WPF Graphical rendering (2 D)

WPF Resources and animations

Styles and Templates

WPF 2D Graphical rendering (since .NET 4.0)

- *Principle in WPF 2D : Retained mode graphics*
  - XAML or code is used
- *Principle in GDI: Immediate mode Graphics*
  - **Programmer is responsible for graphical update**

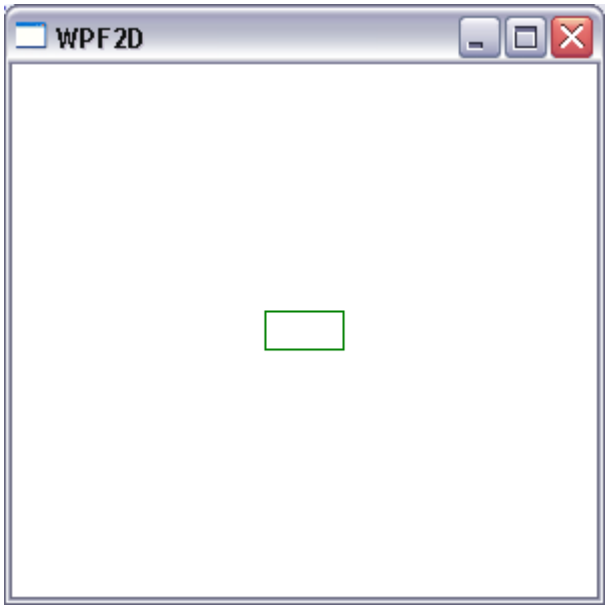
WPF namespaces

- System.Windows.Shapes
  - Geometric objects: Rectangles, ellipses, polygones and much more
- System.Windows.Media.Drawing
  - Image and drawing specific – ightweight but less feature-rich
- System.Windows.Media.Visual
  - Fastest and lightweight rendering of graphical data

Note WPF also ships with a full-blown API that can be used to render and manipulate 3D graphics, which is not addressed in this edition of the text. Please consult the .NET Framework 4.7 SDK documentation if you are interested in incorporating 3D graphics into your applications.

Shapes: Example (XAML based)

(Very Simple Example)

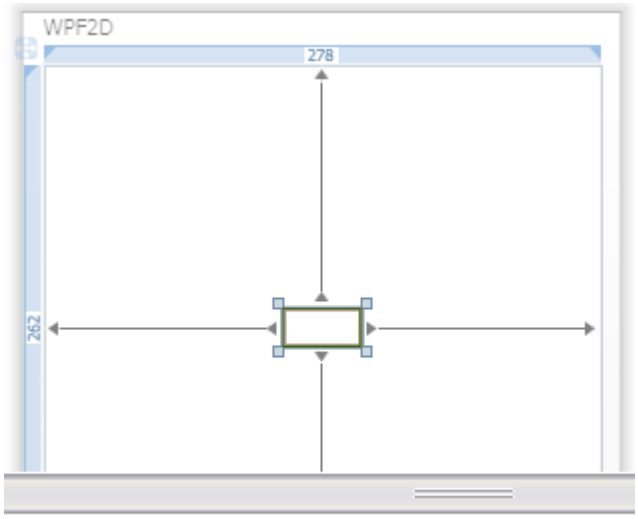


XAML code

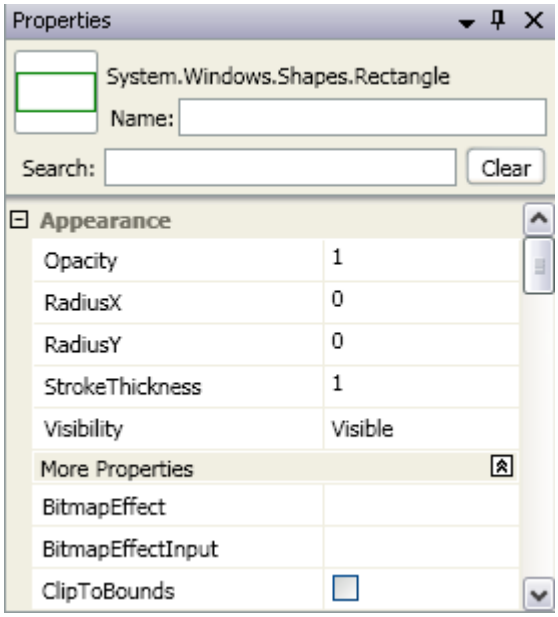
```
Window x:Class="RenderingEksempel.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF2D" Height="300" Width="300">
    <Grid>
        <Rectangle Height="20" Width="40" Stroke="Green"
Margin="20,40"></Rectangle>
    </Grid>
</Window>
```

Properties

- Full access to properties in Designer




```
Height="20" Width="40" Stroke="Green" Ma
```



Inheritance hierarchy for Shapes:

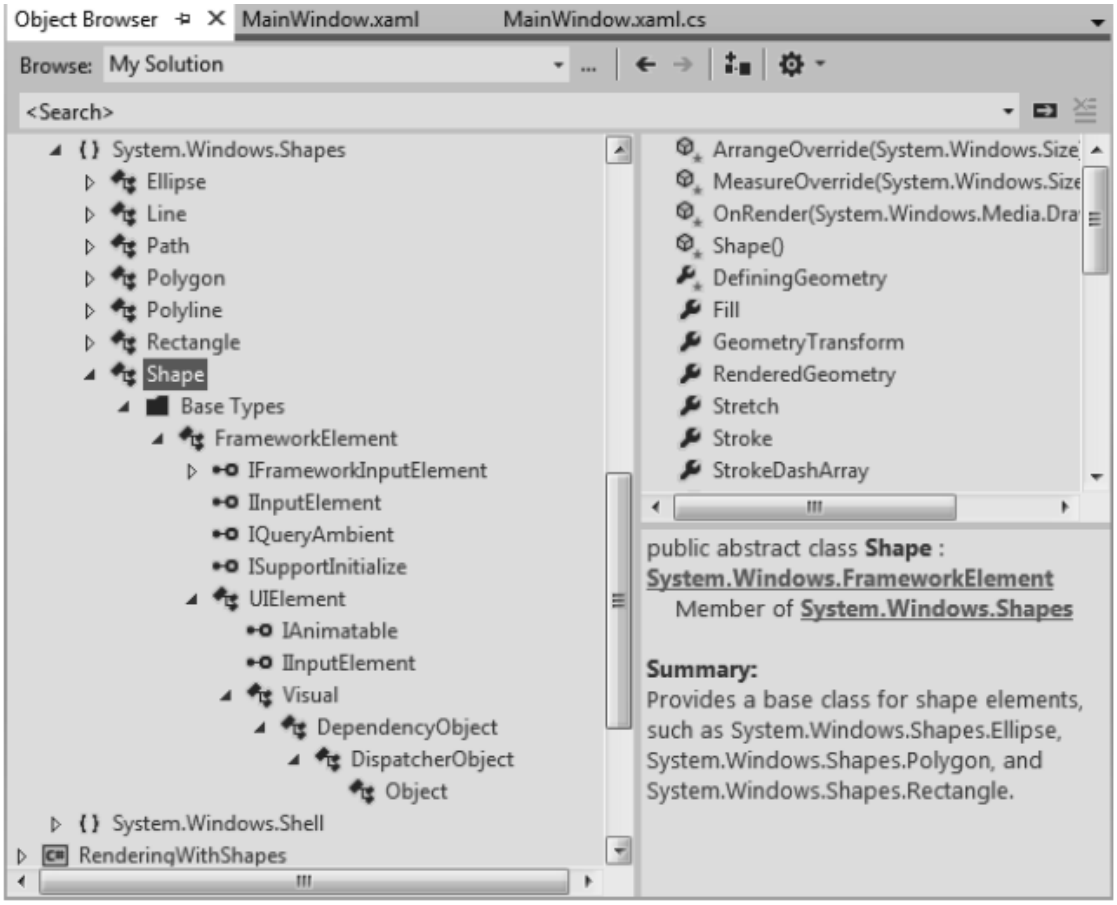


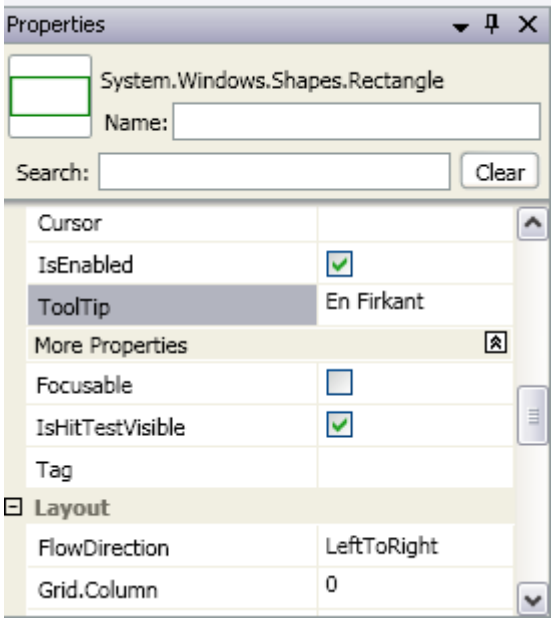
Figure 29-1. *The Shape base class receives a good deal of functionality from its parent classes*

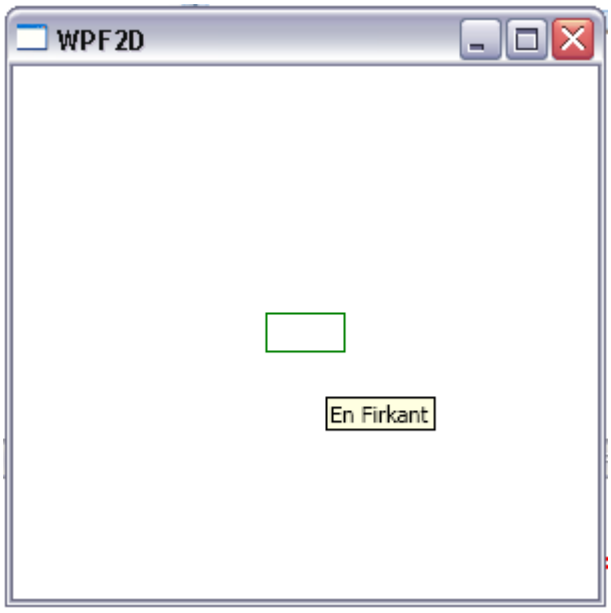
The Shape class

Table 29-1. *Key Properties of the Shape Base Class*

Properties	Meaning in Life
DefiningGeometry	Returns a Geometry object that represents the overall dimensions of the current shape. This object contains *only* the plot points that are used to render the data, and has no trace of the functionality from UIElement or FrameworkElement.
Fill	Allows you to specify a "brush object" to render the interior portion of a shape.
GeometryTransform	Allows you to apply transformations to a shape, *before* it is rendered on the screen. The inherited RenderTransform property (from UIElement) applies the transformation *after* it has been rendered on the screen.
Stretch	Describes how to fill a shape within its allocated space, such as its position within a layout manager. This is controlled using the corresponding System.Windows.Media.Stretch enumeration.
Stroke	Defines a brush object, or in some cases, a pen object (which is really a brush in disguise) that is used to paint the border of a shape.
StrokeDashArray, StrokeEndLineCap, StrokeStartLineCap,	These (and other) stroke-related properties control how lines are configured when drawing the border of a shape. In a majority of cases, these properties will

