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 Dependency 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 with 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 the 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



## 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 resolution items. The follow section shows the base value resolution items from lowest precedence to highest precedence.

1. ➊Default Value

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



1. ➋Inherited 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.



1. ➌Default 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



1. ➍Default Theme Trigger



1. ➎Style setters

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



1. ➏Template 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



1. ➐Style Triggers (Non default style)

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



1. ➑Implicit Style

Only applies to depdency properties whose type is a Style itself

1. ➒Template 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



1. ➓Template Parent Trigger

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



1. ➊➊Local 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.



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

Listing 2Custom Control



Listing 3Themes/Generic.xaml



Listing 4MainWindow.xaml



Listing 5MainWindow.xaml.cs



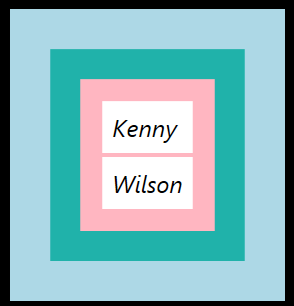
## Value Inheritance

In the following piece of code the FontStyle dependency property is set on the Window but because of property value inheritance 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



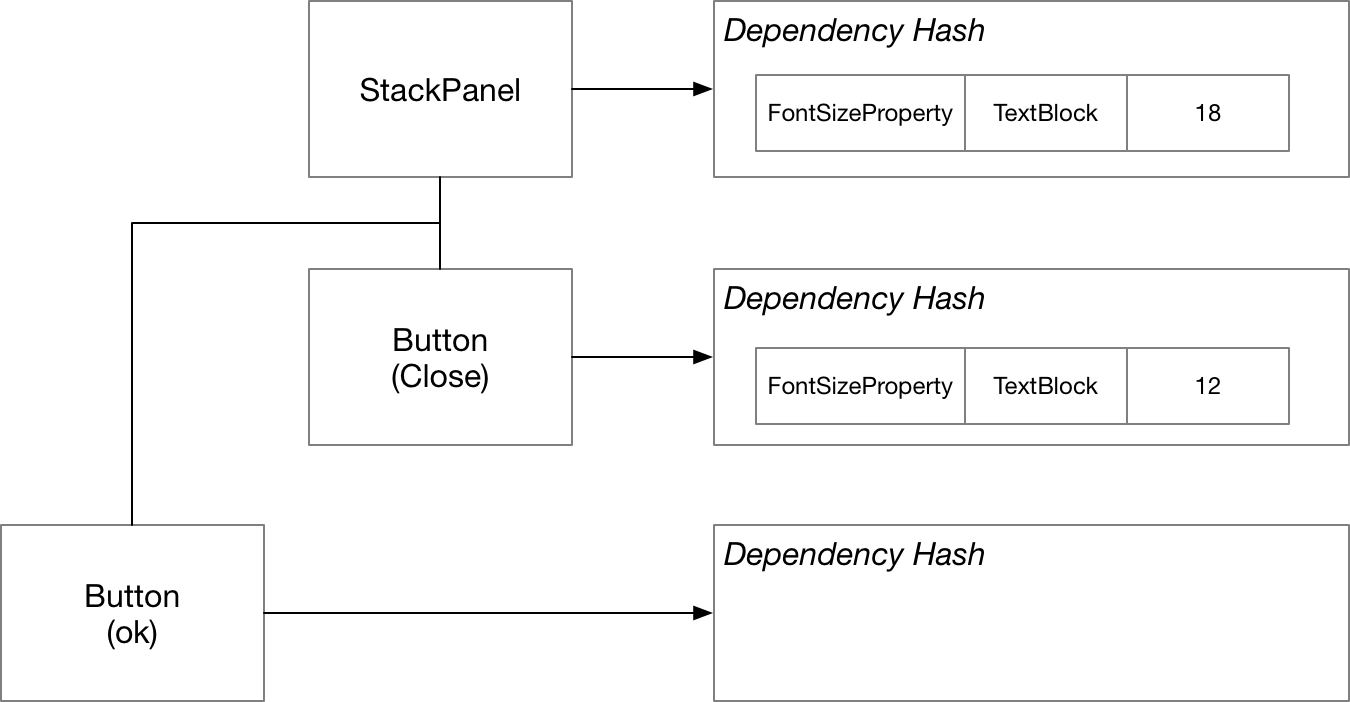
### How Value Inheritance works

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



Logically our tree is as follows. The panel has two child button elements. 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 Inheritance XAML



