CONTENTS INCLUDE:

- XAML
- Markup Extensions
- Controls
- Panels
- Data Binding
- Events and more...

Getting Started with Windows Presentation Foundation

By Christopher Bennage and Rob Eisenberg

ABOUT WPF

Windows Presentation Foundation or WPF is a next generation UI framework for creating desktop applications on the Windows Platform. It brings together a number of features and concepts such as a declarative language for constructing interfaces, rich media support, scalable vector graphics, timeline-based animations, sophisticated data binding, and much more.

WPF is a very large topic. The intent of this Refcard is to help you understand the basics of WPF. After we're done you should be able to look at the source of a WPF application and understand what you are seeing. However, in order to effectively use WPF, you will need to continue to learn more though additional and more extensive resources.

XAML

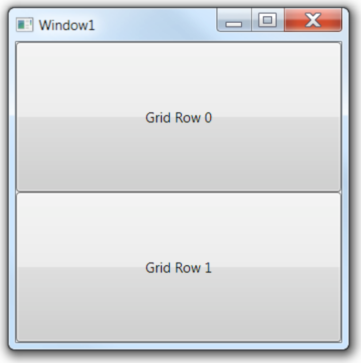
XAML, or eXtensible Application Markup Language, is the lingua franca of both WPF and Silverlight. It is an XML dialect that was designed to represent hierarchical object graphs in a fashion that is both human-readable and easy to parse. In WPF, XAML is used to represent the composition of the user interface, its style, animations and almost any declarative aspect of the user experience. Let's see an example:

```
<Window x:Class=”XamlSamples.Window1”
     xmlns=”http://schemas.microsoft.com/winfx/2006/xamlpresentation”
     xmlns:x=”http://schemas.microsoft.com/winfx/2006/xaml”
     Title=”Window1”
     Height=”300”
     Width=”300”>
 <Grid x:Name=”layoutRoot”>
 </Grid>
</Window>
```

If you create a new WPF project in Visual Studio, the XAML listed will be generated for you. It shows some common features of the markup language. The first thing you should notice is the use of XML namespaces, denoted by `xmlns`. These namespaces tell the XAML parser where to find the elements declared in the markup. The first declaration makes WPF the default namespace. This allows the majority of elements to be declared without explicit namespaces. The second namespace is that of the XAML parser itself, which enables some special cases, such as naming elements. `x:Name` is an example of this. Notice that the root node is `Window` and it has attributes of `Title`, `Height` and `Width`. In XAML, elements correspond to instances of objects that will be created, and attributes specify what the instances' properties should be set to. The XAML translates to the following code:

```
var layoutRoot = new Grid();
var window = new Window
{
    Title = “Window1”,
    Width = 300,
    Height = 300,
    Content = layoutRoot
};
```

This sample also demonstrates an important concept in WPF: the `Content Model`. Notice that `Window` in the XAML has a `Grid` as its child element, but it translates to setting the `Content` property on the `Window` in code. Most WPF elements declare a default content property so that XAML creation is made more intuitive. Let's discover some more features of XAML by putting some elements in our `Grid`:



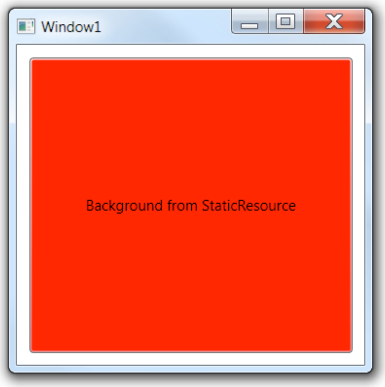
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height=”*” />
        <RowDefinition Height=”*” />
    </Grid.RowDefinitions>
    <Button Grid.Row=”0”
            Content=”Grid Row 0”/>
    <Button Grid.Row=”1”
            Content=”Grid Row 1”/>
</Grid>
```

Knowing that elements map to instances and attributes map to properties, you may be wondering how you would set a property with a complex value; one that could not be expressed as an attribute. This is accomplished by using `Property Element Syntax`, demonstrated by the `Grid.RowDefinitions` element. A `Grid` is a `Panel` composed of children arranged in columns and rows. To declare the rows in a `Grid`, you set its `RowDefinitions` property, which is a collection of `RowDefinition` objects. In XAML, you can turn any property into an element using the syntax `TypeName.PropertyName`, as we have done with `Grid.RowDefinitions`. By combining `Property Attribute Syntax` and `Property Element Syntax` with the conventions of the `Content Model` you can represent almost any object hierarchy. Additionally, there is another common markup usage demonstrated: `Attached Properties`. You can see them on the `Button` declarations. WPF enables containing elements to attach information to their children using the pattern `ParentType.AttachedProperty`. In this case, the `Grid.Row` properties tell the parent `Grid` where to place each `Button`.

MARKUP EXTENSIONS

Despite the flexibility of the XAML syntax, there are still many scenarios that are tricky to accomplish with standard XML. To address this issue, XAML offers Markup Extensions. Here's a typical example:

```
<Window x:Class="MarkupExtensionSamples.Window1"
     xmlns="http://schemas.microsoft.com/winfx/2006/xamlpresentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Title="Window1"
     Height="300"
     Width="300">
 <Window.Resources>
     <SolidColorBrush x:Key="myBrush"Color="Red" />
 </Window.Resources>
 <Grid>
     <Button Background="{StaticResource myBrush}"
             Margin="10"
             Content="Background from StaticResource"/>
 </Grid>
</Window>
```



Often, a designer likes to declare colors, brushes and styles for consistent use throughout an application. These can be stored in a `ResourceDictionary`. In this example, we have declared a `SolidColorBrush` which we would like to later use as our button's background. The `StaticResourceExtension` class is the perfect tool for the task. We use it by enclosing the words "StaticResource" in braces, followed by the `Key` of the resource we want to reference. There are several things common to all markup extensions that we should note:

1. Markup extensions are enclosed in braces.
2. XAML allows us to drop the word "Extension" from the declaration, even though it is part of the class name.
3. Most extensions have a default value, which will be passed into the extension's constructor. In this example that value is the resource key, "myBrush."

Any read/write property on the extension class can also be set in XAML. These properties can even be set using other extensions! Here's a typical databinding sample:

```
<TextBox Text="{Binding Source={StaticResource myDataSource},
                Path=FirstName, UpdateSourceTrigger=PropertyChanged}"
/>
```

Notice that we set the Source property of the `BindingExtension` using another `MarkupExtension`, `StaticResourceExtension`. We then set two additional properties on the `BindingExtension`.

Table 1 shows several of the most common built-in extensions along with their purpose:

Extension	Description
{StaticResource}	Injects a previously defined resource into markup.
{DynamicResource}	Creates a dynamically updatable link to a resource.
{Binding}	Enables databinding.
{TemplateBinding}	Simplifies binding inside a ControlTemplate.
{x:Static}	References static variables.
{x:Type}	References instances of a Type object.
{x:Null}	Represents a Null value.

In addition to what XAML provides out-of-the-box, you can create your own markup extensions. All you have to do is derive your class from `MarkupExtension`. For example:

```
public class HelloExtension : MarkupExtension
{
    private readonly string _name;

    public HelloExtension(string name)
    {
        _name = name;
    }

    public override object ProvideValue(IServiceProvider
serviceProvider)
    {
        return "Hello " + _name + "! Nice to meet you.";
    }
}
```

And you use it in code like so:

```
<Window x:Class="MarkupExtensionSamples.Window1"
     xmlns="http://schemas.microsoft.com/winfx/2006/xamlpresentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:local="clr-namespace:MarkupExtensionSamples"
     Title="Window1"
     Height="300"
     Width="300">
 <TextBox Text="{local:Hello Fauntleroy }" />
</Window>
```

Whenever you create custom extensions, or any class not defined by the framework, you must remember to add its namespace before you use it in XAML. A common convention is to use the name "local" for classes defined in the project.

Controls

WPF ships with a number of the standard controls that you would expect. Here's a short list of the most common:

- Label
- Button
- TextBox
- ListBox
- ComboBox
- CheckBox
- Slider

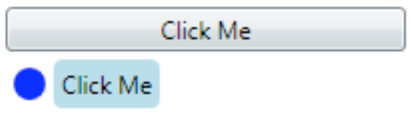
These controls all look and behave according to traditional desktop UI metaphors. However, the nature of controls in WPF is very different from traditional UI frameworks. You can completely redefine the look of a control in WPF without altering its behavior. You can do this using `Control Templates`.

The most common scenario for manipulating a control template is changing the way a button looks. All controls have a Template property. Let's examine the following XAML:

```
<Button Content="Click Me">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <StackPanel Orientation="Horizontal"
                           Margin="4">
                <Ellipse Width="16"
                         Height="16"
                         Fill="Blue"
                         Margin="0 0 4 0" />
                <Border Background="LightBLue"
                        CornerRadius="4"
                        Padding="4">
                    <ContentPresenter />
                </Border>
            </StackPanel>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Using the `Property Element Syntax`, we set the button's template to an instance of `ControlTemplate`. The `TargetType` is necessary to tell the control template what sort of control it will be applied to. Control templates require this because they typically exist independent of the controls themselves. The content of a control template can be just about anything. This demonstrates the compositional nature of WPF. A control template can be composed of any other WPF elements, even another Button control.

Below is an example of how this XAML will render compared to a plain button.



There are additional elements that are used to compose an interface in WPF which technically are not controls. Some examples from the XAML are `StackPanel`, `Ellipse`, `Border`, and `ContentPresenter`.

`StackPanel` is part of a family of elements called panels that assist us with layout. We'll talk about them in the next section.

`Ellipse` and its siblings are shapes. Shapes are a convenient set of classes for drawing basic shapes within the UI.

`Border` is an element that you will encounter frequently. It is used to decorate other elements with a border. It is also the quick way to put rounder corners around an element.

`ContentPresenter` is a special element that is used when you are constructing control templates. It is a placeholder for the actual content of the control. For example, we set the Content property of our `Button` to "Click Me". This content was injected into our template where we placed the `ContentPresenter`. It's also interesting to note, that if we had omitted the `ContentPresenter` then our control template would simply have ignored the content.

Control templates are especially powerful when they are used in combination with WPF's styles and resources.

Missing Controls

There are a number of controls that you might expect to be present in WPF that are not. The most obvious examples are `DataGrid`, `DatePicker`, `Calendar`, and charting controls. Microsoft has chosen to release these controls separately from the .NET platform itself. They are available as part of the WPF Toolkit on CodePlex (http://www.codeplex.com/wpf). You'll also find the official Ribbon control on CodePlex. Look for most of these controls to join the platform in version 4.0.

Lookless Controls

Two of the most important controls in WPF are the `ContentControl` and the `ItemsControl`. These controls do not have any defined look. They rely on a concept called `Data Templates`. Data templates are similar to control templates, except that instead of targeting a specific control, they target a specific type.

With a `ContentControl`, you can set the `Content` property to an instance of any class. Then you can define a data template that is specific to that class. Say that we have a simple class like this one:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

We can define a data template is our resources:

```
<DataTemplate x:Key="personTemplate">
    <StackPanel>
        <TextBlock Text="{Binding FirstName}" />
        <TextBlock Text="{Binding LastName}" />
        <TextBlock Text="{Binding Age}" />
    </StackPanel >
```

You can also define an instance of Person in the resources:

```
<ContentControl Content="{Binding Source={StaticResource myPerson}}"
                ContentTemplate="{StaticResource personTemplate}" />
```

You might be wondering why this is valuable. Why not simply create a user control that has the same content as the data template? The ContentControl has no inherent behavior beyond rending its content. With a user control, you could at least add behavior specific to the Person class. We'll come back to this question.

