**WPF**

PV 2012-02

Table of Contents

[WPF Applications 3](#_Toc318266013)

[Programming with WPF 3](#_Toc318266014)

[Markup and Code-Behind 3](#_Toc318266015)

[Markup 3](#_Toc318266016)

[Code Behind 3](#_Toc318266017)

[Applications 4](#_Toc318266018)

[Standalone Applications 4](#_Toc318266019)

[Browser-Hosted Applications 4](#_Toc318266020)

[The Application Class 4](#_Toc318266021)

[General Model 4](#_Toc318266022)

[Content Model 5](#_Toc318266023)

[Triggers 5](#_Toc318266024)

[Control Templates 5](#_Toc318266025)

[Data Templates 6](#_Toc318266026)

[Styles 8](#_Toc318266027)

[Resources 9](#_Toc318266028)

[Themes and Skins 10](#_Toc318266029)

[Custom Controls 11](#_Toc318266030)

[Controls 11](#_Toc318266031)

[Layouts 11](#_Toc318266032)

[Dependency Properties 12](#_Toc318266033)

[Attached Properties 12](#_Toc318266034)

[Routed Events 13](#_Toc318266035)

[Data Binding 13](#_Toc318266036)

[Graphics 13](#_Toc318266037)

[2-D Shapes 13](#_Toc318266038)

[Paths and Geometries 14](#_Toc318266039)

[Painting Shapes 14](#_Toc318266040)

[Stretchable Shapes 15](#_Toc318266041)

[Transforming Shapes 15](#_Toc318266042)

[2-D Geometries 16](#_Toc318266043)

[2-D Effects 16](#_Toc318266044)

[3-D Rendering 17](#_Toc318266045)

[Animation 17](#_Toc318266046)

[Media 17](#_Toc318266047)

[Images 17](#_Toc318266048)

[Video and Audio 17](#_Toc318266049)

[Text and Typography 18](#_Toc318266050)

[Documents 18](#_Toc318266051)

[Flow Documents 18](#_Toc318266052)

[Fixed Documents 19](#_Toc318266053)

[XPS Documents 19](#_Toc318266054)

[Annotations 20](#_Toc318266055)

[Packaging 20](#_Toc318266056)

[Printing 20](#_Toc318266057)

[XAML 20](#_Toc318266058)

[XAML Object Elements 20](#_Toc318266059)

[Attribute Syntax (Properties) 21](#_Toc318266060)

[Property Element Syntax 21](#_Toc318266061)

[Collection Syntax 21](#_Toc318266062)

[XAML Content Properties 22](#_Toc318266063)

[Layout 22](#_Toc318266064)

[Application Startup 22](#_Toc318266065)

[Data Binding 23](#_Toc318266066)

[Styles 23](#_Toc318266067)

[Triggers 24](#_Toc318266068)

[Property Triggers 24](#_Toc318266069)

[Property Triggers 25](#_Toc318266070)

[MultiTriggers 25](#_Toc318266071)

[Templates 25](#_Toc318266072)

[VisualStateManager 25](#_Toc318266073)

[2D Graphics 26](#_Toc318266074)

[Drawings 26](#_Toc318266075)

[Geometries 27](#_Toc318266076)

[Visuals 27](#_Toc318266077)

[Shapes 27](#_Toc318266078)

[Form Events 28](#_Toc318266079)

# WPF Applications

## Programming with WPF

WPF exists as a subset of .NET Framework types that are for the most part located in the System.Windows namespace.

To support some of the more powerful WPF capabilities and to simplify the programming experience, WPF includes additional programming constructs that enhance properties and events: dependency properties and routed events.

## Markup and Code-Behind

You generally use Extensible Application Markup Language (XAML) markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior.

### Markup

XAML is an XML-based markup language that is used to implement an application's appearance declaratively. It is typically used to create windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

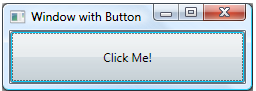
Title="Window with Button"

Width="250" Height="100">

<!-- Add button to window -->

<Button Name="button" Click="button\_Click" >Click Me!</Button>

</Window>



XAML language defines the language-related features x:Class Directive, x:Subclass Directive, x:ClassModifier Directive and x:Code (inline code, not recommended).

### Code Behind

using System.Windows; // Window, RoutedEventArgs, MessageBox

namespace SDKSample

{

public partial class AWindow : Window

{

public AWindow()

{

// InitializeComponent call is required to merge the UI

// that is defined in markup with this class, including

// setting properties and registering event handlers

InitializeComponent();

}

void button\_Click(object sender, RoutedEventArgs e)

{

// Show message box when button is clicked

MessageBox.Show("Hello, Windows Presentation Foundation!");

}

}

}

## Applications

### Standalone Applications

For standalone applications, you can use the Window class to create windows and dialog boxes that are accessed from menu bars and tool bars.

For more information, see WPF Windows Overview.

### Browser-Hosted Applications

For browser-hosted applications, known as XAML browser applications (XBAPs), you can create pages (Page) and page functions ( PageFunction<T>) that you can navigate between using hyperlinks (Hyperlink classes).

WPF applications can be hosted in both Microsoft Internet Explorer 6 and Internet Explorer 7. WPF offers the two following options for alternative navigation hosts:

* Frame, to host islands of navigable content in either pages or windows.
* NavigationWindow, to host navigable content in an entire window.

For more information, see Navigation Overview.

### The Application Class

Both XBAPs and standalone applications are often complex enough to require additional application-scoped services, including startup and lifetime management, shared properties, and shared resources. The Application class encapsulates these services and more, and it can be implemented by just using XAML, as shown in the following example.

<Application

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

StartupUri="MainWindow.xaml" />

This markup is the application definition for a standalone application, and instructs WPF to create an Application object that automatically opens MainWindow when the application is started.

A key concept to understand about Application is that it provides a common platform of support for both standalone and browser-hosted applications. For example, the preceding XAML could be used by a browser-hosted application to automatically navigate to a page when an XBAP is started, as shown in the following example.

<Application

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

StartupUri="HomePage.xaml" />

For more information, see Application Management Overview.

## General Model

Use core WPF building blocks for developing applications.

Use the application model to host and deliver application content, which consists mainly of controls.

