <!-- 1a. Set the Background DependencyProperty on the StackPanel -->
            <StackPanel Background="LightGreen">

                <!-- 1b.Border inherits the value of Background from its Stack Parent
                in this ControlTemplate -->
                <Border>
                    <ContentPresenter></ContentPresenter>
                </Border>

                <!-- 2b. Take the value of background from the templated parent using
this template -->
                <Border Background="{TemplateBinding Background}">
                    <ContentPresenter></ContentPresenter>
                </Border>

            </StackPanel>
        </ControlTemplate>
    </Window.Resources>

    <StackPanel>
        <!-- 2a. Set the value on the templated parent -->
        <Button Margin="10" Background="LightGray"
                Template="{StaticResource ButtonControlTemplate}">First Button</Button>
        <!-- 2a. Set the value on the templated parent -->
        <Button Margin="10"
                Background="DarkGray" Template="{StaticResource ButtonControlTem-
plate}">First Button</Button>
    </StackPanel>
</Window>
```



1. Simple property value inheritance used to inheri values from the parent elments in the visual tree.
2. Use the TemplateBinding to access values from the TemplatesParent. In this case the Border's TemplatedParent is the Button

Binding to Collection

- INotifycollectionchanged
- Inotifypropertychanged
- Observablecollection and built in change notification
- Items controls and display member path
- Selectors e.g. ListBox
- IsSynchronizedWithCurrentItem (not support multiple selections)

Format Strings to format target properties

When the target property is a string we can use the StringFormat property to tell XAML how to convert a source object of various types to a string. The following shows how to convert display a double with some text around it.

```
<Window x:Class="DataBinding_01.FormatStrings"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Title="FormatStrings" Height="300" Width="300">
    <Grid>
        <StackPanel>
            <StackPanel.Resources>
                <system:Double x:Key="Num">3.145</system:Double>
            </StackPanel.Resources>
            <TextBox Text="{Binding Source={StaticResource Num}, Mode=OneWay,
            StringFormat=This is a double {0}}"/>

            <!-- If the part that reference the object is the first thing
            in the format string  we need to escape it with {}to prevent
            XAML treating it as MarkupExtension syntax. -->
            <TextBox Text="{Binding Source={StaticResource Num},
                Mode=OneWay,StringFormat={}{0} is a double}"/>
        </StackPanel>

    </Grid>
</Window>
```

If the part that reference the object is the first thing in the format string we need to escape it with {} to prevent XAML treating it as MarkupExtension syntax (see above example)

A Databinding puzzle

Attach a dependency property to a window and set its value to be a binding with implicit source (data context). Then set a data context in the window Now set the windows content to be any object that uses the attached property and also set the object to have a different datacontext from the window. The object will use the datacontext from the window and not the object

ToDo

X:reference markup extension

Coding a Dependency Property

1. Add a static instance with ‘Property’ prefix

```
public static readonly DependencyProperty ContentStringProperty;
```

2. Register using the Register method and handle change

```
static StringControl()
{
    ContentStringProperty =
        DependencyProperty.Register("ContentString",
        typeof(string), typeof(StringControl),
        new FrameworkPropertyMetadata(HandlePropertyChanged));
}

private static void HandlePropertyChanged(DependencyObject dob,
    DependencyPropertyChangedEventArgs ea)
{
   // Deal with any changes
}
```

3. Add standard .NET property wrapper

```
public string ContentString
{
    get => GetValue(ContentStringProperty) as string;
    set => SetValue(ContentStringProperty, value);
}
```

Coding an Attached Dependency Property

1. Add a static instance with 'Property' prefix

```
public static readonly DependencyProperty ContentStringProperty;
```

2. Register using the RegisterAttached method and handle change

```
static StringControl()
{
    var md = new FrameworkPropertyMetadata("",
        FrameworkPropertyMetadataOptions.Inherits,
        HandlePropertyChanged);

    ContentStringProperty =
        DependencyProperty.RegisterAttached("ContentString",
        typeof(string), typeof(StringControl), md);
}

private static void HandlePropertyChanged(DependencyObject dob,
    DependencyPropertyChangedEventArgs args)
{
    // Handle change
}
```

3. Add standard .NET property wrapper

```
public string ContentString
{
    get => GetValue(ContentStringProperty) as string;
    set => SetValue(ContentStringProperty, value);
}
```

4. Add static get/set methods

```
public static void SetContentString(DependencyObject dob, string value)
    => dob.SetValue(ContentStringProperty, value);
public static string GetContentString(DependencyObject dob)
    => dob.GetValue(ContentStringProperty) as string;
```

Questions

Compare Dependency Properties to standard .NET properties

1. *DependencyProperty value is resolved dynimically when GetValue is called. Standard .NET prorties are usually backed by a filed in a class*

2. *DependencyProperties have build in change notification*

How can one react to changes in a DependencyProperty (3 ways)

1. *DataBind to it. The dependency property is the binding source*
2. *Define a trigger on it*
3. *If we own the property register a value changed callback on it*

List the order of resolution of DP value providers from lowest to highest for element in the Logical Tree (Not as part of a template)

1. Default Value
2. Inherited Value
3. Default Style Value
4. Default Style Trigger
5. Non Default Style Value
6. Template Trigger (Targeting element on which we evaluate the DP)
7. Non Default Style Trigger
8. Local Value

List the order of resolution of DP value providers from lowest to highest for element not in the logical tree (Template)

1. Default Value
2. Inherited Value
3. Default Style Value
4. Default Style Trigger
5. Non Default Style Value
6. Template Trigger (Targeting element on which we evaluate the DP)
7. Non Default Style Trigger
8. Templated Parent Template Value
9. Templated Parent Template Trigger
10. Local Value

What are the benefits of attached dependency properties

They allow leaf nodes to share a value set on a parent even though the parent does not undertand or use that property.

Why should you provide a property wrapper around GetValue/SetValue

To make the code more readable

To enable the value to be set from XAML

Why should a property wrapper not contain custom logic?

Because at runtime WPF will call SetValue directlry

Where should custom logic be implemented?

Use the registered callback

Data Binding

What 4 components does each data binding require?

Target Object

Target property

Source object

Source property

What kind of object must a binding target be?

The binding target must be a dependency object

What constraint is there on the binding target property?

The binding targets property must be a dependency property

What constraints are on the binding source object?

None. The binding source can be any ,NET property. If we want to receive change notifications however it needs to implement INotifyPropertyChanged

What constraints are there on the binding source property?

It can be any CLR or dependency property on the binding source