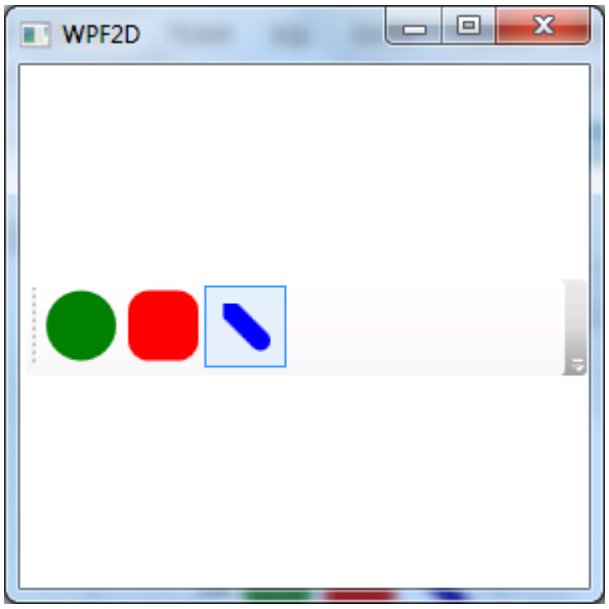
ToolTip is built-in





RadioButton

```
<Window x:Class=" GraphicsTest.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF2D" Height="300" Width="300"><Grid>
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
            <RadioButton Name="circleOption" GroupName="shapeSelection">
                <Ellipse Fill="Green" Height="35" Width="35" />
            </RadioButton>
            <RadioButton Name="rectOption" GroupName="shapeSelection">
                <Rectangle Fill="Red" Height="35"
Width="35" RadiusY="10" RadiusX="10" />
            </RadioButton>
            <RadioButton Name="lineOption" GroupName="shapeSelection">
                <Line Height="35" Width="35"
StrokeThickness="10" Stroke="Blue"
X1="10" Y1="10" Y2="25" X2="25"
StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
            </RadioButton>
        </ToolBar>
</Grid>
</Window>
```



Adding an event to ”Circle” <RadioButton>

```
<Window x:Class="GraphicsTest.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF2D" Height="300" Width="300">
    <Grid>
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
            <RadioButton Name="circleOption" GroupName="shapeSelection"
Checked="Circle_Click">
                <Ellipse Fill="Green" Height="35" Width="35" />
            </RadioButton>
            <RadioButton Name="rectOption" GroupName="shapeSelection">
                <Rectangle Fill="Red" Height="35"
Width="35" RadiusY="10" RadiusX="10" />
            </RadioButton>
            <RadioButton Name="lineOption" GroupName="shapeSelection">
                <Line Height="35" Width="35"
StrokeThickness="10" Stroke="Blue"
X1="10" Y1="10" Y2="25" X2="25"
StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
            </RadioButton>
        </ToolBar>
</Grid>
</Window>
```

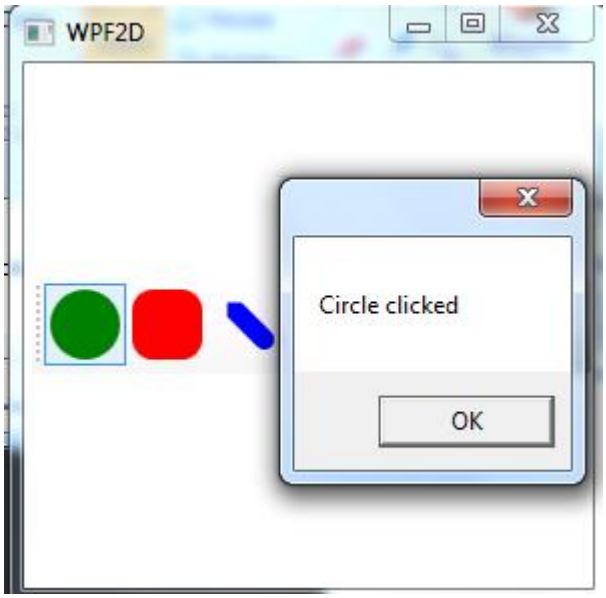
```
...
        protected void Circle_Click(object sender, RoutedEventArgs rea)
        {
            MessageBox.Show("Circle clicked");
        }
...
```

- Or use an enum:

```
public partial class MainWindow : Window
{
  private enum SelectedShape
  { Circle, Rectangle, Line }
  private SelectedShape _currentShape;
}
```

Within each `Click` event handler, set the `currentShape` member variable to the correct `SelectedShape` value, as follows:

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
  _currentShape = SelectedShape.Circle;
}
```



Adding shapes on run-time

Method 1:

```
…
        Shape shape = null;
         Canvas myCanvas = new Canvas();

…

        private void MenuItem_Click(object sender, RoutedEventArgs e)
         {
              shape = new Ellipse();
              shape.Height = 20;
              shape.Width = 20;

              SolidColorBrush mySolidColorBrush = new SolidColorBrush();


              mySolidColorBrush.Color = Color.FromArgb(100, 100, 100, 0);
              shape.Fill = mySolidColorBrush;
              shape.StrokeThickness = 2;
              shape.Stroke = Brushes.Black;


              Canvas.SetLeft(shape, 250);
              Canvas.SetTop(shape, 250);

              myCanvas.Children.Add(shape);

              this.Content = myCanvas;

         }


…
```

Visual Types

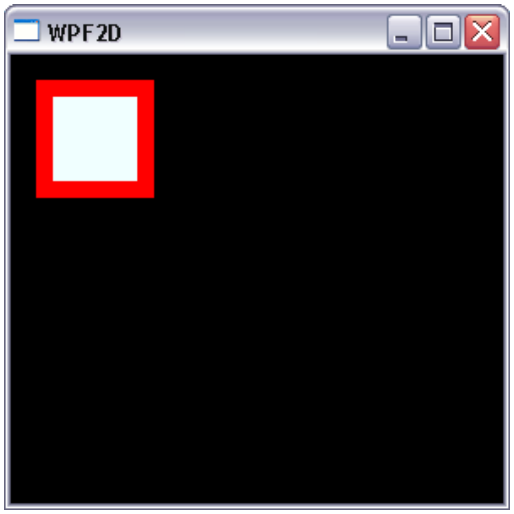
Method 2:

- Restricted support via XAML
- Uses C# code

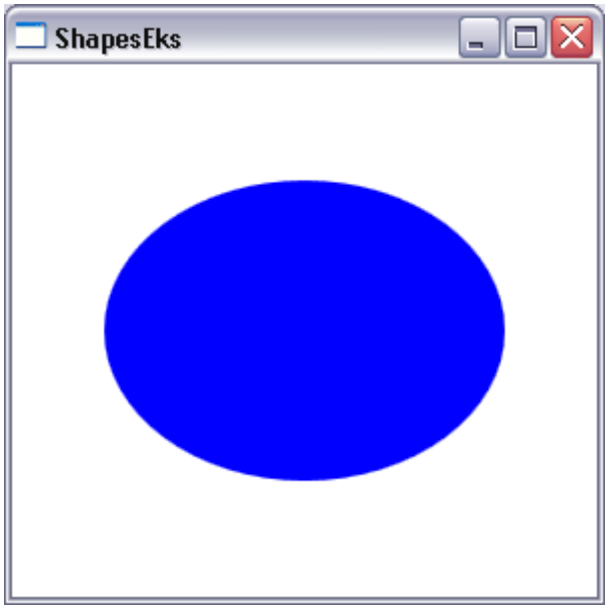
Example using Visual types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace RenderingEksempel
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        private DrawingVisual dv = new DrawingVisual();
        private const int antal = 1;
        public Window1()
        {
            InitializeComponent();
            TegnFirkant();
        }
        public void TegnFirkant()
        {

            DrawingContext dc = dv.RenderOpen();

            Rect enFirkant = new Rect(20, 20, 60, 60);

            dc.DrawRectangle(Brushes.Azure,
                             new Pen(Brushes.Red, 10), enFirkant);
            dc.Close();


            AddVisualChild(dv);
            AddLogicalChild(dv);
        }

        protected override int VisualChildrenCount
        {
            get
            {
                return antal;
            }
        }
        protected override Visual GetVisualChild(int index)
        {
            return dv;
        }
    }
}
```

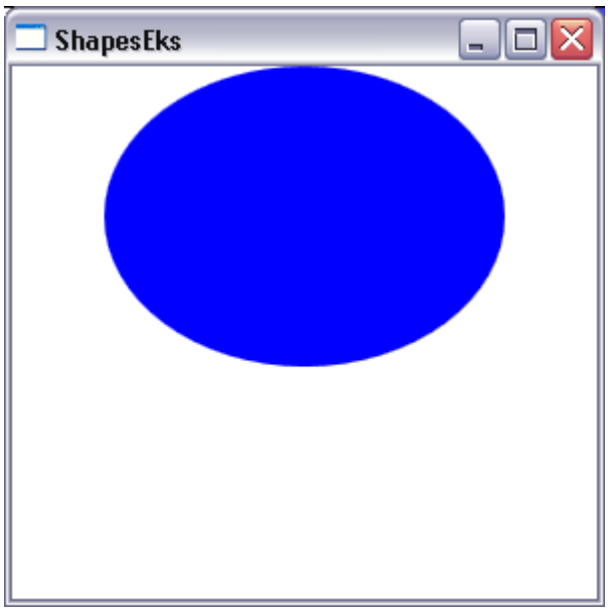
- The last two methods **MUST be override**



Examples from Shapes



Example A: (Ellipse drawn directly inside the Window tag)



Example B: Use of StackPanel

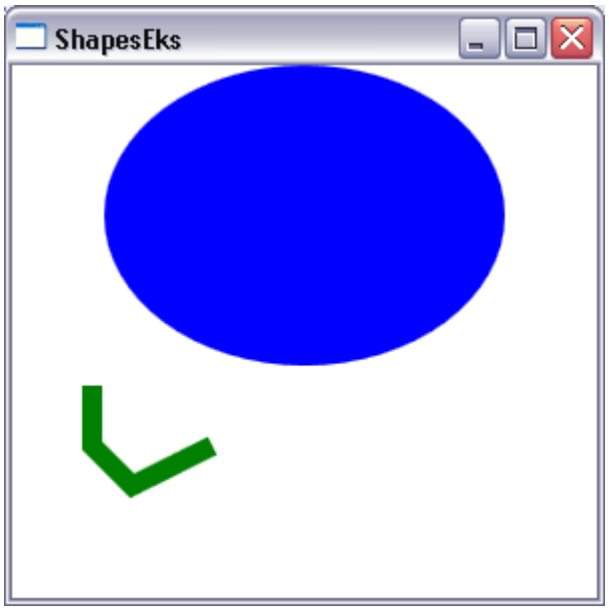
A

```
<Window x:Class="RenderingEksempel.ShapesEks"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ShapesEks" Height="300" Width="300">
    <Ellipse Name="A"  Height="150" Width="200" Fill="Blue"/>
    </Window>
```