Another way to associate the data template is to provide a `DataType`. Here's same template using this approach:

```
<DataTemplate DataType="{x:Type local:Person}">
    <StackPanel>
        <TextBlock Text="{Binding FirstName}" />
        <TextBlock Text="{Binding LastName}" />
        <TextBlock Text="{Binding Age}" />
    </ StackPanel>
</DataTemplate>
```

Depending on where this data template was stored, you could now do the following:

```
<ContentControl Content="{Binding Source={StaticResource myPerson}}" />
```

We'll discuss DataContext in the section on Data Binding.

The ItemsControl is very similar to ContentControl, except that it binds to a collection of objects using the ItemsSource property. In fact, if we bound it to a collection of Person instances, the same data template would be applied.

```
<ItemsControl ItemsSource="{Binding Source={StaticResource
myPersonCollection}}" />
```

Of course, you can also explicitly designate the template:

```
<ItemsControl ItemsSource="{Binding Source={StaticResource
myPersonCollection}}"
              ItemTemplate="{StaticResource personTemplate}" />
```

These two controls combined with data templates are very powerful. They are an essential part of Separated Presentation patterns such as MVVM or Presentation Model.

PANELS

You've already seen a couple of panels, namely the `StackPanel` and `Grid`. Panels are UI elements that provide layout functionality. They are used to layout windows, user controls, and even the standard controls that ship with WPF. If you examine the control templates for the default controls themes, you will find these panels everywhere. Table 2 lists the commonly used panels.

Panel	Description
Canvas	Allows you to place elements at specific coordinates. Typically, you will use one of the other panels unless you have a specific need to place elements exactly.
DockPanel	This is a versatile panel that allows you to "dock" elements to any of the four sides. It is commonly used to layout the "shell" of an application: dock the menu bar to the top, dock the status bar to the bottom, dock the explorer panel to the left-hand side.
Grid	Similar to a table in HTML. You define a number of rows and columns that are used to arrange the elements. This is not to be confused with any data grid controls. This panel is used only for laying out other elements.
StackPanel	As its name implies, this panel simply stacks elements one on top of the other. You can also tell it to use a horizontal orientation in order to "stack" elements sideways.
WrapPanel	This panel is similar to StackPanel with a horizontal orientation except that its child elements wrap to a new row automatically when all of the horizontal space has been used.

You'll frequently need to nest panels of various types in order to achieve the exact layout that you desire.

The child elements of a `Canvas`, `DockPanel`, or `Grid` use attached properties to interact with their parent.

For example, to place a red circle inside of a `Canvas` at 100,200:

```
<Canvas Width="800"
        Height="600">
    <Ellipse Width="32"
             Height="32"
             Fill="Red"
             Canvas.Top="100"
             Canvas.Left="200" />
</Canvas>
```

`DockPanel` uses the attached property `DockPanel.Dock`. The values are `Left`, `Top`, `Right`, and `Bottom`.

`Grid` uses two attached properties: `Grid.Column` and `Grid.Row`. The values of the properties are the index of the column and row where you want the child element to appear.

DATA BINDING

Data binding allows you to declaratively establish a relationship between an element of the UI and some other bit of data. A clear understanding of the way data binding works in WPF will revolutionize the way you design your applications. We'll start by examining a simple example of data binding.

```
<StackPanel>
    <TextBox x:Name="mySource" />
    <TextBox Text="{Binding ElementName=mySource, Path=Text}" />
</StackPanel>
```

In this example, we have two text boxes. We gave the name `mySource` to the first text box so that we can reference it in the binding. The binding itself is a value that we are setting to the Text property of the second text box. We're using the markup extension `Binding` to define it. This is not the only way to declare bindings, but it is the most common.

We need to tell our binding where to locate its source data. We do this using the property `ElementName` to reference the source element. Next, we need to identify what property on `mySource` we'd like to use. The `Path` property on the binding allows just that. We can use it to draw a link from the source to the actual bit of data we want to bind. In this example, the path is very simple.

Data Context

UI elements are not the only possibilities for a data source in WPF. Frequently, you'll need to bind some business object or view model. Nearly every element in WPF has a property called DataContext. As its name implies, it provides an object that is the context for data bindings. Take a simple example derived from the data template we discussed earlier:

```
<TextBlock Text="{Binding Path=FirstName}" />
```

Because we are not explicitly providing a source, the binding assumes that it should use the `DataContext` of the `TextBlock` as the source for the binding.