Use the WPF layout system to simplify the arrangement of controls in a UI, and to ensure the arrangement is maintained in the face of changes to window size and display settings.

Use data binding to reduce the work of integrating your UI with data.

Use graphics, media and animation to enhance the visual appearance of your application.

If your application operates over text and documents, use the WPF text, typography, document, annotation, packaging, and printing capabilities.

WPF also provides a variety of mechanisms for creating unique user experiences, including a rich content model for controls, triggers, control and data templates, styles, UI resources, and themes and skins.

## Content Model

The main purpose of a majority of the WPF controls is to display content. In WPF, the type and number of items that can constitute the content of a control is referred to as the control's content model. Some controls can contain a single item and type of content; for example, the content of a TextBox is a string value that is assigned to the Text property

Other controls, however, can contain multiple items of different types of content; the content of a Button, specified by the Content property, can contain a variety of items including layout controls, text, images, and shapes.

## Triggers

Although the main purpose of XAML markup is to implement an application's appearance, you can also use XAML to implement some aspects of an application's behavior. One example is the use of triggers to change an application's appearance based on user interactions. For more information, see "Triggers" in Styling and Templating.

## Control Templates

The default UIs for WPF controls are typically constructed from other controls and shapes. For example, a Button is composed of both ButtonChrome and ContentPresenter controls. The ButtonChrome provides the standard button appearance, while the ContentPresenter displays the button's content, as specified by the Content property.

Sometimes the default appearance of a control may be incongruent with the overall appearance of an application. In this case, you can use a ControlTemplate to change the appearance of the control's UI without changing its content and behavior.

For example, the following example shows how to change the appearance of a Button by using a ControlTemplate.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.ControlTemplateButtonWindow"

Title="Button with Control Template" Height="158" Width="290">

<!-- Button using an ellipse -->

<Button Content="Click Me!" Click="button\_Click">

<Button.Template>

<ControlTemplate TargetType="{x:Type Button}">

<Grid Margin="5">

<Ellipse Stroke="DarkBlue" StrokeThickness="2">

<Ellipse.Fill>

<RadialGradientBrush Center="0.3,0.2" RadiusX="0.5" RadiusY="0.5">

<GradientStop Color="Azure" Offset="0.1" />

<GradientStop Color="CornflowerBlue" Offset="1.1" />

</RadialGradientBrush>

</Ellipse.Fill>

</Ellipse>

<ContentPresenter Name="content" HorizontalAlignment="Center"

VerticalAlignment="Center"/>

</Grid>

</ControlTemplate>

</Button.Template>

</Button>

</Window>

using System.Windows; // Window, RoutedEventArgs, MessageBox

namespace SDKSample

{

public partial class ControlTemplateButtonWindow : Window

{

public ControlTemplateButtonWindow()

{

InitializeComponent();

}

void button\_Click(object sender, RoutedEventArgs e)

{

// Show message box when button is clicked

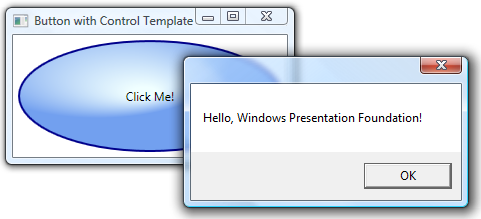
MessageBox.Show("Hello, Windows Presentation Foundation!");

}

}

}

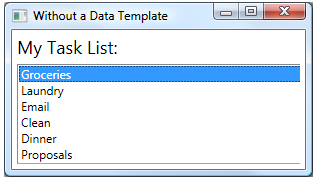
In this example, the default button UI has been replaced with an Ellipse that has a dark blue border and is filled using a RadialGradientBrush. The ContentPresenter control displays the content of the Button, "Click Me!" When the Button is clicked, the Click event is still raised as part of the Button control's default behavior.



## Data Templates

Whereas a control template lets you specify the appearance of a control, a data template lets you specify the appearance of a control's content. Data templates are frequently used to enhance how bound data is displayed.

The following figure shows the default appearance for a ListBox that is bound to a collection of Task objects, where each task has a name, description, and priority.



The default appearance is what you would expect from a ListBox. However, the default appearance of each task contains only the task name. To show the task name, description, and priority, the default appearance of the ListBox control's bound list items must be changed by using a DataTemplate. The following XAML defines such a DataTemplate, which is applied to each task by using the ItemTemplate attribute.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.DataTemplateWindow"

Title="With a Data Template">

...

<Window.Resources>

<!-- Data Template (applied to each bound task item in the task collection) -->

<DataTemplate x:Key="myTaskTemplate">

<Border Name="border" BorderBrush="DarkSlateBlue" BorderThickness="2"

CornerRadius="2" Padding="5" Margin="5">

<Grid>

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<TextBlock Grid.Row="0" Grid.Column="0" Padding="0,0,5,0" Text="Task Name:"/>

<TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=TaskName}"/>

<TextBlock Grid.Row="1" Grid.Column="0" Padding="0,0,5,0" Text="Description:"/>

<TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Description}"/>

<TextBlock Grid.Row="2" Grid.Column="0" Padding="0,0,5,0" Text="Priority:"/>

<TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=Priority}"/>

</Grid>

</Border>

</DataTemplate>

</Window.Resources>

...

<!-- UI -->

<DockPanel>

<!-- Title -->

<Label DockPanel.Dock="Top" FontSize="18" Margin="5" Content="My Task List:"/>

<!-- Data template is specified by the ItemTemplate attribute -->

<ListBox

ItemsSource="{Binding}"

ItemTemplate="{StaticResource myTaskTemplate}"

HorizontalContentAlignment="Stretch"

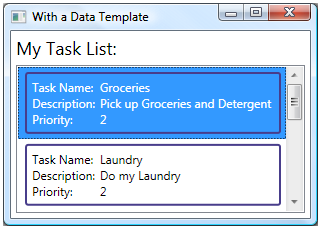
IsSynchronizedWithCurrentItem="True"

Margin="5,0,5,5" />

</DockPanel>

...

</Window>



For more information, see Data Templating Overview.

## Styles

Styles enable developers and designers to standardize on a particular appearance for their product. WPF provides a strong style model, the foundation of which is the Style element.

The following example creates a style that sets the background color for every Button on a window to Orange.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.StyleWindow"

Title="Styles">

...