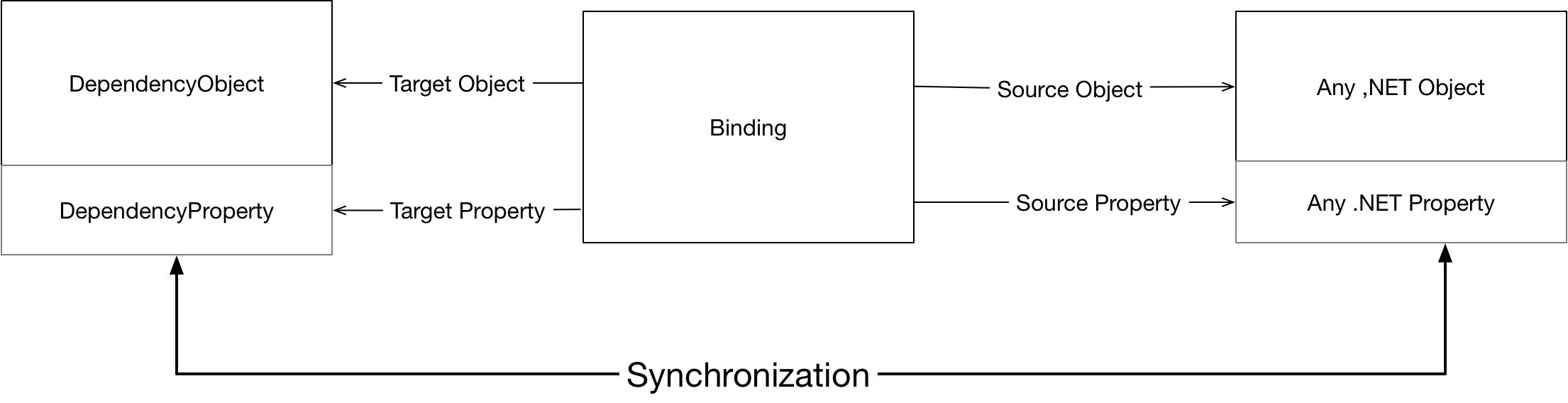
### AttachedProperty

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 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 hash maps 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 DependencyObject. 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