B

```
<Window x:Class="RenderingEksempel.ShapesEks"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ShapesEks" Height="300" Width="300">
    <StackPanel>
    <Ellipse Name="A"  Height="150" Width="200" Fill="Blue"/>
        </StackPanel>
    </Window>
```

PolyLines



XAML

```
Window x:Class="RenderingEksempel.ShapesEks"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ShapesEks" Height="300" Width="300">
    <StackPanel>
```

```
    <Ellipse Name="A"  Height="150" Width="200" Fill="Blue"/>
        <Polyline Stroke="Green" StrokeThickness="10" Points="40,10 40,40 60,60
                  100,40
                  "></Polyline>
        </StackPanel>
    </Window>
```

System.Windows.Media.Geometry

- geometrical areas

Table 29-2. *Select Members of the System.Windows.Media.Geometry Type*

Member	Meaning in Life
Bounds	Establishes the current bounding rectangle containing the geometry.
FillContains()	Determines whether a given Point (or other Geometry object) is within the bounds of a particular Geometry-derived class. This is useful for hit-testing calculations.
GetArea()	Returns the entire area a Geometry-derived type occupies.
GetRenderBounds()	Returns a Rect that contains the smallest possible rectangle that could be used to render

Table 29-3. *Geometry-Derived Classes*

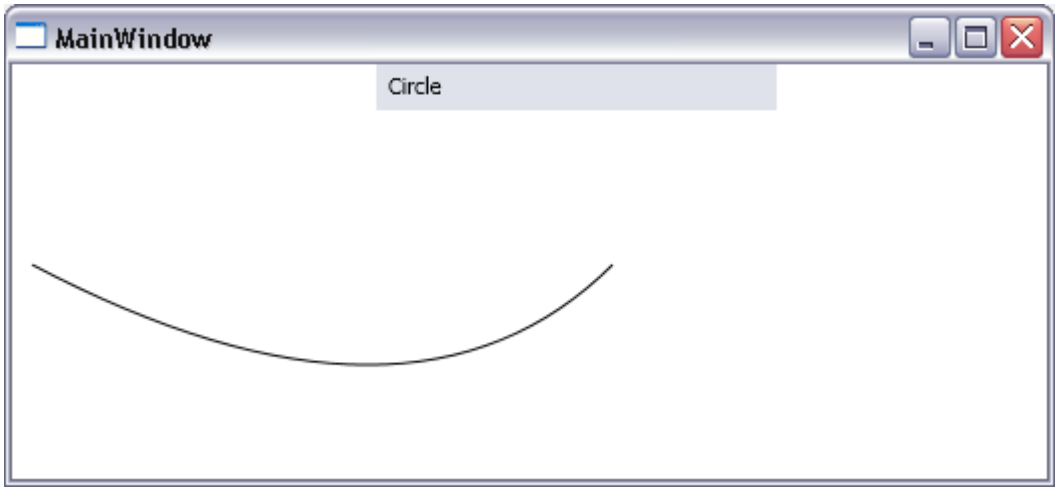
Geometry Class	Meaning in Life
LineGeometry	Represents a straight line.
RectangleGeometry	Represents a rectangle.
EllipseGeometry	Represents an ellipse.
GeometryGroup	Allows you to group together several Geometry objects.
CombinedGeometry	Allows you to merge two different Geometry objects into a single shape.
PathGeometry	Represents a figure composed of lines and curves.

- Notice: <GeometryGroup>
- Example

```
...
<!-- A Path contains a set of geometry objects,
set with the Data property. -->
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
<Path.Data>
<GeometryGroup>
<EllipseGeometry Center = "75,70"
RadiusX = "30" RadiusY = "30" />
<RectangleGeometry Rect = "25,55 100 30" />
<LineGeometry StartPoint="0,0" EndPoint="70,30" />
<LineGeometry StartPoint="70,30" EndPoint="0,30" />
</GeometryGroup>
</Path.Data>
</Path>
…
```

Path

- Can for instance draw Bezier curves



```
<Window x:Class="RenderingsEksempler.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Menu Height="23" HorizontalAlignment="Left" Margin="182,0,0,0" Name="menu1"
VerticalAlignment="Top" Width="200">
            <MenuItem Header="Circle" Click="MenuItem_Click">
                <MenuItem Header="Circle" />
                <MenuItem Header="Rectangle" />
            </MenuItem>
        </Menu>



        <Path Stroke="Black" StrokeThickness="1">
            <Path.Data>
                <PathGeometry>
                    <PathGeometry.Figures>
                        <PathFigureCollection>
                            <PathFigure StartPoint="10,100">
                                <PathFigure.Segments>
                                    <PathSegmentCollection>
                                        <QuadraticBezierSegment Point1="200,200"
                                                                Point2="300,100" />
                                    </PathSegmentCollection>
                                </PathFigure.Segments>
                            </PathFigure>
                        </PathFigureCollection>
                    </PathGeometry.Figures>
                </PathGeometry>
            </Path.Data>
        </Path>

    </Grid>
</Window>
```

The Path Modeling "Mini-Language"

Of all the classes listed in Table 26-3, `PathGeometry` is the most complex to configure in terms of XAML or code. This has to do with the fact that each *segment* of the `PathGeometry` is composed of objects that contain various segments and figures (for example, `ArcSegment`, `BezierSegment`, `LineSegment`, `PolyBezierSegment`, `PolyLineSegment`, `PolyQuadraticBezierSegment`, etc.). Here is an example of a `Path` object whose `Data` property has been set to a `<PathGeometry>` composed of various figures and segments:

- Another example:

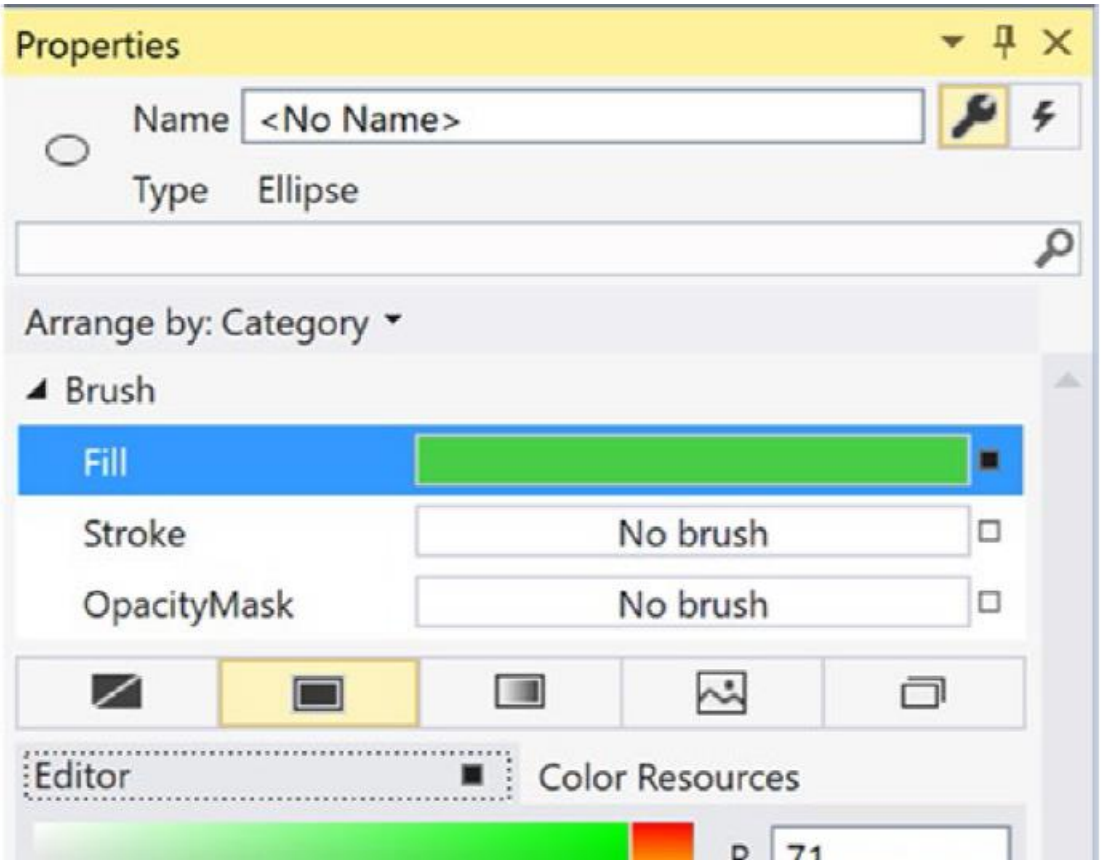
```
<Path Stroke="Black" StrokeThickness="1" >
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
           <BezierSegment
             Point1="100,0"
             Point2="200,200"
             Point3="300,100"/>
           <LineSegment Point="400,100" />
           <ArcSegment
             Size="50,50" RotationAngle="45"
             IsLargeArc="True" SweepDirection="Clockwise"
             Point="200,100"/>
           </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
```

WPF Brushes

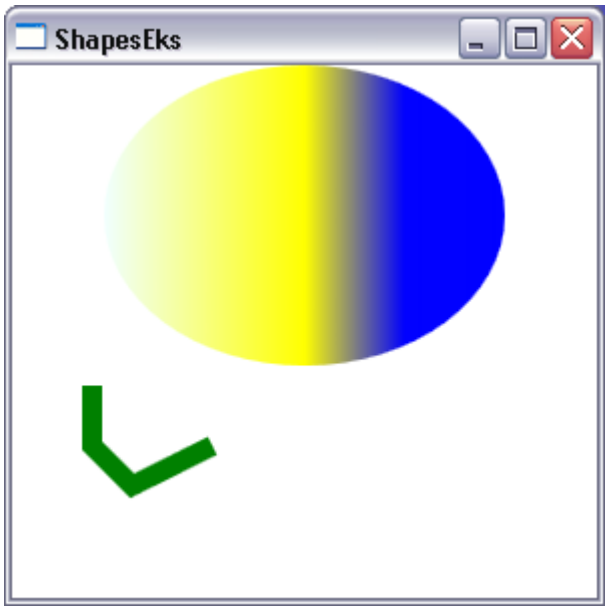
Table 29-4. WPF Brush-Derived Types

Brush Type	Meaning in Life
DrawingBrush	Paints an area with a Drawing-derived object (GeometryDrawing, ImageDrawing, or VideoDrawing)
ImageBrush	Paints an area with an image (represented by an ImageSource object)
LinearGradientBrush	Paints an area with a linear gradient
RadialGradientBrush	Paints an area with a radial gradient
SolidColorBrush	Paints a single color, set with the Color property
VisualBrush	Paints an area with a Visual-derived object (DrawingVisual, Viewport3DVisual, and ContainerVisual)