<!-- Style that will be applied to all buttons -->

<Style TargetType="{x:Type Button}">

<Setter Property="Background" Value="Orange" />

<Setter Property="BorderBrush" Value="Crimson" />

<Setter Property="FontSize" Value="20" />

<Setter Property="FontWeight" Value="Bold" />

<Setter Property="Margin" Value="5" />

</Style>

...

<!-- This button will have the style applied to it -->

<Button>Click Me!</Button>

<!-- This label will not have the style applied to it -->

<Label>Don't Click Me!</Label>

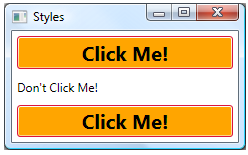
<!-- This button will have the style applied to it -->

<Button>Click Me!</Button>

...

</Window>

Because this style targets all Button controls, the style is automatically applied to all the buttons in the window, as shown in the following figure.



## Resources

Controls in an application should share the same appearance, which can include anything from fonts and background colors to control templates, data templates, and styles. You can use WPF's support for user interface (UI) resources to encapsulate these resources in a single location for reuse.

The following example defines a common background color that is shared by a Button and a Label.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.ResourcesWindow"

Title="Resources Window">

<!-- Define window-scoped background color resource -->

<Window.Resources>

<SolidColorBrush x:Key="defaultBackground" Color="Red" />

</Window.Resources>

...

<!-- Button background is defined by window-scoped resource -->

<Button Background="{StaticResource defaultBackground}">One Button</Button>

<!-- Label background is defined by window-scoped resource -->

<Label Background="{StaticResource defaultBackground}">One Label</Label>

...

</Window>

This example implements a background color resource by using the Window.Resources property element. This resource is available to all children of the Window. There are a variety of resource scopes, including the following, listed in the order in which they are resolved:

1. An individual control (using the inherited FrameworkElement.Resources property).
2. A Window or a Page (also using the inherited FrameworkElement.Resources property).
3. An Application (using the Application.Resources property).

The variety of scopes gives you flexibility with respect to the way in which you define and share your resources.

As an alternative to directly associating your resources with a particular scope, you can package one or more resources by using a separate ResourceDictionary that can be referenced in other parts of an application. For example, the following example defines a default background color in a resource dictionary.

<ResourceDictionary

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<!-- Define background color resource -->

<SolidColorBrush x:Key="defaultBackground" Color="Red" />

<!-- Define other resources -->

...

</ResourceDictionary>

The following example references the resource dictionary defined in the previous example so that it is shared across an application.

<Application

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.App">

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="BackgroundColorResources.xaml"/>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

...

</Application>

Resources and resource dictionaries are the foundation of WPF support for themes and skins.

## Themes and Skins

The appearance that is defined by a theme defines the default appearance for a WPF application. WPF, however, does not integrate directly with Windows themes. Because the appearance of WPF is defined by templates, WPF includes one template for each of the well-known Windows themes, including Aero (Windows Vista), Classic (Microsoft Windows 2000), Luna (Microsoft Windows XP), and Royale (Microsoft Windows XP Media Center Edition 2005). These themes are packaged as resource dictionaries that are resolved if resources are not found in an application. Many applications rely on these themes to define their visual appearance; remaining consistent with Windows appearance helps users become familiar with more applications more easily.

On the other hand, the user experience for some applications does not necessarily come from the standard themes. For example, Microsoft Windows Media Player operates over audio and video data and benefits from a different style of user experience. Such UIs tend to provide customized, application-specific themes. These are known as skins, and applications that are skinned often provide hooks by which users can customize various aspects of the skin. Microsoft Windows Media Player has several prefabricated skins as well as a host of third-party skins.

Both themes and skins in WPF are most easily defined using resource dictionaries.

For more information, see "Shared Resources and Themes" in Styling and Templating.

## Custom Controls

There are three WPF models to create a new control. Each model targets a specific scenario and requires your custom control to derive from a particular WPF base class:

* **User Control Model**. A custom control derives from UserControl and is composed of one or more other controls.
* **Control Model**. A custom control derives from Control and is used to build implementations that separate their behavior from their appearance using templates, much like the majority of WPF controls. Deriving from Control allows you more freedom for creating a custom UI than user controls, but it may require more effort.
* **Framework Element Model**. A custom control derives from FrameworkElement when its appearance is defined by custom rendering logic (not templates).

$$$$

# Controls

The user experiences that are delivered by the application model are constructed controls. In WPF, "control" is an umbrella term that applies to a category of WPF classes that are hosted in either a window or a page, have a user interface (UI), and implement some behavior.

Here are the built-in WPF controls:

* Buttons: Button and RepeatButton.
* Data Display: DataGrid, ListView,and TreeView.
* Date Display and Selection: Calendar and DatePicker.
* Dialog Boxes: OpenFileDialog, PrintDialog, and SaveFileDialog.
* Digital Ink: InkCanvas and InkPresenter.
* Documents: DocumentViewer, FlowDocumentPageViewer, FlowDocumentReader, FlowDocumentScrollViewer, and StickyNoteControl.
* Input: TextBox, RichTextBox, and PasswordBox.
* Layout: Border, BulletDecorator, Canvas, DockPanel, Expander, Grid, GridView, GridSplitter, GroupBox, Panel, ResizeGrip, Separator, ScrollBar, ScrollViewer, StackPanel, Thumb, Viewbox, VirtualizingStackPanel, Window, and WrapPanel.
* Media: Image, MediaElement, and SoundPlayerAction.
* Menus: ContextMenu, Menu, and ToolBar.
* Navigation: Frame, Hyperlink, Page, NavigationWindow, and TabControl.
* Selection: CheckBox, ComboBox, ListBox, RadioButton, and Slider.
* User Information: AccessText, Label, Popup, ProgressBar, StatusBar, TextBlock, and ToolTip.

# Layouts

When you create a UI, you arrange your controls by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. Rather than forcing you to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you.

The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. In addition, the layout system manages the negotiation between controls to determine the layout. The negotiation is a two-step process: first, a control tells its parent what location and size it requires; second, the parent tells the control what space it can have.

