**[Dr WPF. Articles](http://drwpf.com/blog/2007/08/)**

**2007**

**August**

**Can my value converter access the target of the binding?**

Use a binding in a static resource ie Content="{StaticResource {Binding Path=ID}}"

**Ans**: You can use multibinding.

<ContentControl.Content>

        <MultiBinding Converter="{StaticResource MyConverter}">

          <MultiBinding.Bindings>

            <Binding RelativeSource="{RelativeSource Self}" />

            <Binding Path="ID" />

          </MultiBinding.Bindings>

        </MultiBinding>

      </ContentControl.Content>

**September**

**How to pass instance of object in markup**

**Ans:** You can use different ways

1. Use multibinding as above
2. Create MarkupExtension with weakReference

**October**

**Description of Resources**

Static Resource can be used on CLR as well as Dependency properties.

Then it discusses what happens to get a WPF class started.

Deals with hosting WPF inside WinForms app.

**ToolTip Placement**

**ItemsControl A** and **B**; Just a gentle introduction.

**November**

**MenuItem trigger when mouse is underneath it…**

**Borrowing Dependency Properties:** Sometimes you borrow properties rather than creating a new one. Be careful to know exactly what this borrowed DP(or AP) does under the hood as it might impact performance.

**ItemsControl C:** As the ItemsCollection of an Items Control can be IEnumerable, IIist, ICollection, IObservableCollection

It creates a collectionViewSource of the data which which abstract the code to check what type of collection its data is. If it’s an IList it creates a ListCollectionView which is more performant than a CollectionView (used by IEnumerable)

Items and the Element Trees: Items in Items collection make up the logical children of the ItemsControl. If they’re visual then they also are members of the visual tree.

**December**

**Stop Clipping when ClipToBounds=false**

ClipToBounds=false is a misnomer so it doesn’t work all the time. If elements desired size is larger than available size then it will be clipped. To overcome this **wrap the element in a canvas which always arranges its children to their desired size.**

**2008**

**January**

[**ItemsControl: 'D' is for DataTemplate**](http://drwpf.com/blog/2008/01/03/itemscontrol-d-is-for-datatemplate/)

A template is just a tree of visual elements (along with some resources and triggers) used to define the look and feel of a member of the logical tree. As it builds the logical tree, the framework watches for controls and data items that have corresponding templates. When such an element is encountered, the appropriate template is “inflated” into the actual visuals that represent the logical item and those visuals are inserted into the visual tree.

**February**

**ItemsControl: 'P' is for Panel**

ItemsControls uses a panel to layout its children. This is the items host or items panel.

**Layout overview:** WPF layout engine uses a 2 pass cycle to measure and position elements

1. **Measure pass.**
   1. An element is asked what size it wants to be by way of FE.**MeasureOverride**(size availableSize) with the size parm being the available space.
   2. Before we return from MeasureOverride the element asks each of its children(via Panel.Children prop) what size they want my calling Measure() on child which indirectly calls MeasureOverride on that child…
   3. Value returned from MeasureOverride is the DesiredSize of the element
2. **Arrange pass**
   1. Override ArrangeOveride(size availableSize) and call each childs ArrangeSize passing in a rect. The position of the rect is the position of the child relative to its parent. The size of the rect is the size of the child.
   2. After arranging its children it should know its ActualSize which you then return from ArrangeOverride.

**ItemsPanelTemplate** is inflated within the **ItemsPresenter**, thereby creating a panel for the layout of the ItemsControl’s items.

IT can be done by xaml : **<ListBox.ItemsPanel>**

**<ItemsPanelTemplate>**

**<WrapPanel />**

**</ItemsPanelTemplate>**

**</ListBox.ItemsPanel>**

Or it can be also done by setting Panel.IsItemHost=true

<ScrollViewer Padding="{TemplateBinding Padding}">

**<WrapPanel IsItemsHost="True" />**

</ScrollViewer>

**March**

**ItemsControl: 'I' is for Item Container**

Items container examples are ComboBoxItem, ListBoxItem and for the basic ItemsControl a ContentPresenter.

As practically anything can be contained within an ItemsControl, the ItemContainer is a way of bringing uniformity. Useful for virtualization, for a common selected state, common background state, for mapping the visual to the item (in the case of data template, it’s the control into which the template is inflated).

If elements background is transparent it means element is still hittable with mouseclicks

**ItemContainerStyle** is a good place to set size and positioning properties as opposed to the datatemplate that renders the items as the template might be used elsewhere.