Configuration of Brushes using VS

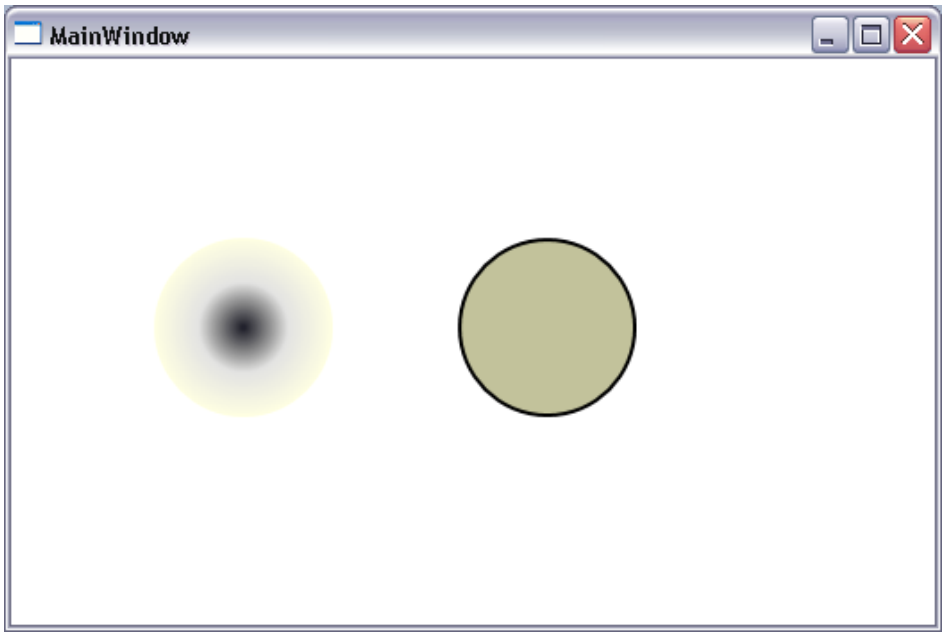


Example Gradient Brushes



```
<Window x:Class="RenderingEksempel.ShapesEks"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ShapesEks" Height="300" Width="300">
    <StackPanel>
        <Ellipse Name="A"  Height="150" Width="200">
            <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">
                    <GradientStop Color="Azure" Offset="0.0"/>
                    <GradientStop Color="Yellow" Offset="0.5"/>
                    <GradientStop Color="Blue" Offset="0.75"/>
                </LinearGradientBrush>
            </Ellipse.Fill>
        </Ellipse>
        <Polyline Stroke="Green" StrokeThickness="10" Points="40,10 40,40 60,60
                  100,40
                  "></Polyline>
        </StackPanel>
    </Window>
```

Configuring brushes in Code: Example



- RadialGradientBrush

```
…
        private void MenuItem_Click(object sender, RoutedEventArgs e)
         {

             shape[0] = new Ellipse();
             shape[0].Height = 100;
             shape[0].Width = 100;

             SolidColorBrush mySolidColorBrush = new SolidColorBrush();

             mySolidColorBrush.Color = Color.FromArgb(100, 100, 100, 0);
             shape[0].Fill = mySolidColorBrush;
             shape[0].StrokeThickness = 2;
             shape[0].Stroke = Brushes.Black;


             Canvas.SetLeft(shape[0], 250);
             Canvas.SetTop(shape[0], 100);

             myCanvas.Children.Add(shape[0]);

             shape[1] = new Ellipse();
             shape[1].Height = 100;
             shape[1].Width = 100;

             RadialGradientBrush rgb = new RadialGradientBrush();

             rgb.GradientStops.Add(new
GradientStop((Color)ColorConverter.ConvertFromString("#2020201b"), 0.5));
             rgb.GradientStops.Add(new
GradientStop((Color)ColorConverter.ConvertFromString("#FF20202b"), 0.015));
             rgb.GradientStops.Add(new
GradientStop((Color)ColorConverter.ConvertFromString("#20FFFF1b"), 1.0));
             shape[1].Fill = rgb;


             Canvas.SetLeft(shape[1], 80);
             Canvas.SetTop(shape[1], 100);


             myCanvas.Children.Add(shape[1]);



             this.Content = myCanvas;







         }


…
```

Graphical transformations

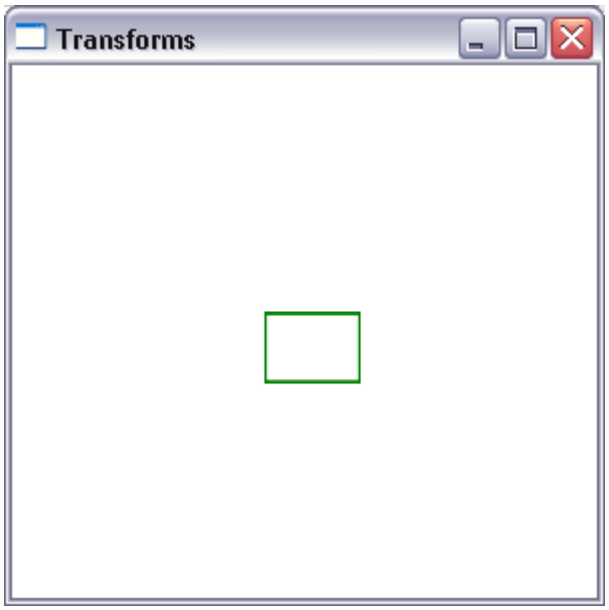
Table 29-5. Key Descendants of the System.Windows.Media.Transform Type

Type	Meaning in Life
MatrixTransform	Creates an arbitrary matrix transformation that is used to manipulate objects or coordinate systems in a 2D plane
RotateTransform	Rotates an object clockwise about a specified point in a 2D (x, y) coordinate system
ScaleTransform	Scales an object in the 2D (x, y) coordinate system
SkewTransform	Skews an object in the 2D (x, y) coordinate system
TranslateTransform	Translates (moves) an object in the 2-D (x-y) coordinate system
TransformGroup	Represents a composite Transform composed of other Transform objects

Notice:

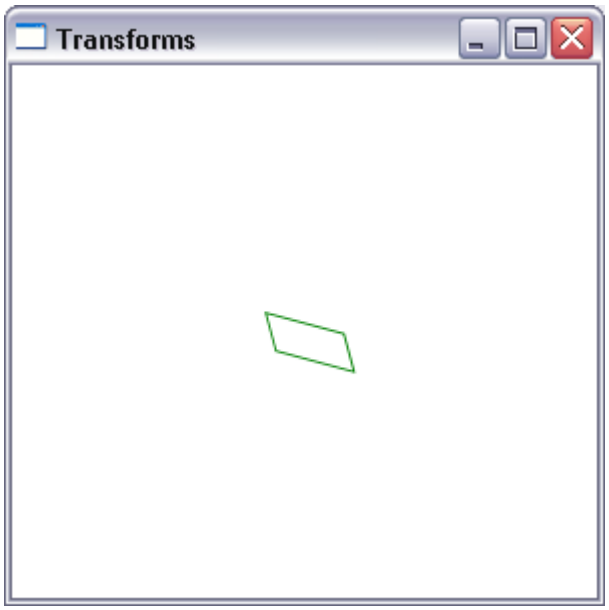
■ **Note** While transformation objects can be used anywhere, you will find them most useful when working with WPF animations and custom control templates. As you will see later in the text, you can use WPF animations to incorporate visual cues to the end user for a custom control.

- Scaling




```
<Window x:Class="RenderingEksempel.Transforms"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Transforms" Height="300" Width="300">
    <Grid>
          <Rectangle Height="20" Width="40" Stroke="Green" Margin="20,40"
ToolTip="En Firkant">
        <Rectangle.RenderTransform>
            <ScaleTransform ScaleX="1.2" ScaleY="1.8">

            </ScaleTransform>

        </Rectangle.RenderTransform>
        </Rectangle>


    </Grid>
</Window>
```

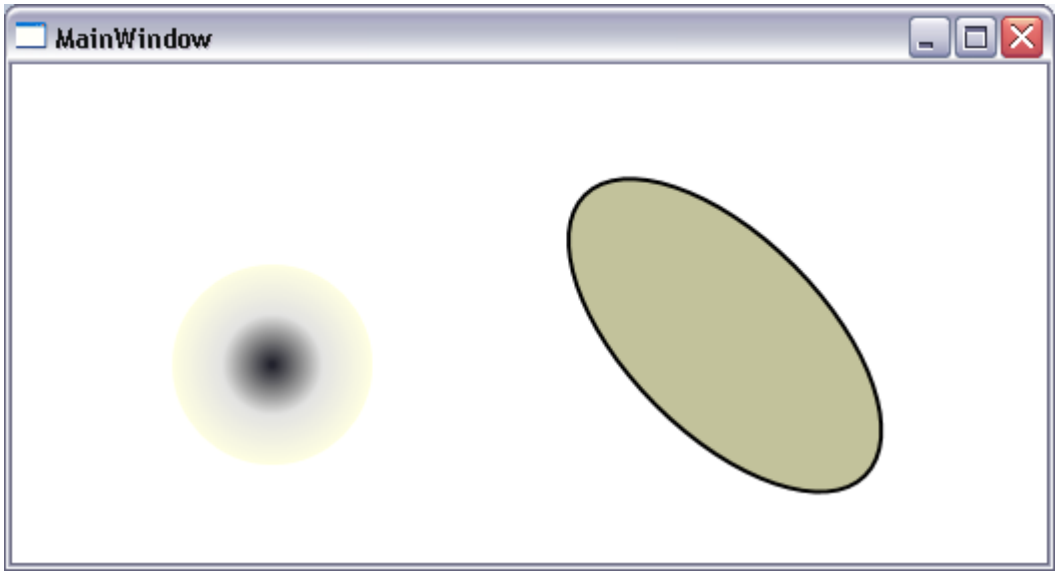
- Skew




```
<Window x:Class="RenderingEksempel.Transforms"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Transforms" Height="300" Width="300">
    <Grid>
          <Rectangle Height="20" Width="40" Stroke="Green" Margin="20,40"
ToolTip="En Firkant">
        <Rectangle.RenderTransform>
                <SkewTransform AngleX="15" AngleY="15">

                </SkewTransform>

        </Rectangle.RenderTransform>
        </Rectangle>


    </Grid>
</Window>
```

- Example using C# code:

```
...

            RotateTransform rt = new RotateTransform(-45);

             shape[0].RenderTransform = rt;


...
```