WPF includes several layout controls:

* Canvas: Child controls provide their own layout (supports absolute positioning of its child objects).
* DockPanel: Child controls are aligned to the edges of the panel.
* Grid: Child controls are positioned by rows and columns.
* StackPanel: Child controls are stacked either vertically or horizontally.
* VirtualizingStackPanel: Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.
* WrapPanel: Child controls are positioned in left-to-right order and wrapped to the next line when there are more controls on the current line than space allows.

# Dependency Properties

Represents a property that can be set through methods such as, styling, data binding, animation, and inheritance.

Biggest features:

* built-in ability to provide change notification
* inheritance of properties in WPF tree
* sparse memory storage.

Coming from DependencyObject:

Object

L DispatcherObject

L DependencyObject

// C# standard property accessor

public double LargeurD

{

get { return (double)GetValue(LargeurDProperty); }

set { SetValue(LargeurDProperty, value); }

}

// Using a DependencyProperty as the backing store for LargeurD.

// This enables animation, styling, binding, etc...

public static readonly DependencyProperty LargeurDProperty =

DependencyProperty.Register("LargeurD", typeof(double),

typeof(ClassDP), new UIPropertyMetadata(0.0));

Typical setting of a new value:

ClassDP dp = new ClassDP();

dp.SetValue(ClassDP.LargeurDProperty, 5.5);

# Attached Properties

A property that is implemented by a parent control for use by child controls is a WPF construct called an attached property. This is a simple extension on Dependency Properties, with a static getter and a setter.

Example: DockPanel implements a Dock attached property.

public static double GetLongueur(DependencyObject obj)

{

return (double)obj.GetValue(LongueurProperty);

}

public static void SetLongueur(DependencyObject obj, double value)

{

obj.SetValue(LongueurProperty, value);

}

// Using a DependencyProperty as the backing store for Longueur.

// This enables animation, styling, binding, etc...

public static readonly DependencyProperty LongueurProperty =

DependencyProperty.RegisterAttached("Longueur", typeof(double),

typeof(ClassAP), new UIPropertyMetadata(0.0));

Typical setting of a new value:

ClassAP ap = new ClassAP();

ClassAP.SetLongueurA(ap, 3.14);

# Routed Events

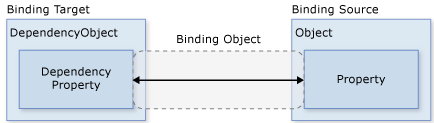
Provided by UIElement, registered using EventManager.RegisterRoutedEvent.

WPF events that support routing, that is, that can tunnel or bubble in the tree of objects.

Preview-prefixed events are tunneling events.

# Data Binding

The core unit of the data binding engine is the Binding class, whose job is to bind a control (the binding target) to a data object (the binding source).



Binding Target = Control

Binding Source = Data Object

# Graphics

Resolution-independent and device-independent graphics:

The basic unit of measurement in the WPF graphics system is the device independent pixel, which is 1/96th of an inch, regardless of actual screen resolution, and provides the foundation for resolution-independent and device-independent rendering.

## 2-D Shapes

WPF provides a library of common vector-drawn 2-D shapes, such as the rectangles and ellipses.

All shape objects inherit from the Shape class.

Available shape objects include Ellipse, Line, Path, Polygon, Polyline, and Rectangle.

Shape objects share the following common properties:

* Stroke: Describes how the shape's outline is painted.
* StrokeThickness: Describes the thickness of the shape's outline.
* Fill: Describes how the interior of the shape is painted.
* Data properties to specify coordinates and vertices, measured in device-independent pixels.

The Canvas panel is a particularly good choice for creating complex drawings because it supports absolute positioning of its child objects.

## Paths and Geometries

The Path class enables you to draw curves and complex shapes. These curves and shapes are described using Geometry objects. To use a Path, you create a Geometry and use it to set the Path object's Data property.

There are a variety of Geometry objects to choose from. The LineGeometry, RectangleGeometry, and EllipseGeometry classes describe relatively simple shapes. To create more complex shapes or create curves, use a PathGeometry.

PathGeometry and PathSegments

PathGeometry objects are comprised of one or more PathFigure objects; each PathFigure represents a different "figure" or shape. Each PathFigure is itself comprised of one or more PathSegment objects, each representing a connected portion of the figure or shape. Segment types include the following: LineSegment, BezierSegment, and ArcSegment.

XAML Abbreviated Syntax

In Extensible Application Markup Language (XAML), you may also use a special abbreviated syntax to describe a Path. In the following example, abbreviated syntax is used to draw a complex shape.

|  |  |
| --- | --- |
| <Path Stroke="DarkGoldenRod" StrokeThickness="3"  Data="M 100,200 C 100,25 400,350 400,175 H 280" /> |  |

The Data attribute string begins with the "moveto" command, indicated by M, which establishes a start point for the path in the coordinate system of the Canvas. Path data parameters are case-sensitive. The capital M indicates an absolute location for the new current point. A lowercase m would indicate relative coordinates. The first segment is a cubic Bezier curve beginning at (100,200) and ending at (400,175), drawn using the two control points (100,25) and (400,350). This segment is indicated by the C command in the Data attribute string. Again, the capital C indicates an absolute path; the lowercase c would indicate a relative path.

The second segment begins with an absolute horizontal "lineto" command H, which specifies a line drawn from the preceding subpath's endpoint (400,175) to a new endpoint (280,175). Because it is a horizontal "lineto" command, the value specified is an x-coordinate.

## Painting Shapes

Brush objects are used to paint a shape's Stroke and Fill.

Valid input for brush properties can be either a keyword or hexadecimal color value.

|  |  |
| --- | --- |
| <Canvas Background="LightGray">  <Ellipse  Canvas.Top="50" Canvas.Left="50"  Fill="#FFFFFF00"  Height="75" Width="75"  StrokeThickness="5"  Stroke="#FF0000FF"/>  </Canvas> |  |

Alternatively, you can use property element syntax to explicitly create a SolidColorBrush object to paint the shape with a solid color:

|  |  |
| --- | --- |
| <!-- This polygon shape uses pre-defined color values for its Stroke and  Fill properties.  The SolidColorBrush's Opacity property affects the fill color in  this case by making it slightly transparent (opacity of 0.4) so  that it blends with any underlying color. -->    <Polygon  Points="300,200 400,125 400,275 300,200"  Stroke="Purple"  StrokeThickness="2">  <Polygon.Fill>  <SolidColorBrush Color="Blue" Opacity="0.4"/>  </Polygon.Fill>  </Polygon> |  |

You can also paint a shape's stroke or fill with gradients, images, patterns, and more.