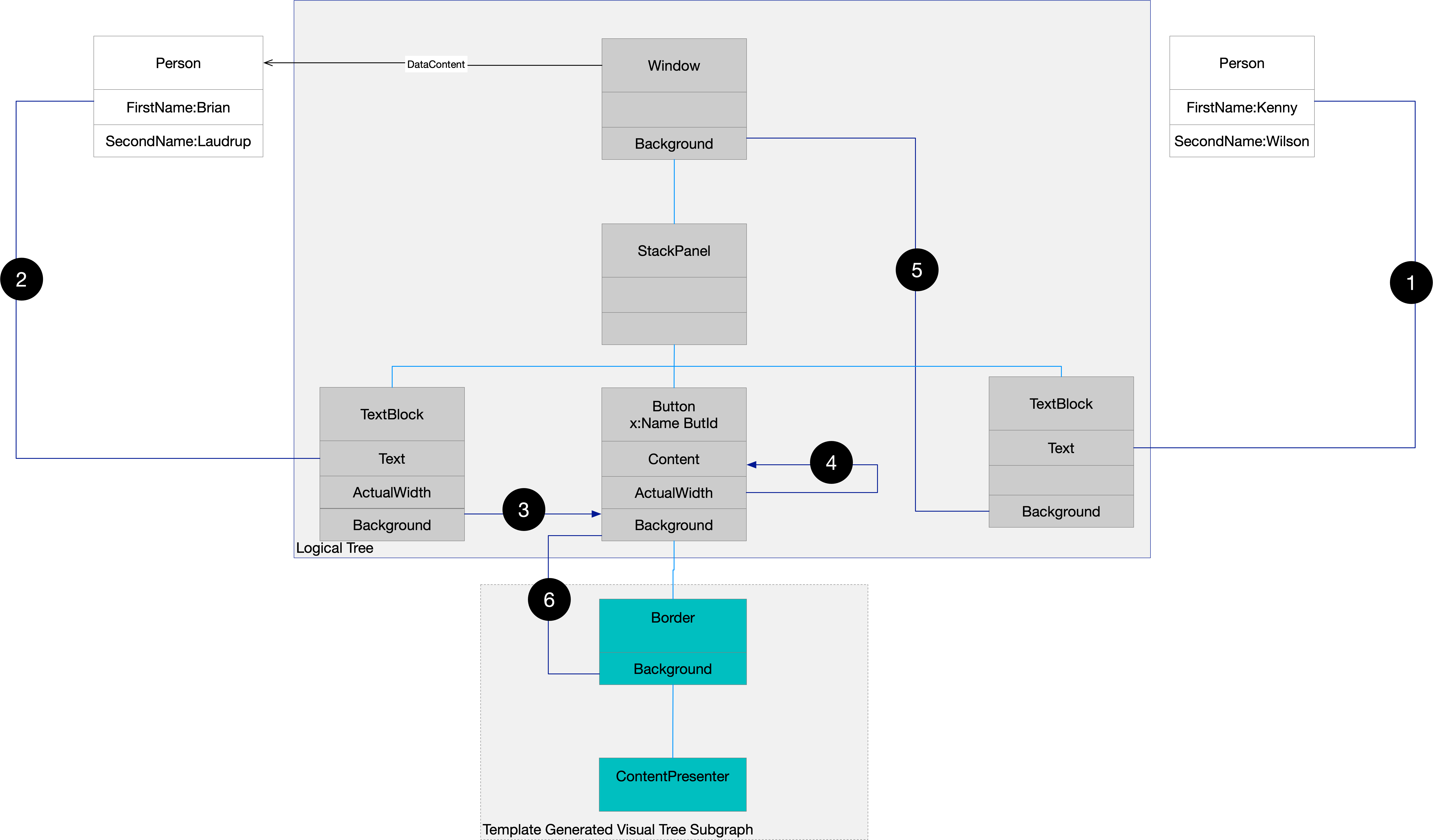
|  |  |
| --- | --- |
| ➊ Target Object | Must be a DependencyObject |
| ➋ Target Property | Must be DependencyProperty |
| ➌ Source Object | Any .NET object but must implement INotifyPropertyChanged if we want change propagation |
| ➍ Source Property | Any .NET property but again must raise INotifyPropertyChanged 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



### Specifying the Binding Source

Figure Binding Source



Key

|  |  |
| --- | --- |
| ➊ Explicit Source | {Binding Source={StaticResource APerson}, Path=FirstName} |
| ➋ DataContext (Implicit Source) | {Binding Path=FirstName} |
| ➌ Element Name | {Binding ElementName=ButId, Path=Background} |
| ➍ RelativeSource Self | {Binding RelativeSource={RelativeSource Self}, Path=ActualWidth} |
| ➎RelativeSource AncestorType | {Binding RelativeSource={RelativeSource Mode=FindAncestor,AncestorType=Window},Path=Background} |
| ➏RelativeSource TemplatedParent | {Binding RelativeSource={RelativeSource TemplatedParent},Path=Background} |

Listing DataBindng Source



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 up 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 descendants who do not explicitly override it will have access to the same binding source.