- - 45 degrees

Transforming canvas data

Transforming Your Canvas Data

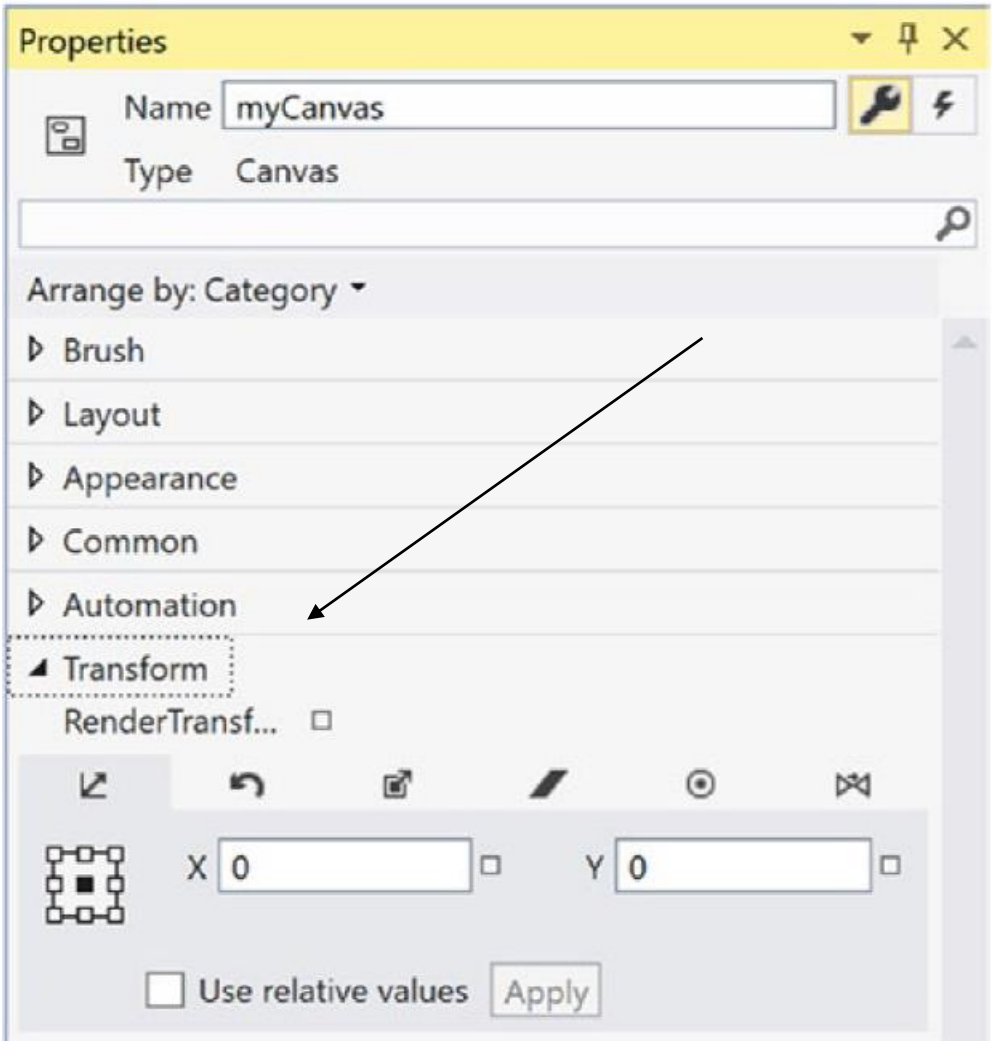
Now, let's incorporate some transformational logic into your RenderingWithShapes example. In addition to applying a transformation object to a single item (e.g., `Rectangle`, `TextBox`, etc.), you can also apply transformation objects to a layout manager in order to transform all of the internal data. You could, for example, render the entire `DockPanel` of the main window at an angle.

```
<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
...
```

Visual Studio transform editor

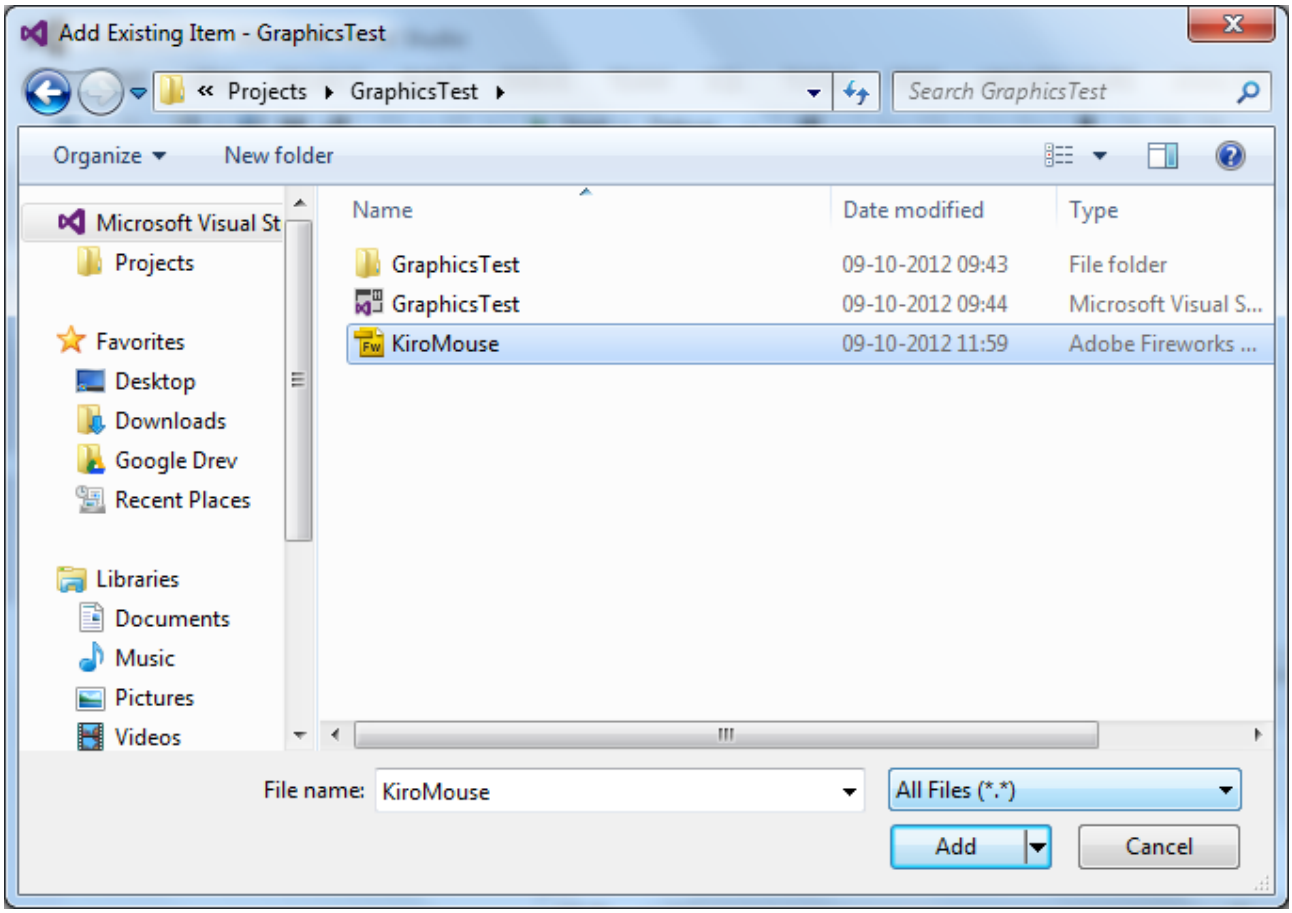
Working with the Visual Studio Transform Editor

In the previous example, you applied various transformations by manually entering markup and authoring some C# code. While this is certainly useful, you will be happy to know that the latest version of Visual Studio ships with an integrated transformation editor. Recall that any UI element can be the recipient of transformational services, including a layout system containing various UI elements. To illustrate the use of Visual Studio's transform editor, create a new WPF application named FunWithTransforms.



Images – ImageBrush

- Remember:



- Now the resource is known

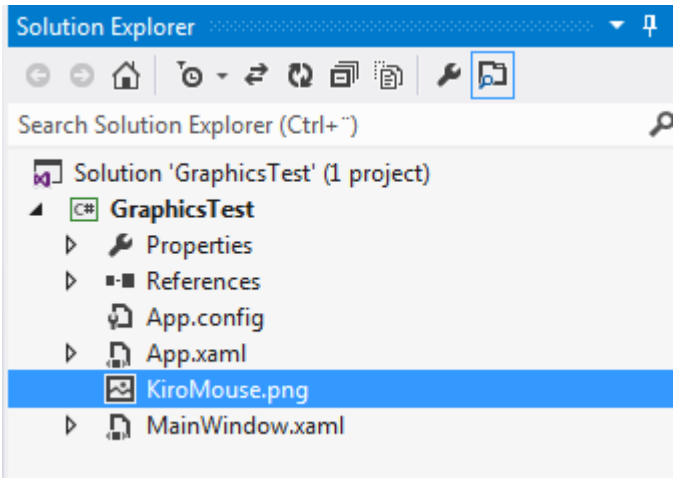
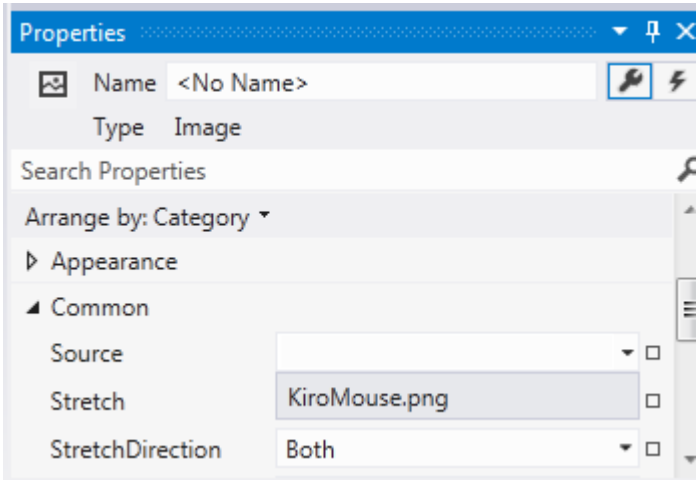

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'GraphicsTest' (1 project)
GraphicsTest
Properties
References
App.config
App.xaml
KiroMouse.png
MainWindow.xaml

Image:

- Add <Image> to Canvas
- Use Properties for <Image>
- Find Source (Choose “arrow down”)
- Find .png (or other format)



XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:Properties="clr-namespace:GraphicsTest.Properties"
x:Class="GraphicsTest.MainWindow"
    Title="WPF2D" Height="300" Width="300">
    <Canvas x:Name="canvasDrawingArea" Width="300" >
        <Image Height="100" Canvas.Left="95" Canvas.Top="77" Width="100"
Source="KiroMouse.png">