**April**

[**Independence for visual and logical children**](http://www.codeproject.com/KB/WPF/ConceptualChildren.aspx)

**May**

[**Using freezables to provide inheritance context**](https://blogs.msdn.microsoft.com/mikehillberg/2008/05/21/model-see-model-do-and-the-poo-is-optional/)

<Button Content="Rename">

      <Button.Command>

        <mc:MethodCommand MethodName="Rename">

          <mc:MethodArgument Value="{Binding Text, ElementName=\_renameTextBox}" />

        </mc:MethodCommand>

      </Button.Command>

    </Button>

By using freezable on a **non frameworkelement** class (in this case our MethodCommand– [Button.Command is not a FE]) we can then use that class in the logical tree as it will have the datacontextof its parent.

BindingOperations.SetBinding(this, ElementDataContextProperty, new Binding()) sets a defaultbinding onto a private object dp. A default binding binds to the inherited DataContext and since the class that the code is in is a freezable (therefore a visual) the datacontext is the DC on the visual tree!

Then in our MethodCommand.Execute() we use the datacontext to pass the text in the renameTextBox to a ‘Rename’ method on the Datacontext.

**July 2008**

[**ItemsControl: 'G' is for Generator**](http://drwpf.com/blog/2008/07/20/itemscontrol-g-is-for-generator/) **Don’t really think I need this.**

How the Items containers are created.

Done by an **ItemContainerGenerator** that can create and remove items. (it removes items via a weak event listener to the collections INotifyCollectionChanged).

The items panel holds the ItemContainerGenerator and calls it. The icg uses the ItemsControl by calling its GetContainerForItem() which creates and returns an item.

The datacontext of the ItemContainer is also the datacontext of the item it contains.

Good tip: to get at container of item template do this

<DataTrigger Binding="{Binding **RelativeSource={RelativeSource FindAncestor,**

**AncestorType={x:Type ListBoxItem}},** Path=IsSelected}" Value="True">

<Setter Property="Foreground" Value="#A1927E" TargetName="tb" />

</DataTrigger>

**Oct**

[**ItemsControl: 'E' is for Editable Collection**](http://drwpf.com/blog/2008/10/20/itemscontrol-e-is-for-editable-collection/)

**ListCollectionView** sorts and groups its data when it creates data only, not when data is being edited.

You can update the data by calling **Refresh**() on **ListCollectionView,** however this will clear and create the entire list.

You could also just remove and readd the item in question, however this is not great and the item will no longer be the current item.

Use **IEditableObject**. on the data object and then call **CommitEdit**() on the ListCollectionView

**2009**

**January**

[**Size Matters:** Some tips on how to decrease size of WPF apps](http://drwpf.com/blog/2009/01/)

**March**

[**ValueConverter**](http://drwpf.com/blog/2009/03/17/tips-and-tricks-making-value-converters-more-accessible-in-markup/)

Creates a static valueconverter combined with MarkupExtension so you don’t have to store it in the resources.

**May**

[**ItemsControl: ‘L’ is for Lookless**](http://drwpf.com/blog/2009/05/)**.**

Lookless control model: The separation of the actions of the control from how it looks. Achieved using xaml and templates

The ControlTemplate class exposes a TargetType property. This property *must* always be set.

Template should use templatebinding on its values whenever possible.

By using binding a data trigger is very handy for triggering off of a property on another object,

The **ItemsPresenter** reserves space within the items control template for the **ItemsPanel**. By including this ItemsPresenter element within the control template and specifying the ItemsPanelTemplate separately, it is possible to have a swappable items panel without having to retemplate the entire ItemsControl.

However you can include the **ItemsPanel** in the control template by adding a panel and setting its **IsItemsHost** to true.

Some controls will always need certain parts to work. IE a scrollbar will always need a trackbar. A combobox will always need a popup. We yse template parts using prefix ‘PART\_’ to deal with this. Don’t forget if you don’t provide 1 the control will use its default template!

By overriding **OnApplyTemplate**(), a control can be notified when it’s template is inflated and loaded into the visual tree. The **GetTemplateChild**() method can then be used to look up a template

**August 2009**

[**ItemsControl: 'N' is for Natural User Interface**](http://drwpf.com/blog/2009/08/05/itemscontrol-n-is-for-natural-user-interface/)

Custom panel that scrolls

# [**Thomaslevesque wpf blog**](https://www.thomaslevesque.com/category/wpf/page/2/)

**Custom MarkUp Extensions**

**EnumValues:** When displaying Enums, as opposed to using an ObjectDataProvider, you can create a MarkupExtension with a Type as a property. Then in the **ProvideValue** method return Enum.GetValues(type).

**Updateable Extension.**  ME’s only fire once but by getting the Target property (via IProvideValueTarget) in **ProvideValue** and storing that you can then send it values multiple times

**EventBinding Extension**: Pass in methodname. In **ProvideValue** get event from Target.Property and then create delegate using reflection and datacontext (from Target) and event along with methodname.