### Binding Details

Mode Types

|  |  |
| --- | --- |
| OneWay | The target is updated 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 dependency 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

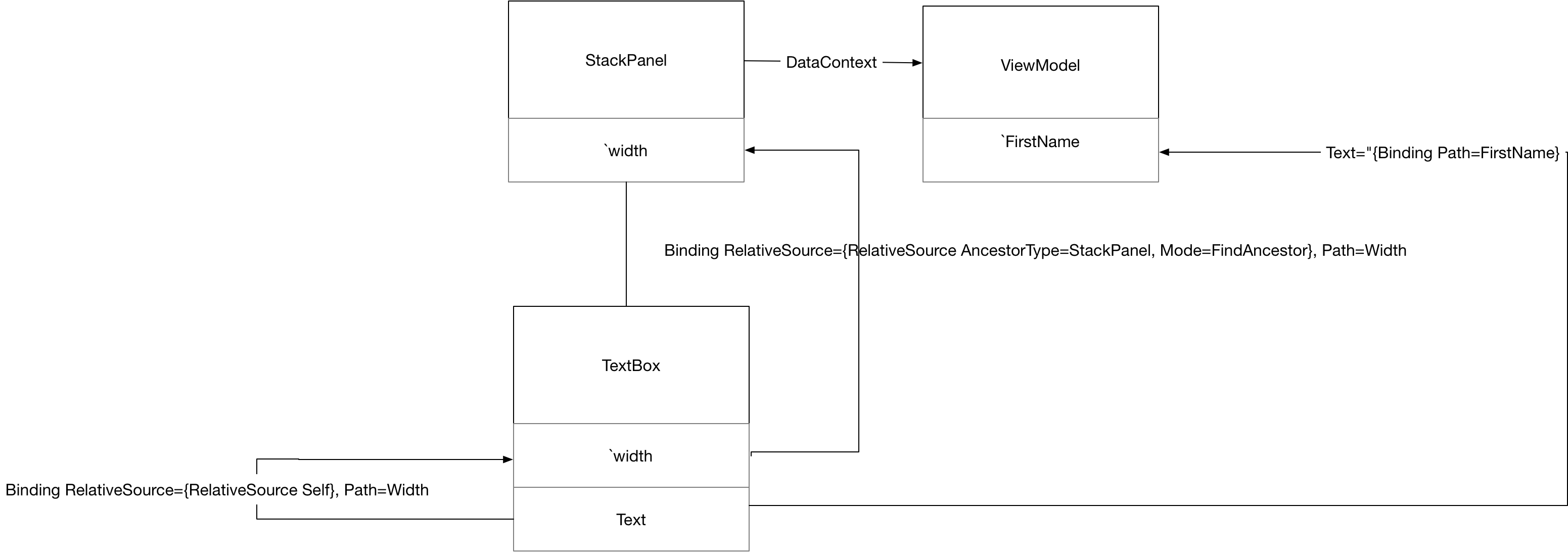
|  |  |
| --- | --- |
| PropertyChanged | Source updated whenever the property changes |
| LostFocus | Source only updated when the target element loses focus |
| Explcit | Source updated when you call BindingExpression.UpdateSource |

### Data Conversion

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 reflect changes in the target