        </Image>
    </Canvas>
</Window>
```

Rendering graphics using Drawings and Geometry

- WPF Drawing types

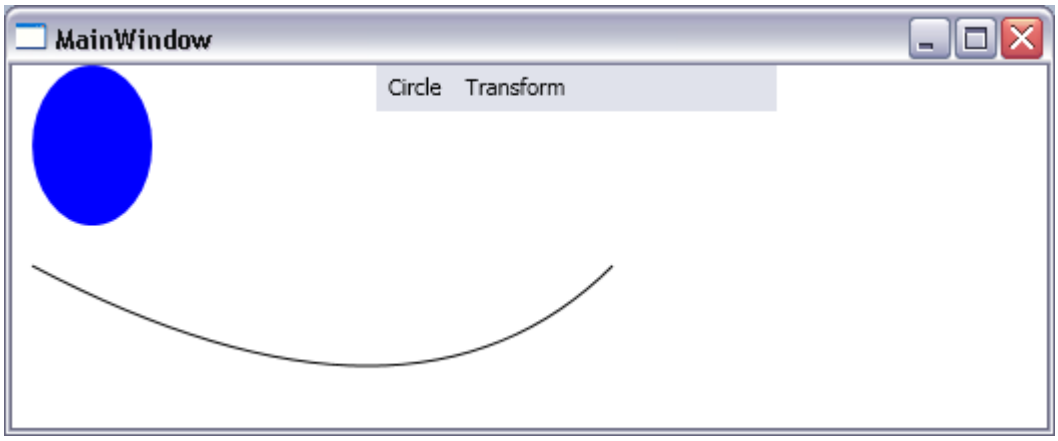
Table 29-7. *WPF Drawing-Derived Types*

Type	Meaning in Life
DrawingGroup	Used to combine a collection of separate Drawing-derived objects into a single composite rendering.
GeometryDrawing	Used to render 2D shapes in a very lightweight manner.
GlyphRunDrawing	Used to render textual data using WPF graphical rendering services.
ImageDrawing	Used to render an image file, or geometry set, into a bounding rectangle.
VideoDrawing	Used to play an audio file or video file. This type can only be fully exploited using procedural code. If you would like to play videos via XAML, the MediaPlayer type is a better choice.

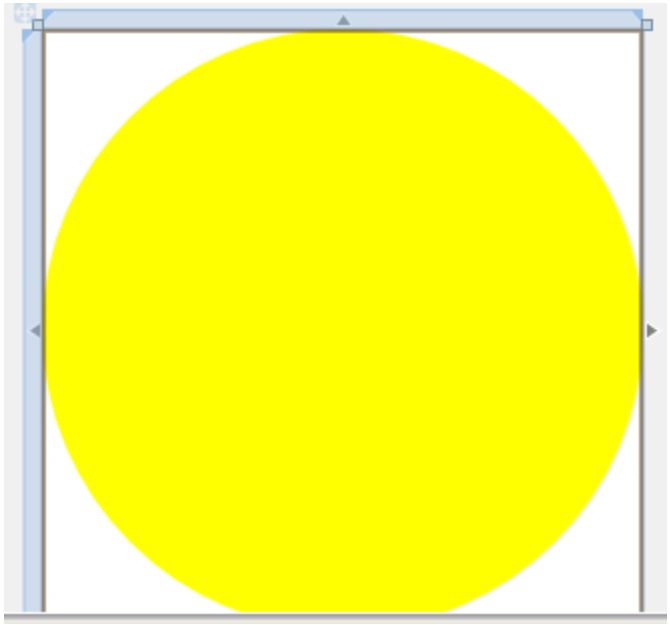
- Advantage: “lightweight”: is **not using** heavy inheritance

- Uses XAML

```
<Window x:Class="RenderingsEksempler.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Menu Height="23" HorizontalAlignment="Left" Margin="182,0,0,0" Name="menu1"
VerticalAlignment="Top" Width="200">
            <MenuItem Header="Circle" Click="MenuItem_Click">
                <MenuItem Header="Circle" />

            </MenuItem>

            <MenuItem Header="Transform" Click="MenuItem_Click_1">
                <MenuItem Header="Transform" />
            </MenuItem>

        </Menu>

        <Path Stroke="Black" StrokeThickness="1">
            <Path.Data>
                  <PathGeometry>
                    <PathGeometry.Figures>
                        <PathFigureCollection>

                            <PathFigure StartPoint="10,100">
                                <PathFigure.Segments>
                                    <PathSegmentCollection>
                                        <QuadraticBezierSegment Point1="200,200"
                                                                Point2="300,100" />

                                    </PathSegmentCollection>
                                </PathFigure.Segments>
                            </PathFigure>
                        </PathFigureCollection>
                    </PathGeometry.Figures>
                </PathGeometry>
           </Path.Data>
        </Path>


        <Path Fill="Blue">
            <Path.Data>
                <EllipseGeometry Center="40,40" RadiusX="30" RadiusY="40"></EllipseGeometry>
            </Path.Data>
        </Path>

    </Grid>
</Window>
```



DrawingBrush



```
<Page x:Class="RenderingsEksempler.DrawBrush"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
            Title="DrawBrush">

    <Grid>
        <Page>
            <Page.Background>
                <DrawingBrush>
                    <DrawingBrush.Drawing>
```

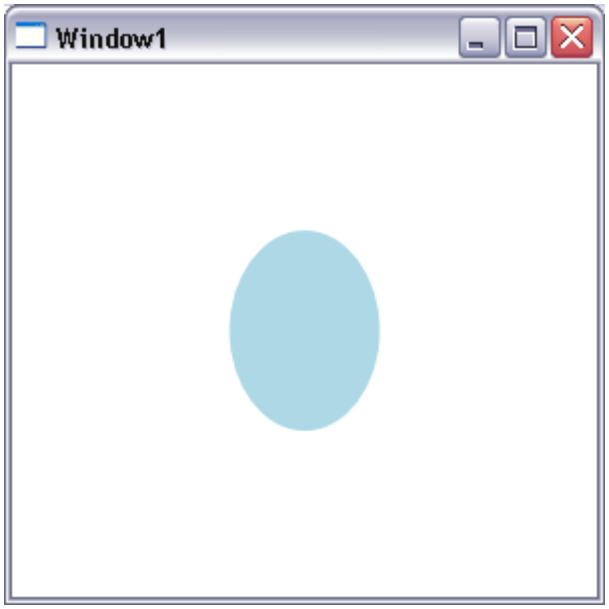
```
                        <GeometryDrawing>
                            <GeometryDrawing.Geometry>
                                <GeometryGroup>
                                    <EllipseGeometry Center="60,60" RadiusX="80"
RadiusY="30">

                                    </EllipseGeometry>
                                </GeometryGroup>
                            </GeometryDrawing.Geometry>
                                <GeometryDrawing.Brush>
                                <SolidColorBrush Color="Yellow">

                                </SolidColorBrush>
                            </GeometryDrawing.Brush>
                        </GeometryDrawing>
                    </DrawingBrush.Drawing>
                </DrawingBrush>

            </Page.Background>
        </Page>
        <Path Fill="Green">
            <Path.Data>
                <GeometryGroup>

                </GeometryGroup>

            </Path.Data>

        </Path>
    </Grid>
</Page>
```

DrawingImage

- The <DrawingImage> type allows to draw geometrical figures in <Image>

```
<Window x:Class="RenderingsEksempler.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
    <Grid>
        <Image Height="100" Width="100">
            <Image.Source>
                <DrawingImage>
                    <DrawingImage.Drawing>
                        <GeometryDrawing>
                            <GeometryDrawing.Geometry>
                                <GeometryGroup>
```

```
                                    <EllipseGeometry Center="80,80" RadiusX="60"
RadiusY="80">

                                    </EllipseGeometry>
                                </GeometryGroup>
                            </GeometryDrawing.Geometry>
                            <GeometryDrawing.Brush>
                                <SolidColorBrush Color="LightBlue"></SolidColorBrush>
                            </GeometryDrawing.Brush>
                        </GeometryDrawing>
                    </DrawingImage.Drawing>
                </DrawingImage>
            </Image.Source>

        </Image>

    </Grid>
</Window>
```



Complex Vector Graphics – working with vector images

Working with Vector Images

As you might agree, it would be quite challenging for a graphic artist to create a complex vector-based image using the tools and techniques provided by Visual Studio. Graphic artists have their own set of tools that can produce amazing vector graphics. Neither Visual Studio nor its companion Expression Blend for Visual Studio have that type of design power. Before you can import vector images into WPF application, they must be converted into Path expressions. At that point, you can program against the generated object model using Visual Studio.

Note You can find the image being used (`LaserSign.svg`) as well as the exported path (`LaserSign.xaml`) data in the `Chapter 26` folder of the download files. The image is originally from Wikipedia, located in this article: `https://en.wikipedia.org/wiki/Hazard_symbol`.

Convert!

Converting a Sample Vector Graphic File into XAML

Before you can import complex graphical data (such as vector graphics) into a WPF application, you need to convert the graphics into path data. As an example of how to do this, start with a sample `.svg` image file, such as the laser sign referenced in the preceding note. Then download and install an open source tool called Inkscape (located at `www.inkscape.org`). Using Inkscape, open the `LaserSign.svg` file from the chapter download. You might be prompted to upgrade the format. Fill in the selections as shown in Figure 26-12.

- See page 1092 –

Visual Layer – DrawingVisual

- Visual Base class

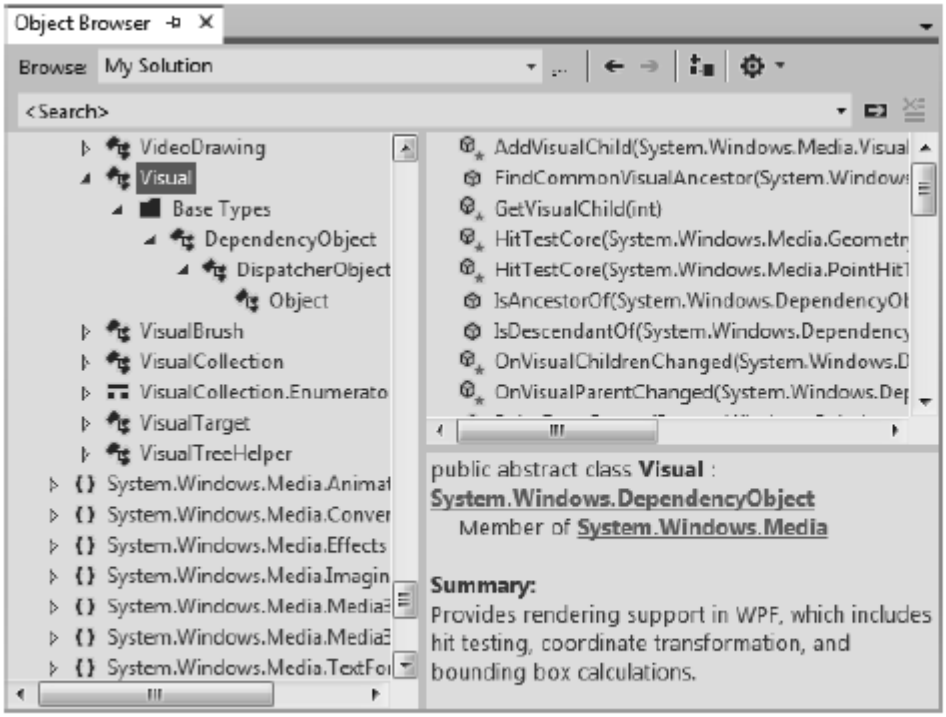
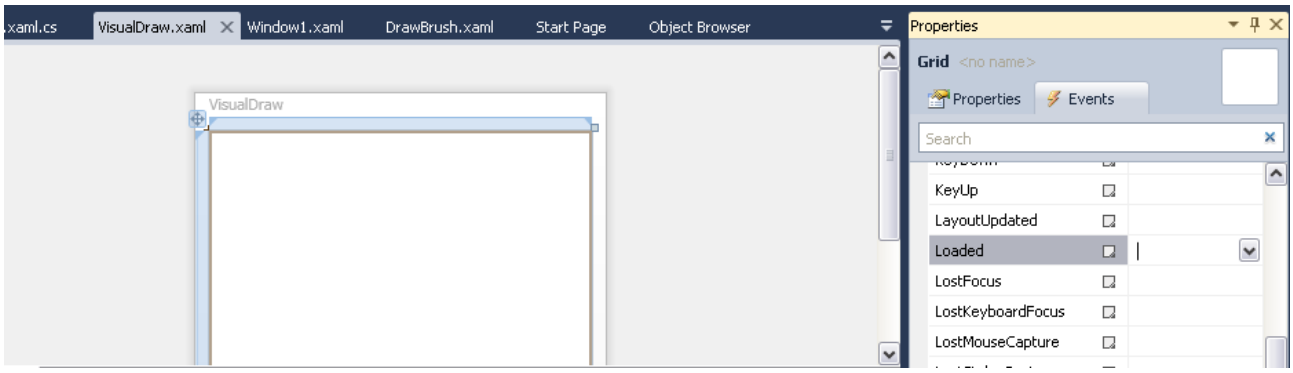


Figure 29-20. *The Visual type provides basic hit-testing, coordinate transformation, and bounding box calculations*

- Example from earlier showed the usage of DrawingVisual
- DrawingVisual is low-level graphical layer
- But also useful to create pictures

- In order to draw this way:
  - Get access to a DrawingContext object
  - Use DrawingContext for drawing

1. Find the Event Loaded (in the Window)




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace RenderingsEksempler
{
    /// <summary>
    /// Interaction logic for VisualDraw.xaml
    /// </summary>
    public partial class VisualDraw : Window
    {
        public VisualDraw()
        {
            InitializeComponent();
        }

        private void Grid_Loaded(object sender, RoutedEventArgs e)
        {
            FormattedText ft = new FormattedText("DrawingVisual", new
System.Globalization.CultureInfo("en-us"), FlowDirection.LeftToRight, new
Typeface(this.FontFamily, FontStyles.Italic, FontWeights.DemiBold,
FontStretches.UltraExpanded), 12.0, Brushes.Blue);

            DrawingVisual dv = new DrawingVisual();
            using (DrawingContext dc = dv.RenderOpen())
            {

                dc.DrawRoundedRectangle(Brushes.Yellow, new Pen(Brushes.Black, 5), new
Rect(5, 5, 100, 50), 80, 20);

                dc.DrawText(ft, new Point(11, 20));
            }
          RenderTargetBitmap rtb = new RenderTargetBitmap(100, 80, 80, 90,
PixelFormats.Pbgra32);

            rtb.Render(dv);
```

```
            MitBillede.Source = rtb;


        }
    }
}
```