## Stretchable Shapes

The Line, Path, Polygon, Polyline, and Rectangle classes all have a Stretch property. This property determines how a Shape object's contents (the shape to be drawn) is stretched to fill the Shape object's layout space. A Shape object's layout space is the amount of space the Shape is allocated by the layout system, because of either an explicit Width and Height setting or because of its HorizontalAlignment and VerticalAlignment settings

The Stretch property takes one of the following values:

* None: The Shape object's contents are not stretched.
* Fill: The Shape object's contents are stretched to fill its layout space. Aspect ratio is not preserved.
* Uniform: The Shape object's contents are stretched as much as possible to fill its layout space while preserving its original aspect ratio.
* UniformToFill: The Shape object's contents are stretched to completely fill its layout space while preserving its original aspect ratio.

In the following example, a Polygon is used to draw a very small triangle from (0,0) to (0,1) to (1,1). The Polygon object's Width and Height are set to 100, and its stretch property is set to Fill. As a result, the Polygon object's contents (the triangle) are stretched to fill the larger space.

|  |  |
| --- | --- |
| <Polygon  Points="0,0 0,1 1,1"  Fill="Blue"  Width="100"  Height="100"  Stretch="Fill"  Stroke="Black"  StrokeThickness="2" /> |  |

## Transforming Shapes

The Transform class provides the means to transform shapes in a two-dimensional plane. The different types of transformation include rotation (RotateTransform), scale (ScaleTransform), skew (SkewTransform), and translation (TranslateTransform).

<Canvas>

<Rectangle

Canvas.Left="75" Canvas.Top="50"

Width="50" Height="75"

Stroke="Black">

</Rectangle>

<Polyline Points="25,25 0,50 25,75 50,50 25,25 25,0"

Stroke="LightBlue" StrokeThickness="10"

Canvas.Left="75" Canvas.Top="50">

</Polyline>

<!-- Rotates the Polyline 45 degrees about the point (0,0). -->

<Polyline Points="25,25 0,50 25,75 50,50 25,25 25,0"

Stroke="Blue" StrokeThickness="10"

Canvas.Left="75" Canvas.Top="50">

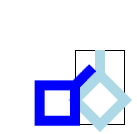
<Polyline.RenderTransform>

<RotateTransform CenterX="0" CenterY="0" Angle="45" />

</Polyline.RenderTransform>

</Polyline>

</Canvas>

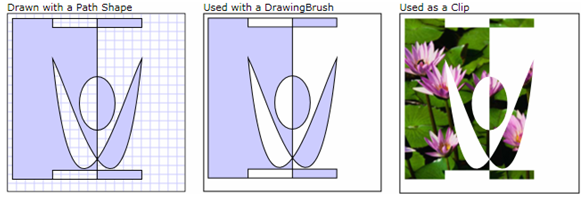


## 2-D Geometries

Path objects can be used to draw closed or open shapes, multiple shapes, and even curved shapes.

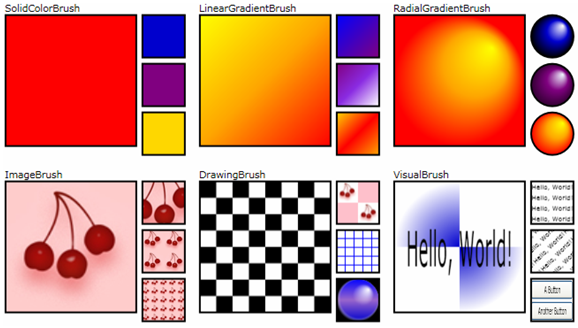
Geometry objects can be used for clipping, hit-testing, and rendering 2-D graphic data.

The following figure demonstrates the use of geometries to create a custom shape that can be drawn directly, used as a brush, or used to clip other shapes and controls.



## 2-D Effects

A subset of WPF 2-D capabilities includes visual effects, such as gradients, bitmaps, drawings, painting with videos, rotation, scaling, and skewing. These are all achieved with brushes; the following figure shows some examples.



## 3-D Rendering

WPF also includes 3-D rendering capabilities such as 2-D images rendered onto 3-D shapes.

# Animation

WPF animation support lets you make controls grow, shake, spin, and fade, to create interesting page transitions, and more. You can animate most WPF classes, even custom classes.

# Media

WPF provides special support for images, video, and audio.

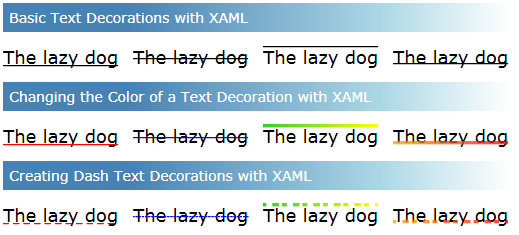
## Images

For more information, see Imaging Overview.

## Video and Audio

The MediaElement control is capable of playing both video and audio, and it is flexible enough to be the basis for a custom media player.

# Text and Typography



For more information, see Typography in WPF.

# Documents

WPF has native support for working with three types of documents: flow documents, fixed documents, and XML Paper Specification (XPS) documents. WPF also provides the services to create, view, manage, annotate, package, and print documents.

For more information, see Documents in WPF

## Flow Documents

Flow documents are designed to optimize viewing and readability by dynamically adjusting and reflowing content when window size and display settings change. The following XAML markup shows the definition of a FlowDocument.

<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

<Paragraph FontSize="18" FontWeight="Bold">Flow Document</Paragraph>

<Paragraph>

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy

nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi

enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.

</Paragraph>

...

</FlowDocument>

The following example demonstrates how to load a flow document into a FlowDocumentReader for viewing, searching, and printing.