### Target Object as source

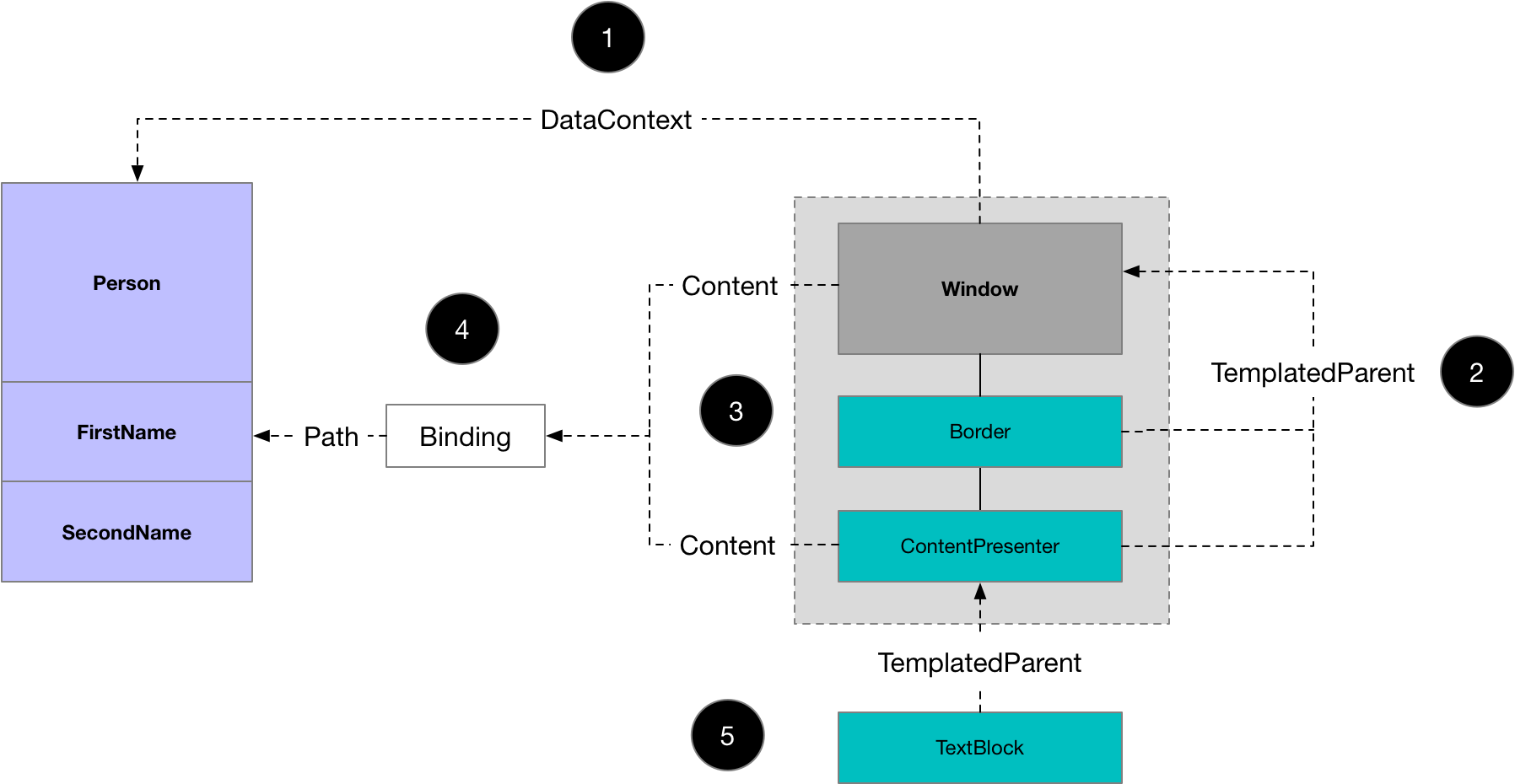
There are different ways of specifying the binding source as the following diagram and piece of XAML highlight