- Create a text
- Create a DrawingVisual object
- Create a DrawingContext object (should be used in a **using** construction)
- Draws a Rectangle
- Draws a text
- Render..
- Attach a picture to <Image> tag in the XAML

Response to GUI Hits

- It is possible to register mouse-hit on UI elements
- Start to create the event: MouseDown
- Create a delegate that points to a helper function
- See code page 1099 -

WPF Resources, Animations, styles and templates

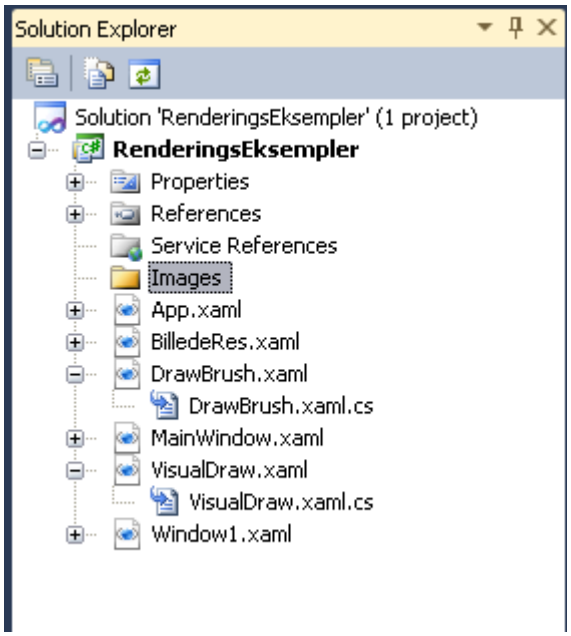
- WPF supports two types of resources
  - Binary
    - Image
    - Sound
  - Object
    - named .NET objects

Binary: Image

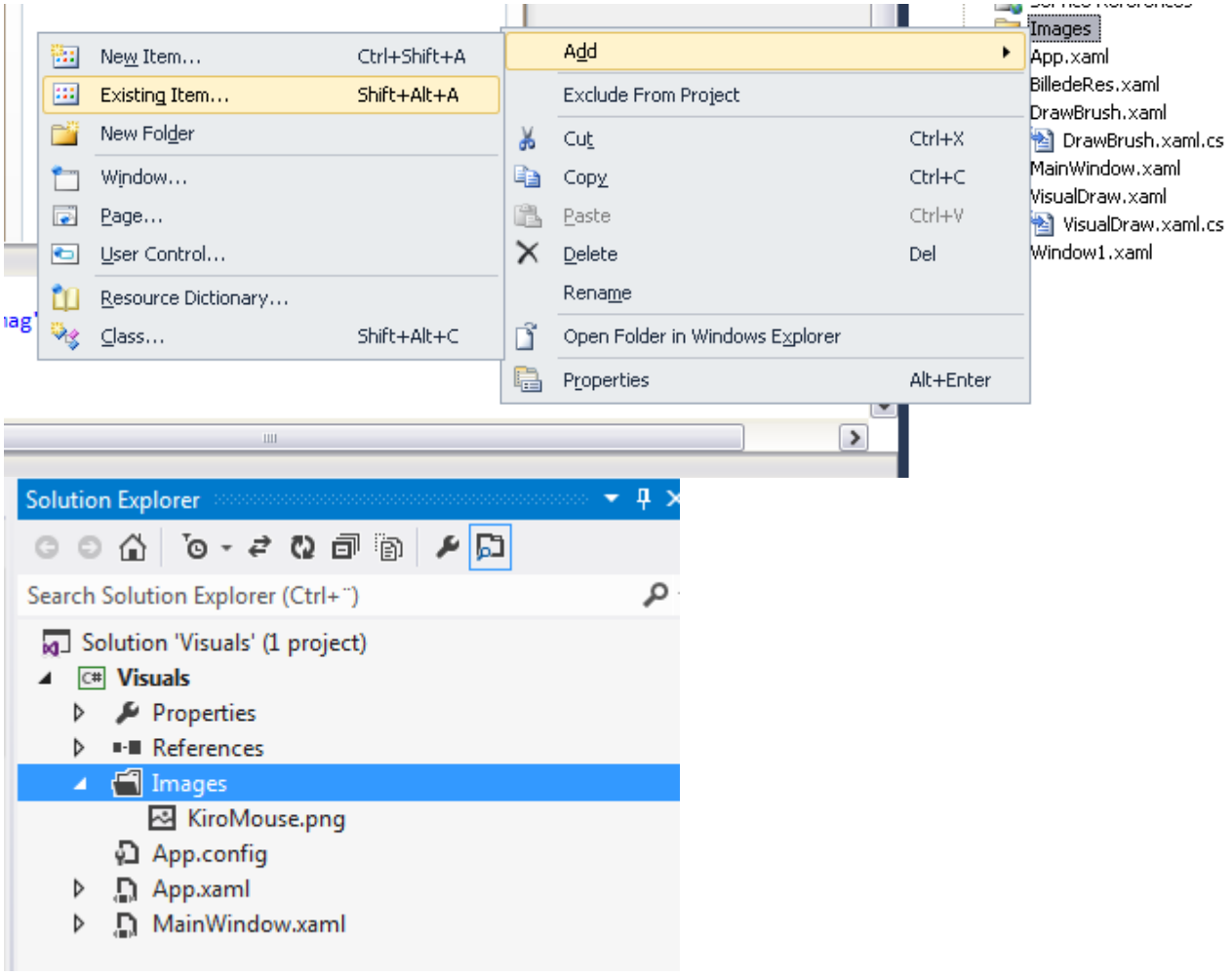
```
<Window x:Class="RenderingsEksempler.BilledeRes"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BilledeRes" Height="300" Width="300">
    <Grid>
        <Image Name="imag" Height="100" Width="100"></Image>
    </Grid>
</Window>
```

- Roadmap:

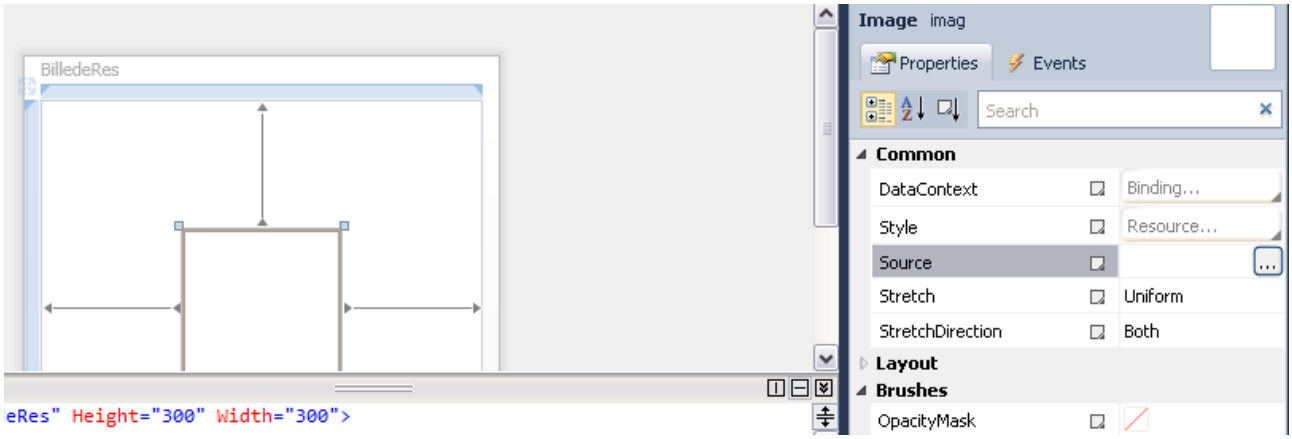
- Add a folder (***Images*** is here chosen as a name)



- On Images: (right-click) Add | Existing Item



- Choose Properties on <Image>. Select **Source**



- Image is ”Well-known”

- Finish!