<Window

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="SDKSample.FlowDocumentReaderWindow"

Title="Flow Document Reader">

<FlowDocumentReader Name="flowDocumentReader" />

</Window>

using System.Windows; // Window

using System.Windows.Documents; // FlowDocument

using System.IO; // FileStream, FileMode

using System.Windows.Markup; // XamlReader

namespace SDKSample

{

public partial class FlowDocumentReaderWindow : System.Windows.Window

{

public FlowDocumentReaderWindow()

{

InitializeComponent();

// Open the file that contains the FlowDocument

using (FileStream xamlFile = new FileStream("AFlowDocument.xaml",

FileMode.Open, FileAccess.Read))

{

// Parse the file with the XamlReader.Load method

FlowDocument content = XamlReader.Load(xamlFile) as FlowDocument;

// Set the Document property to the parsed FlowDocument object

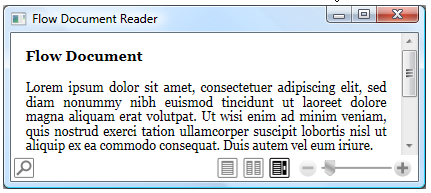
this.flowDocumentReader.Document = content;

}

}

}

}



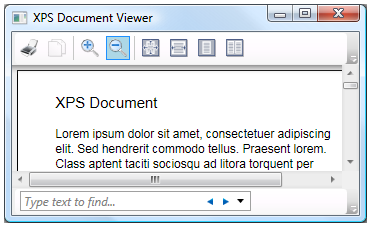
## Fixed Documents

Fixed documents are intended for applications that require a precise "what you see is what you get" (WYSIWYG) presentation, particularly with respect to printing. Typical uses for fixed documents include desktop publishing, word processing, and form layout, where adherence to the original page design is critical.

## XPS Documents

XML Paper Specification (XPS) documents build on WPF's fixed documents.

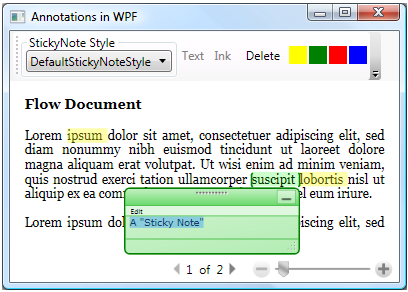
The following figure shows an XPS document that is displayed by a DocumentViewer.



DocumentViewer also allows users to change the view, search, and print XPS documents.

## Annotations

In WPF, an annotations system is provided to support sticky notes and highlights. These annotations can be applied to documents hosted in the DocumentViewer control:



## Packaging

The WPF System.IO.Packaging APIs allow your applications to organize data, content, and resources into single, portable, easy-to-distribute, and easy-to-access ZIP documents. Digital signatures can be included to authenticate items that are contained in a package and to verify that the signed item was not tampered with or modified. You can also encrypt packages by using rights management in order to restrict access to protected information.

## Printing

The .NET Framework includes a printing subsystem that WPF augments with support for enhanced print system control.

# XAML

## XAML Object Elements

An object element typically declares an instance of a type. That type is defined in the assemblies that provide the backing types for a technology that uses XAML as a language.

Object element syntax always starts with an opening angle bracket (<). This is followed by the name of the type where you want to create an instance.

<StackPanel>

<Button Content="Click Me"/>

</StackPanel>

This specifies two object elements: <StackPanel> (with content, and a closing tag later), and <Button .../> (the self-closing form, with several attributes). The object elements StackPanel and Button each map to the name of a class that is defined by WPF and is part of the WPF assemblies. When you specify an object element tag, you create an instruction for XAML processing to create a new instance. Each instance is created by calling the default constructor of the underlying type when parsing and loading the XAML.

## Attribute Syntax (Properties)

An attribute syntax names the property that is being set in attribute syntax, followed by the assignment operator (=). The value of an attribute is always specified as a string that is contained within quotation marks.

<Button Background="Blue" Foreground="Red" Content="This is a button"/>

## Property Element Syntax

For some properties of an object element, attribute syntax is not possible, because the object or information necessary to provide the property value cannot be adequately expressed within the quotation mark and string restrictions of attribute syntax. For these cases, a different syntax known as property element syntax can be used.

The syntax for the property element start tag is <typeName.propertyName>. Generally, the content of that tag is an object element of the type that the property takes as its value.

<Button>

<Button.Background>

<SolidColorBrush Color="Blue"/>

</Button.Background>

<Button.Foreground>

<SolidColorBrush Color="Red"/>

</Button.Foreground>

<Button.Content>

This is a button

</Button.Content>

</Button>

## Collection Syntax

if a particular property takes a collection type, then items that you declare in markup as child elements within that property's value become part of the collection. In this case a collection of child object elements is the value being set to the collection property.

<LinearGradientBrush>

<LinearGradientBrush.GradientStops>

<!-- no explicit new GradientStopCollection, parser knows how to find or create -->

<GradientStop Offset="0.0" Color="Red" />

<GradientStop Offset="1.0" Color="Blue" />

</LinearGradientBrush.GradientStops>

</LinearGradientBrush>

## XAML Content Properties

XAML specifies a language feature whereby a class can designate exactly one of its properties to be the XAML content property. Child elements of that object element are used to set the value of that content property. In other words, for the content property uniquely, you can omit a property element when setting that property in XAML markup and produce a more visible parent/child metaphor in the markup.

For example, Border specifies a content property of Child. The following two Border elements are treated identically. The first one takes advantage of the content property syntax and omits the Border.Child property element. The second one shows Border.Child explicitly.

<Border>

<TextBox Width="300"/>

</Border>

<!--explicit equivalent-->

<Border>

<Border.Child>

<TextBox Width="300"/>

</Border.Child>

</Border>

# Layout

Examples of attached properties: Grid.Row and Grid.Column

Grid.SetColumn(myPath, 0);

Grid.SetRow(myPath, 0);

Grid.SetRow is a static method of Grid

(read specific doc on layout)

# Application Startup

If App.xaml defines for Application element an attribute StartupUri="MainWindow.xaml", in this case it’s automatic.