### Dependency Properties and Control Templates (1)



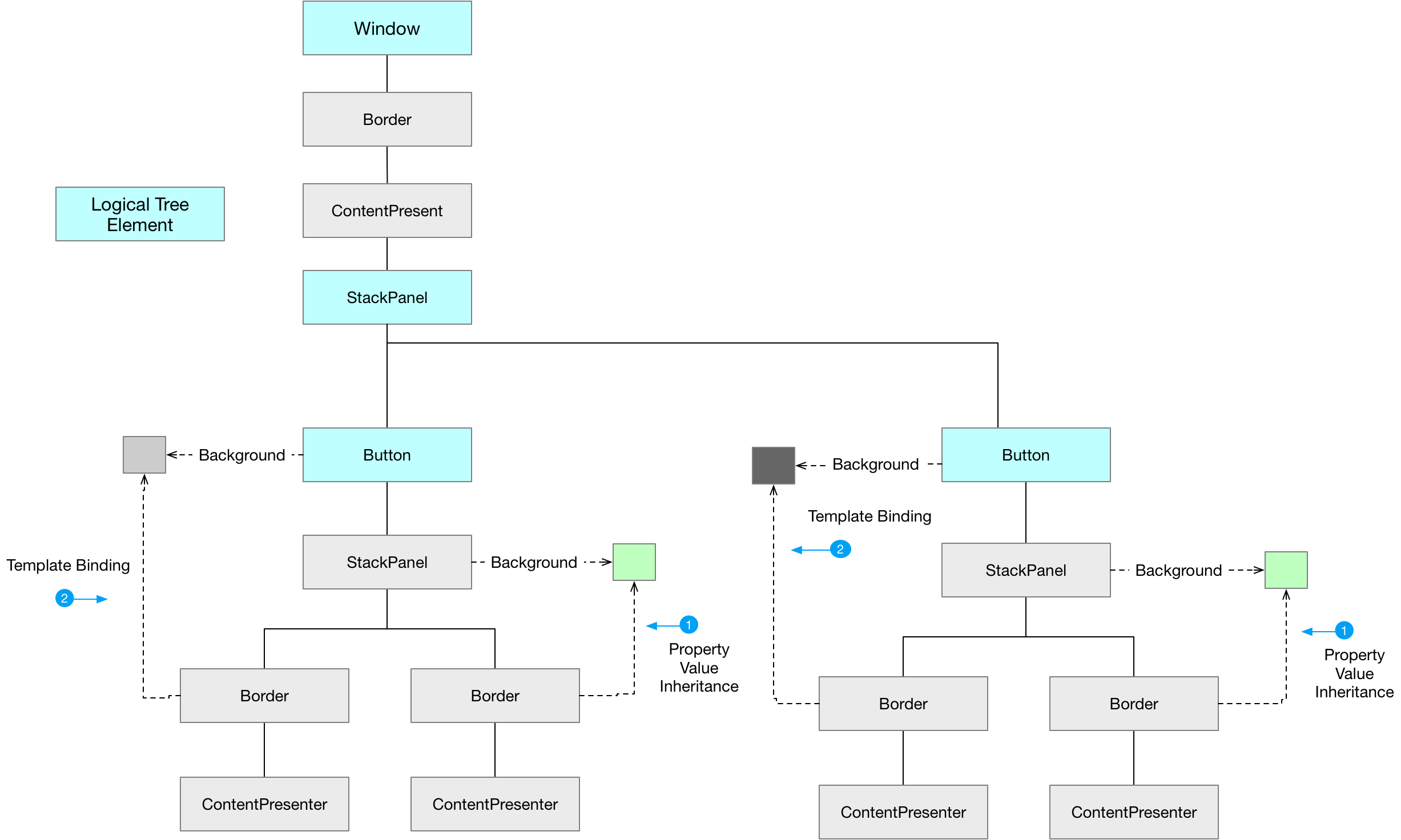


1. The window sets its DataContext to an instance of type Person, a non-visual element. As DataContext is an inheritable DependencyProperty it is inherited by the windows decedents in the visual tree.
2. The elements generated by the Window’s Template can reference their containing Control. This reference 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 from parent objects in the visual tree. It also shows how a visual element in a template can access the value of its templated parent rather than its actual parent in the template.





1. Simple property value inheritance used to inherit values from the parent element in the visual tree.
2. Use the TemplateBinding to access values from the TemplatedParent. 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.



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



1. Register using the Register method and handle change



1. Add standard .NET property wrapper



## Coding an Attached Dependency Property

1. Add a static instance with ‘Property’ prefix



1. Register using the RegisterAttached method and handle change



1. Add standard .NET property wrapper



1. Add static get/set methods



Dependency Property - Questions

Basics

Compare Dependency Properties to standard .NET properties

1. DependencyProperty value is resolved dynamically when GetValue is called. Standard .NET properties are usually backed by a field in a class
2. DependencyProperties have build in change notification

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 DependencyProperties

They allow leaf nodes to share a value set on a parent even though the parent does not understand 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