**Async ObservableCollection**

Override **ObservableCollection** and set a **SynchronizationContext** in its ctr(which will be set to the UI context). Then override the OnCollectionChanged and call its code from within a SynchronizationContext.Post!

**Bind to elements where DataContext does not exist:** If control does not inherit from visual then its not part of the visual tree. Examples are a ContentElement and data grid columns.

1. Add a framework element to tree, set it to collapsed and set its name. Then in the datagridcolumn set its binding source to x:reference of frameworkelement and set its path to DataContext.WhateverProperty.
2. Create a class that overrides freezable and add an AP of object. Then define this object within the xaml resource with the AP point to {Binding}. Then in the datagrid set its source to the object resource and set path to Data.WhateverProperty..

**WPF loading process**

**The app.loadcomponent()** method extracts the BAML (the compiled XAML) from your assembly and uses it to build your user interface. As it parses the BAML, it creates each control object, sets its properties, and attaches any event handlers.

To understand how the Initialized and Loaded events relate, it helps to consider the rendering process. The FrameworkElement implements the ISupportInitialize interface, which provides two methods for controlling the initialization process. The first, **BeginInit**(), is called immediately after the element is instantiated. After **BeginInit**() is called, the XAML parser sets the elements name, event handlers and properties (in that order) and adds any content. The second method **EndInit**(), is called when initialization is complete, at which point the **Initialized** event fires.

When you create a window, each branch of elements is initialized in a bottom-up fashion. That means deeply nested elements are initialized before their containers. When the Initialized event fires, you are guaranteed that the tree of elements from the current element down is completely initialized. However, the element that *contains* your element probably isn’t initialized, and you can’t assume that any other part of the window is initialized.

After each element is initialized, it’s also laid out in its container, styled, and bound to a data source, if required. After the Initialized event fires for the window, it’s time to go on to the next stage. After the initialization process is complete, the Loaded event is fired. The Loaded event follows the reverse path of the Initialized event—in other words, the containing window fires the Loaded event first, followed by more deeply nested elements. When the Loaded event has fired for all elements, the window becomes visible and the elements are rendered.

**ContentProperty** attribute is used by most controls/panels to define which property to add their nested content to. For a Panel its Children, Button its Content, ItemsControl its Items…

**FrameworkElement.MeasureOverride**(Size availableSize)

For each of Panels children calls **UiElement.Measure**(size) passing in some size of your liking . \*

Child then has a desiredSize

Which then added to desiredSizeSum

We return the desiredSizeSum

In English we pass in available size to each of panels children and return the sum of the childrens desiredSize

\* Size of infinity can be passed in here. This is the case of a **Stackpanel** where either the **height** or **width** is infinite depending on the **orientation**

**FrameworkElement.ArrangeOverride**(size finalSize)

For each of elements children call **UIElement.Arrange** passing in a Rect which contains actual size and location

**UIElement.Arrange**  positions the element

Nothing is returned

We return a size which becomes the **ActualSize** and **RenderSize**

**Note:**

1. Margin sizes are automatically subtracted from the availableSize before being given to the MeasureOverride and ArrangeOverride.
2. Alignment is automatically handled by the ArrangeOverride method. It only comes into play if there is available space.

**Controls**

**Dispatcher**

**Dispatcher Dispatcher{get;}**

**DependencyObject**

**ClearValue(dp)**

**CoerceValue(dp)**

**obj GetValue(dp)**

**InvalidateProperty(dp)**

**SetValue(dp)**

**Visual…**

**UiElement**

**AddHandler/RemoveHandler(RoutedEvent,Delegate)**

**Arrange(Rect)**

**Measure(Size)**

**InvalidateArrange():**

**InvalidateMeasure():**

**InvalidateVisual():**

**RaiseEvent(RoutedEventArgs)**

**FrameworkElement**

**AddLogicalChild(obj):** Adds control to logical and visual tree

**ApplyTemplete():** Builds the templates visual tree

**BeginInit/EndInit()**: Starts and ends the initialization for the element.

**obj FindName(string):** Finds resource.(This actually looks throughout the whole tree

**obj FindResource(obj)**: Finds resource with the elements resources

**Size ArrangeOverride(size) :** Arrange and sizes its child elements

**Size MeasureOverride(size):** Measures its child elements

**Alignment:** Horiz and Vert

**Margin**

**Height/Width**

**DataContext**

**Parent**

**Resources**

**Style**

**Panel**

**Visual GetVisualChildren(int)**

**Get/SetZIndex(uielement)**

**UiElementCollection Children**

**Control**

**Size ArrangeOverride(size) :** Arrange and sizes its content

**Size MeasureOverride(size):** Measures its content

**ContentAlignment:** Horizontal and Vertical

**ContentControl**

**AddChild(obj)**

**AddText(string)**

**obj Content**

**DataTemplate ContentTemplate**