Or in App constructor, define it, same thing:

public partial class App : Application

{

public App()

{

//System.Diagnostics.Debugger.Break();

StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);

}

}

Or handle Startup event for App: define for Application element an attribute Startup="Application\_Startup", then in the code behind instantiate a new window and show it calling its Show() method:

private void Application\_Startup(object sender, StartupEventArgs e)

{

Window myMainWindow = new MainWindow();

myMainWindow.Show();

}

For new window creation:

When new MainWindow() is executed:

* Constructor is called first; which in turn calls InitializeComponent (autogenerated from xaml file) to parse and load xaml

When myMainWindow.Show() is executed, it triggers following events:

* Window\_Initialized
* Window\_IsVisibleChanged
* Window\_SizeChanged
* Window\_LayoutUpdated
* Window\_SourceInitialized
* Window\_Activated
* Window\_GotKeyboardFocus
* Window\_LayoutUpdated
* Window\_Loaded
* Window\_ContentRendered

When window is closed:

* Window\_Closing
* Window\_IsVisibleChanged
* Window\_Deactivated
* Window\_LostKeyboardFocus
* Window\_Closed

# Data Binding

<ListBox … ItemsSource ="{Binding ,Path=… }" />

Ø (no source) implies DataContext

DataContext can be defined :

* In procedural code

myGrid.DataContext = b ;

* In XAML

<Window.Resources>

<local:DataBag x:Key="dataBag"/>

…

<Grid DataContext="{StaticResource dataBag}" …

When binding to a collection :

* "{Binding Path=/}" binds to current item
* "{Binding Path=/myProperty}" binds to myProperty of current item
* "{Binding Path=Photos/}" binds to current item of a property called Photos exposing a collection

# Styles

Simple use : define Style in Resources, give it a key and use it

<X.Resources>

<Style x:Key="buttonStyle">

<Setter Property="Button.FontSize" Value="20"/>

<Setter Property="Render.Transform">

<Setter.Value>

<RotateTransform Angle="10"/>

</Setter.Value>

</Setter>

</Style>

</X.Resources>

<Grid>

<Button Style="{StaticResource buttonStyle}">Text</Button>

</Grid>

Use restricted to a specific type: add a TargetType to Style definition, applicable to elements of type TargetType or a derived type

<X.Resources>

<Style x:Key="buttonStyle" TargetType="{x:Type Button}">

<Setter Property="FontSize" Value="20"/>

</Style>

</X.Resources>

<Grid>

<Button Style="{StaticResource buttonStyle}">Text</Button>

</Grid>

Implicit Styles or Typed Styles: Assign a TargetType and define no x:Key, applicable automatically to all subelements matching exactly TargetType

<X.Resources>

<Style TargetType="{x:Type Button}">

<Setter Property="FontSize" Value="20"/>

</Style>

</X.Resources>

<Grid>

<Button Style="{StaticResource buttonStyle}">Text</Button>

</Grid>

# Triggers

Three types of triggers:

* Property triggers, invoked when the value of a dependency property changes
* Data Triggers, invoked when the value of a plain .Net property changes
* Event triggers, invoked when a routed event is raised

Logical place to put a trigger: Styles, even if FrameworkElement, DataTemplate and ControlTemplate also have a Triggers collection.

## Property Triggers

Execute a collection of setters when a specified property has a specified value. When the property no longer has the value, the trigger “undoes” the setters.

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Blue"/>

<Setter Property="FontSize" Value="24"/>

<Setter Property="FontWeight" Value="Bold"/>

</Trigger>

Note: it’s not possible to apply setters in a trigger directly to an object such as a button, the trick is to define them inside a style applied to the element:

<Button MinWidth="75" Margin="10" Click="OkButton\_Click">

<Button.Style>

<Style TargetType="{x:Type Button}">

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Blue"/>

<Setter Property="FontSize" Value="24"/>

<Setter Property="FontWeight" Value="Bold"/>

</Trigger>

</Style.Triggers>

</Style>

</Button.Style>

\_OK

</Button>

## Property Triggers

Similar to triggers, but can be triggered by any .Net property rather than just dependency properties. Setters are sill restricted to setting DP.

<DataTrigger Binding="{Binding RelativeSource={RelativeSource Self}, Path=Text}" Value="disabled">

<Setter Property="IsEnabled" Value="False"/>

</DataTrigger>

## MultiTriggers

MultiTriggers implement a And between multiple conditions:

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsIndeterminate" Value="True"/>

<Condition Property="IsEnabled" Value="True"/>

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter TargetName="pie" Property="Visibility" Value="Hidden"/>

<Setter TargetName="background" Property="Fill">

<Setter.Value>

<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">

<GradientStop Offset="0" Color="Yellow"/>

<GradientStop Offset="1" Color="Brown"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

</MultiTrigger.Setters>

</MultiTrigger>

# Templates

Used to redefine visual tree of a control to almost anything.

Can have resources and styles (that can contain setters and triggers).

As with Styles, ControlTemplate has a TargetType property, but this doesn’t enable to remove x:Key in a resources dictionary since there is no notion of default template (put it in a typed style instead).

<ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type Button}">

## VisualStateManager

Used by control designed to maintain separate groups of mutually exclusive statuses, to ease visual state management.

Full example, see WPF32 Templates (2)\App.xaml.

<ControlTemplate x:Key="progressPie" TargetType="{x:Type ProgressBar}">

….

<!-- Visual State Groupe -->

<VisualStateManager.VisualStateGroups>

<VisualStateGroup Name="CommonStates">

<VisualState Name="Determinate" /> <!-- Nothing to do -->

<VisualState Name="Indeterminate">

<Storyboard>

<DoubleAnimation Storyboard.TargetName="pie" Storyboard.TargetProperty="Opacity" To="0" Duration="0:0:1"/>

<DoubleAnimation Storyboard.TargetName="backgroundNormal" Storyboard.TargetProperty="Opacity" To="0" Duration="0:0:1"/>

<DoubleAnimation Storyboard.TargetName="backgroundIndeterminate" Storyboard.TargetProperty="Opacity" To="1" Duration="0:0:1"/>

</Storyboard>

</VisualState>

</VisualStateGroup>

</VisualStateManager.VisualStateGroups>

….

</ControlTemplate>

# 2D Graphics

WPF works in retained mode, as opposed to GDI, GDI+, DirectX that use immediate mode.

Drawings are simple descriptions of paths and shapes associated with fill and outline brushes.

Visuals are one way to put drawings on a screen, but also give access to lower-level lightweight approach.