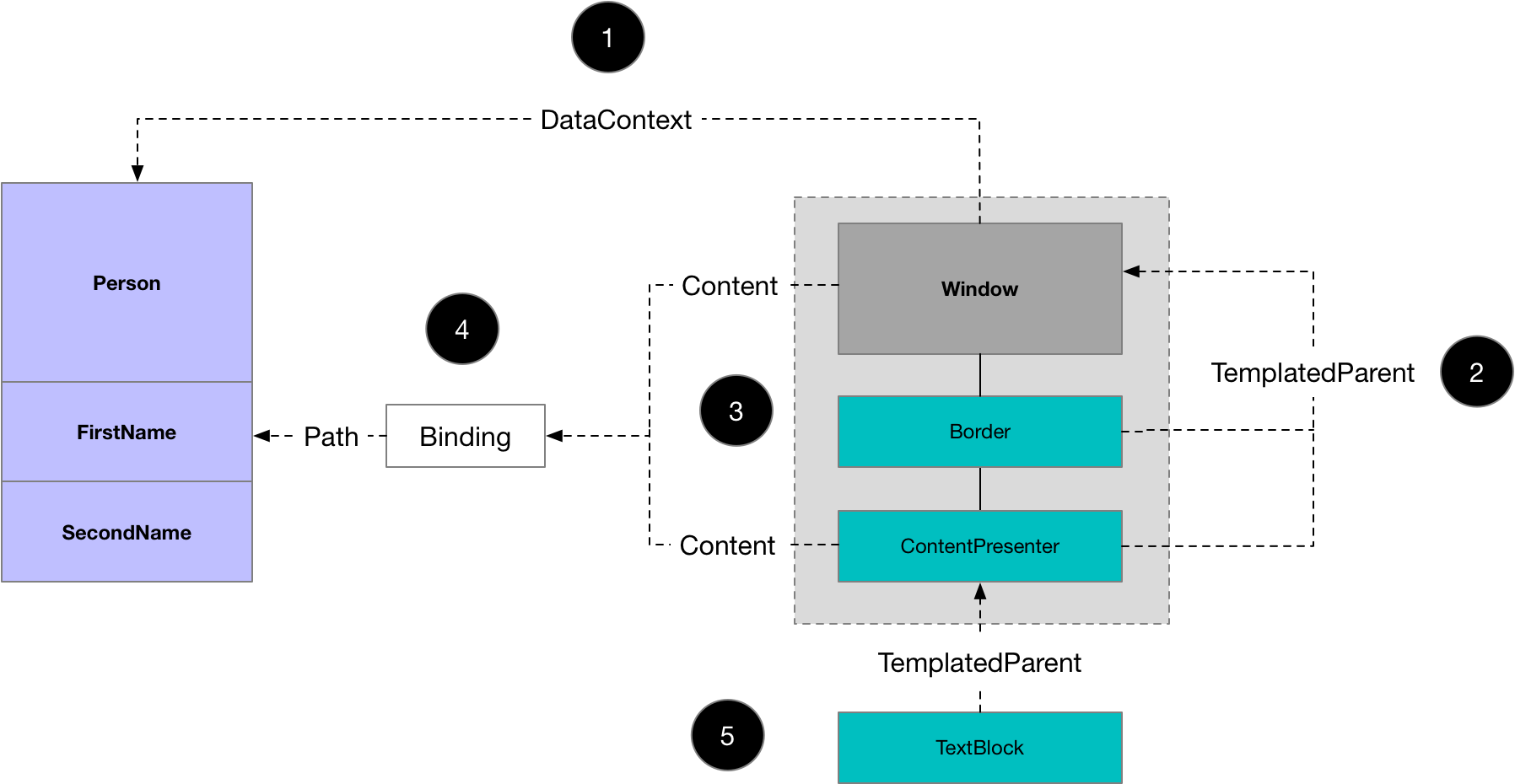
What constraints are on the binding source object?

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

### Dependency Properties and Control Templates (1)



1. The window sets its DataContext 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)