One wonderful characteristic of `DataContext`, is that it is automatically inherited from an element's parent. We can set the `DataContext` once for an entire window (or some other graph of UI elements) and the data context is propagated down to all of the child elements contained in that window.

There are a number of ways to set the data context, however the most direct looks like this:

```
this.DataContext = new Person();
```

If we place this code in the constructor for our window, then all of the elements in our window will have a person instance as their data context.

Hot Tip

Since Path is the default property for the Binding extension, we can (and frequently do) omit the "Path=". Our example binding would then look like this:
`<TextBlock Text="{Binding FirstName}" />`

Different Sources for Bindings

There are three different types of bindings available in WPF. You've already seen several examples of the most common, which is simply called Binding.

With basic binding there are four ways to specify the data source for the binding.

- Simply allow the binding to use the DataContext.
- Designate another element in the UI with ElementName.
- Explicitly provide the source using the Source property. This is useful for accessing data that is part of a static class. For example, if we wanted to bind the number of fonts currently installed we could do this:

```
<TextBlock Text="{Binding Source={x:Static Fonts.SystemFontFamilies},
Path=Count}" />
```

- Finally, you can specify a RelativeSource. Use this when the source of data is the same element as the target. For example, perhaps we want the ToolTip on a ListBox to report the number of items that are current in the ListBox. We could do this:

```
ToolTip="{Binding RelativeSource={x:Static RelativeSource.Self},
Path=Items.Count}"
```

Hot Tip Data bindings do not raise exceptions. If the path doesn't exist or if the source is null, your application will continue to run, but the bindings will not do anything. Keep an eye on the output window if you are having problems. The most common is "BindingExpression path error: 'X' property not found on 'Y'.

Value Converters

Sometimes the data source that you want to use doesn't exactly match the type of the target. One common example is binding when you need to bind a boolean to the visibility of an element in the UI.

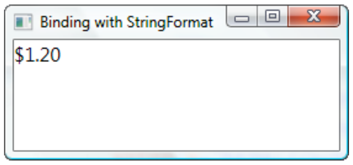
```
<Border Visibility="{Binding IsTrueOrFalse}" />
```

This binding will simply fail silently. The solution to this is to use a value converter. Fortunately, there is a converter built into the framework for this exact scenario. To use it, we'll need to create an instance in the resources. Here's an example:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
    <BooleanToVisibilityConverter x:Key="booleanToVisibility" />
  </Window.Resources>
  <Border Visibility="{Binding IsTrueOrFalse, Converter={StaticResource
booleanToVisibility}}" />
</Window >
```

We set the `Converter` property on the binding to point to the instance of our converter that we keyed in the window's resources.

Creating your own value converters is very easy. Simply create a class that implements `IValueConverter`. There are two simple methods involved, and frequently you'll only need to implement one.



Other Types of Bindings

There are two other types of bindings that are more rare, but very useful in certain scenarios.

- MultiBinding
- PriorityBinding

`MultiBinding` allows you to specify multiple bindings for a single target. However, you will need to implement `IMultiValueConverter` in order to consolidate these bindings down into a single value.

`PriorityBinding` also allows you to specify multiple bindings, but instead of being consolidated, only a single binding from the collection is used. The bindings are listed in order of their priority. The rule for determining which binding to use is: choose the highest priority binding that has resolved a value. This is very useful when you are binding to values that take a significant amount of time to return.

Check the official documentation for more information on how and when to use these bindings.

EVENTS AND COMMANDS

WPF and XAML provide a rich framework for declarative UI design. But how do you respond to user interactions within the interface? This is accomplished through Events and Commands. Let's see the standard approach for responding to a `Button.Click` event.