Shapes are prebuilt visuals (heavyweight) for drawing custom artwork on screen (and the only one exposed by SilverLight)

## Drawings

Abstract class = 2D drawing. Subclasses:

* GeometryDrawing is WPF version of clipart, combines a geometry with a brush and a pen.
* ImageDrawing, combines an ImageSource and a Rect to add bitmaps
* VideoDrawing, combines a MediaPlayer and a bounding rect
* GlyphRunDrawing, low-level text class with a brush
* DrawingGroup, a collection of drawings, with properties to change them in bulk (Opacity, Transform, …). DrawingGroup is itself a Drawing, same relation between TransformGroup and Transform.

Example (inside is EllipseGeometry, not Ellipse!):

<GeometryDrawing Brush="Orange">

<GeometryDrawing.Pen>

<Pen Brush="Black" Thickness="5"/>

</GeometryDrawing.Pen>

<GeometryDrawing.Geometry>

<EllipseGeometry RadiusX="100" RadiusY="50"/>

</GeometryDrawing.Geometry>

</GeometryDrawing>

Drawings are not UIElements (no rendering behavior of their own). To render them, host them in:

* DrawingImage, derives from ImageSource, so can be used inside an image rather than a typical BitmapImage.
* DrawingBrush, derives from Brush, so can be used in many places.
* DrawingVisual, derives from Visual.

Example to display previous GeometryDrawing:

<Image>

<Image.Source>

<DrawingImage>

<DrawingImage.Drawing>

<GeometryDrawing Brush="Orange">

<GeometryDrawing.Pen>

<Pen Brush="Black" Thickness="5"/>

</GeometryDrawing.Pen>

<GeometryDrawing.Geometry>

<EllipseGeometry RadiusX="100" RadiusY="50"/>

</GeometryDrawing.Geometry>

</GeometryDrawing>

</DrawingImage.Drawing>

</DrawingImage>

</Image.Source>

</Image>

ImageDrawing vs DrawingImage

For all graphics classes, class FooBar is a Bar that contains or acts like a Foo

### Geometries

4 basic types, the first three are just special cases of the fourth provided for convenience.

* RectangleGeometry: Rect, RadiusX and RadiusY
* EllipseGeometry: Center, RadiusX and RadiusY
* LineGeometry: StartPoint, EndPoint
* PathGeometry: Collection of PathFigures objects

Each PathFigure in a PathGeometry contains one or more connected PathSegments in its Segments content property.

PathSegments:

* LineSegment
* PolyLineSegment: a connected sequence of LineSegment
* ArcSegment
* BezierSegment
* PolyBezierSegment: a connected sequence of BezierSegment
* QuadraticBezierSegment
* PolyQuadraticBezierSegment: a connected sequence of QuadraticBezierSegment

Geometries aggregations: (both derive from Geometry)

* GeometryGroup: just a combination of several PathGeometry, each one can have its own properties such as transform (not LayoutTransform or RenderTransform!).
* CombinedGeometry: Union/Intersect/Xor/Exclude between two geometries

## Visuals

xsss

## Shapes

x

# Form Events

When a form is created:

Window\_Initialized

Window\_IsVisibleChanged

Window\_SizeChanged

Window\_LayoutUpdated

Window\_SourceInitialized

Window\_Activated

Window\_PreviewGotKeyboardFocus

Window\_IsKeyboardFocusWithinChanged

Window\_IsKeyboardFocusedChanged

Window\_GotKeyboardFocus

Window\_LayoutUpdated

Window\_Loaded

Window\_ContentRendered

On closing (using Alt+F4) :

Window\_PreviewKeyDown

Window\_KeyDown

Window\_PreviewKeyDown

Window\_KeyDown

Window\_Closing

Window\_IsVisibleChanged

Window\_Deactivated

Window\_IsKeyboardFocusWithinChanged

Window\_IsKeyboardFocusedChanged

Window\_LostKeyboardFocus

Window\_Closed

On a right-click:

Window\_PreviewMouseRightButtonDown

Window\_PreviewMouseDown

Window\_MouseRightButtonDown

Window\_MouseDown

Window\_PreviewMouseRightButtonUp

Window\_PreviewMouseUp

Window\_MouseRightButtonUp

Window\_MouseUp

Window\_ContextMenuOpening

On a key press (f):

Window\_PreviewKeyDown

Window\_KeyDown

Window\_PreviewTextInput

Window\_TextInput

Window\_PreviewKeyUp

Window\_KeyUp

# Animations

Example in a style with Events triggers and properties triggers:

<Style TargetType="{x:Type Button}">

<Setter Property="VerticalAlignment" Value="Bottom"/>

<Setter Property="LayoutTransform">

<Setter.Value>

<TransformGroup>

<ScaleTransform/>

<RotateTransform/>

</TransformGroup>

</Setter.Value>

</Setter>

<Setter Property="RenderTransform">

<Setter.Value>

<TranslateTransform/>

</Setter.Value>

</Setter>

<Style.Triggers>

<!-- Event triggers -->

<EventTrigger RoutedEvent="Button.MouseEnter">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard TargetProperty="Width">

<DoubleAnimation To="2" Duration="0:0:0.25"

Storyboard.TargetProperty="LayoutTransform.Children[0].ScaleX" />

<DoubleAnimation To="2" Duration="0:0:0.25"

Storyboard.TargetProperty="LayoutTransform.Children[0].ScaleY" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

<EventTrigger RoutedEvent="Button.MouseLeave">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation To="1" Duration="0:0:0.25"

Storyboard.TargetProperty="LayoutTransform.Children[0].ScaleX" />

<DoubleAnimation To="1" Duration="0:0:0.25"

Storyboard.TargetProperty="LayoutTransform.Children[0].ScaleY" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

<!-- Properties triggers -->

<Trigger Property="IsMouseOver" Value="True">

<Trigger.EnterActions>

<BeginStoryboard>

<Storyboard TargetProperty="RenderTransform.Y">

<DoubleAnimation To="-20" Duration="0:0:0.25"/>

</Storyboard>

</BeginStoryboard>

</Trigger.EnterActions>

<Trigger.ExitActions>

<BeginStoryboard>

<Storyboard TargetProperty="RenderTransform.Y">

<DoubleAnimation To="0" Duration="0:0:0.25"/>

</Storyboard>

</BeginStoryboard>

</Trigger.ExitActions>

</Trigger>

</Style.Triggers>

</Style>