```
<Button Content="Click Me!"
                Click="Button_Click" />

private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Click!");
}
```

Events are special types of properties in .NET, so we can use XAML's `Property Attribute` convention to set them in markup. The code-behind file (.xaml.cs extension) contains the actual method body that is wired to the event. In this case, clicking the Button will cause a MessageBox to be shown. Of particular note are the event handler arguments. The first argument, `sender`, is the `Button` itself. The second argument, e, is a special type of `EventArgs` used by all WPF's events. This argument contains additional contextual information provided by WPF's routed event mechanism.

Routed Events

In WPF, all events travel a path. Bubbling events travel from the source node up to the root element in the UI hierarchy. Tunnelling events travel from the root node down to the source. Events bubble by default, but often have a tunneling counterpart designated by the word "Preview." For example: `MouseLeftButtonUp` and `PreviewMouseLeftButtonUp`. The `RoutedEventArgs` has properties that can help in determining the source of the event. You can also prevent further bubbling or tunneling by setting the `Handled` property of the args to `true`.

Commands

Wiring events is not the only way to handle user interaction. You can also use `Commands`. `Button` has a `Command` property.

```
public class MessageBoxCommand : ICommand
{
    public void Execute(object parameter)
    {
        MessageBox.Show(parameter.ToString());
    }

    public bool CanExecute(object parameter)
    {
        return parameter != null;
    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
}
<Button Content="Click Me!"
        CommandParameter="Hello from a command!">
    <Button.Command>
        <local:MessageBoxCommand />
    </Button.Command>
</Button>
```

Buttons (along with all descendents of `ButtonBase`) and Menus are the two most common implementers of `ICommandSource`. This interface allows you to specify code to execute when the `ICommandSource` is triggered using the `ICommand.Execute` method. Additionally, you can supply code to tell WPF under what conditions execution is possible using the `ICommand.CanExecute` method. The `ICommandSource` will typically wire itself to the `ICommand.CanExecuteChanged` event and update its `IsEnabled` property whenever things change. It is typically sufficient to allow WPF's `CommandManager` to handle the event. Also, notice that the `Execute` and `CanExecute` methods have a single parameter which can be supplied by setting the `CommandParameter` on the `ICommandSource`.

Routed Commands

Like events, WPF's commands can route. This enables one part of the UI to trigger a command while another part actually handles the execution. To create your own routed commands, you must inherit from `RoutedCommand`. To handle a `RoutedCommand`, you must add a handler to the `CommandBindings` collection of the node you wish to capture the command from. A `CommandBinding` provides hooks for simple event handlers that allow you to respond to the command.

Gestures

Everything that inherits from `UIElement` has an `InputBindings` collection. An `InputBinding`, such as a `KeyBinding` or `MouseBinding` allows you to connect arbitrary user interaction directly to a `Command`. This allows you to trigger code execution based on complex `Gestures` not easily achievable through standard events.

Built-In Commands

WPF has a host of pre-defined routed commands that you may find useful in building your own applications. Their static instances are hosted on utility classes named according to their usage category. Table 3 explains the built-in commands.

ApplicationCommands	New, Open, Close, Print, Undo, Redo, etc.
ComponentCommands	ScrollPageUp, ScrollPageDown, MoveLeft, etc.
MediaCommands	Play, Pause, Stop, FastForward, etc.
NavigationCommands	NextPage, PreviousPage, Search, Zoom, etc.
EditingCommands	AlignCenter, IncreaseIndentation, etc.

ABOUT THE AUTHORS

Rob Eisenberg is a .NET architect and developer working out of Tallahassee, FL. where he is a partner with Christopher Bennage at Blue Spire Consulting. Rob publishes technical articles regularly at devlicio.us and has spoken at regional events and to companies concerning .NET technologies and Agile software practices. He is coauthor of Sam's Teach Yourself WPF in 24 Hours and is the architect and lead developer of the Caliburn Application Framework for WPF and Silverlight.

Christopher Bennage likes to make things. He's particularly fond of computers, WPF, and Silverlight, as well as the glorious and mysterious field of UX. You can follow him on twitter @bennage or through his blog on devlicio.us. He promises not to bite.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-86-8
ISBN-10: 1-934238-86-4
50795
9 781934 238868
$7.95