Programmable load of images

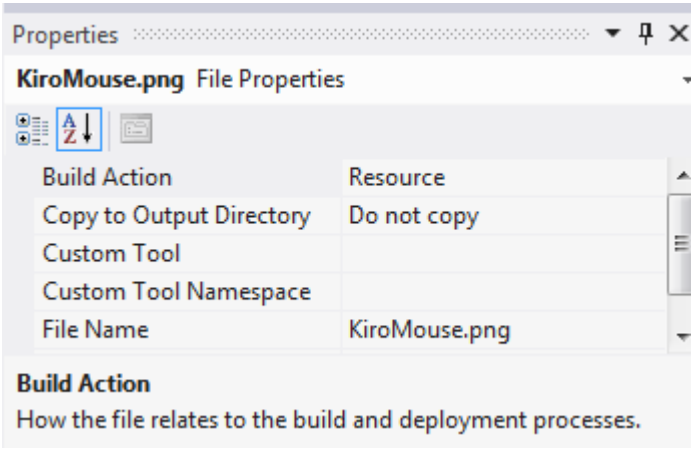
- A folder called ”Images” **is placed in the folder with the .exe file**
- Use **Environment.CurrentDirectory**
- Place an <Image> tag to contain the picture (Here called **imag2**)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace RenderingsEksempler
{
    /// <summary>
    /// Interaction logic for BilledeRes.xaml
    /// </summary>
    public partial class BilledeRes : Window
    {

        public BilledeRes()
        {
            InitializeComponent();

        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            String path = @Environment.CurrentDirectory;

            BitmapImage bi = new BitmapImage(new Uri(string.Format(@"{0}\Images\DTU-
                                                                    logo.gif", path)));
            imag2.Source = bi;
        }
    }
}
```

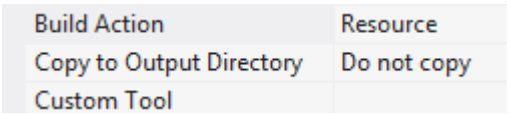
- Full path

```
...
  <Grid>
        <Image Name="imag" Width="100"
Source="/RenderingsEksempler;component/Images/3apps.jpg" Margin="91,18,91,87"></Image>
        <Image Name="imag2" Height="100" Width="100"></Image>
  </Grid>
...
```

Embedded application resources



- Choose the .png in Solution Explorer. Right click!
- Choose Resource
- Choose also "Do not copy" in **Copy to Output**



- Now the Images can be removed (subfolder where the application run)
- Change the code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace RenderingsEksempler
{
    /// <summary>
    /// Interaction logic for BilledeRes.xaml
    /// </summary>
    public partial class BilledeRes : Window
```

```
    {

        public BilledeRes()
        {
            InitializeComponent();

        }

      private void Window_Loaded(object sender, RoutedEventArgs e)
      {

        BitmapImage bi = new BitmapImage(new Uri(@"/Images/filename",UriKind.Relative));
        imag2.Source = bi;
      }
    }
}
```

```
// Extract from the assembly and then load images
_images.Add(new BitmapImage(new Uri(@"/Images/Deer.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"/Images/Dogs.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"/Images/Welcome.jpg", UriKind.Relative)));
```

In this case, you no longer need to determine the installation path and can simply list the resources by name, which takes into account the name of the original subdirectory. Also notice, when you create your `Uri` objects, you specify a `UriKind` value of `Relative`. At this point, your executable is a stand-alone entity that can be run from any location on the machine because all the compiled data is within the binary.

Object (logical) resources

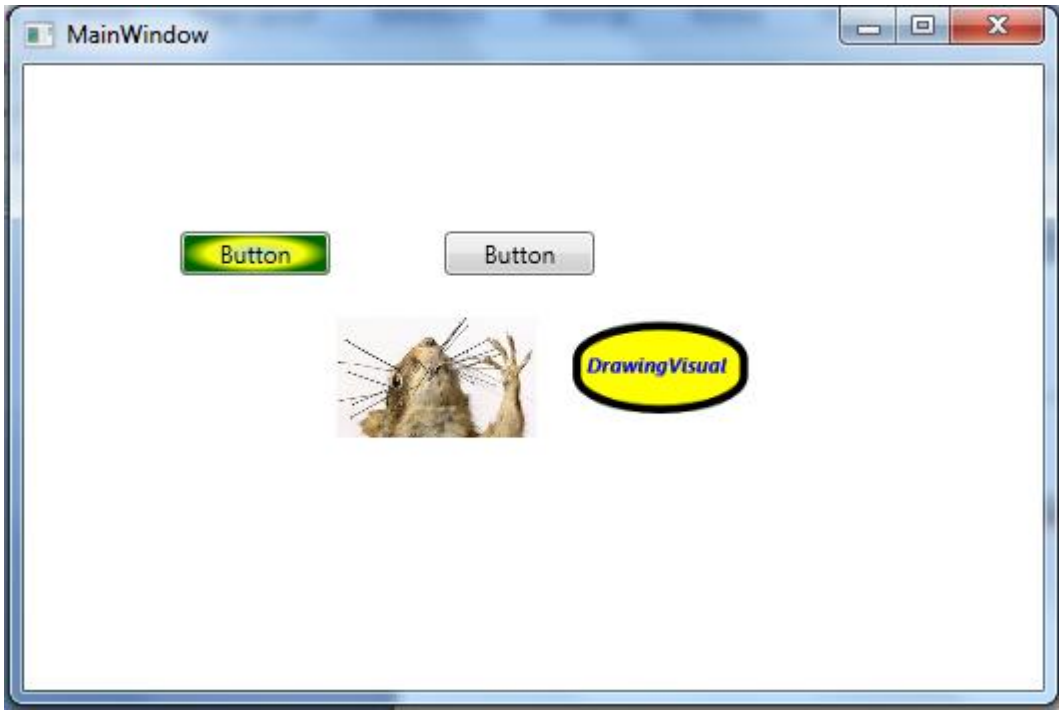
- Imagine that XAML should be “reused” of several components
- Awful to copy-paste
- Here object resources can be used
- Give the objects name
- Store them!

Example:

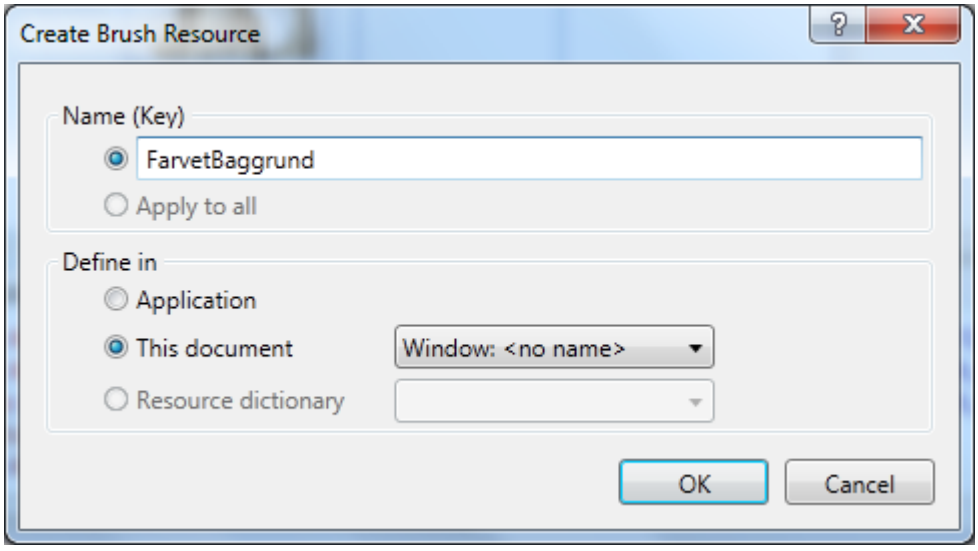
```
<Window x:Class="RenderingsEksempler.BilledeRes"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BilledeRes" Height="176" Width="304" Loaded="Window_Loaded">
    <StackPanel>
        <Image Name="imag" Width="100"
Source="/RenderingsEksempler;component/Images/3apps.jpg" Margin="91,18,91,87"></Image>
        <Image Name="imag2" Height="100" Width="100"></Image>
        <Button Content="Knap1" Height="23" HorizontalAlignment="Left" Name="button1"
VerticalAlignment="Top" Width="75" >
```

```
            <Button.Background>
                <RadialGradientBrush>
                    <GradientStop Color="LightBlue" Offset="0"></GradientStop>
                    <GradientStop Color="DarkGreen" Offset="1"></GradientStop>
                    <GradientStop Color="Yellow" Offset="0.669"></GradientStop>
                </RadialGradientBrush>


            </Button.Background>
        </Button>
        <Button Content="Knap2" Height="23" HorizontalAlignment="Left" Margin="0,0,0,87"
Name="button2" VerticalAlignment="Bottom" Width="75" />
    </StackPanel>
</Window>
```

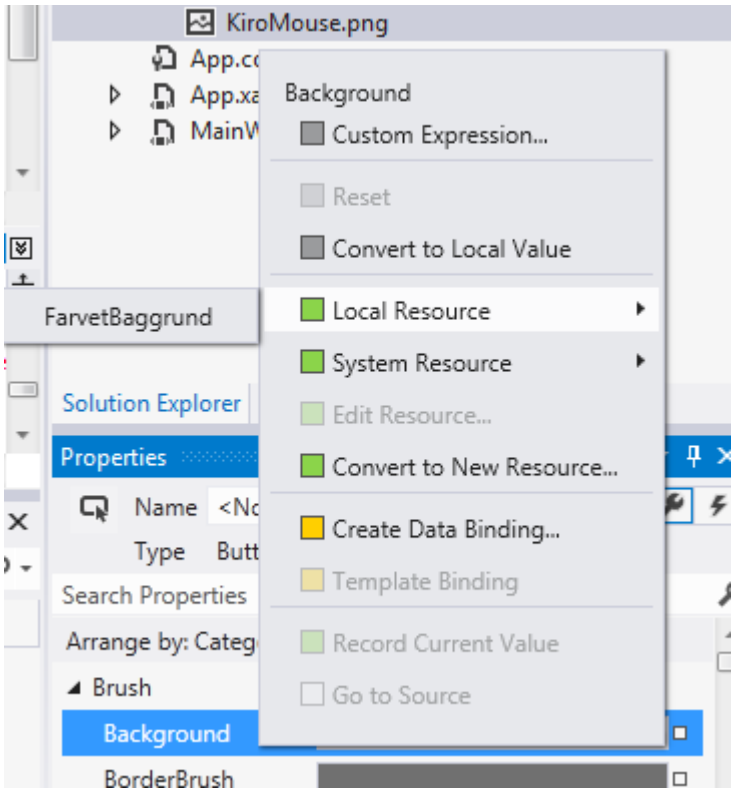
- One button (Knap1) has here a <Button.BackGround>
- **The other button (Knap2) has not**



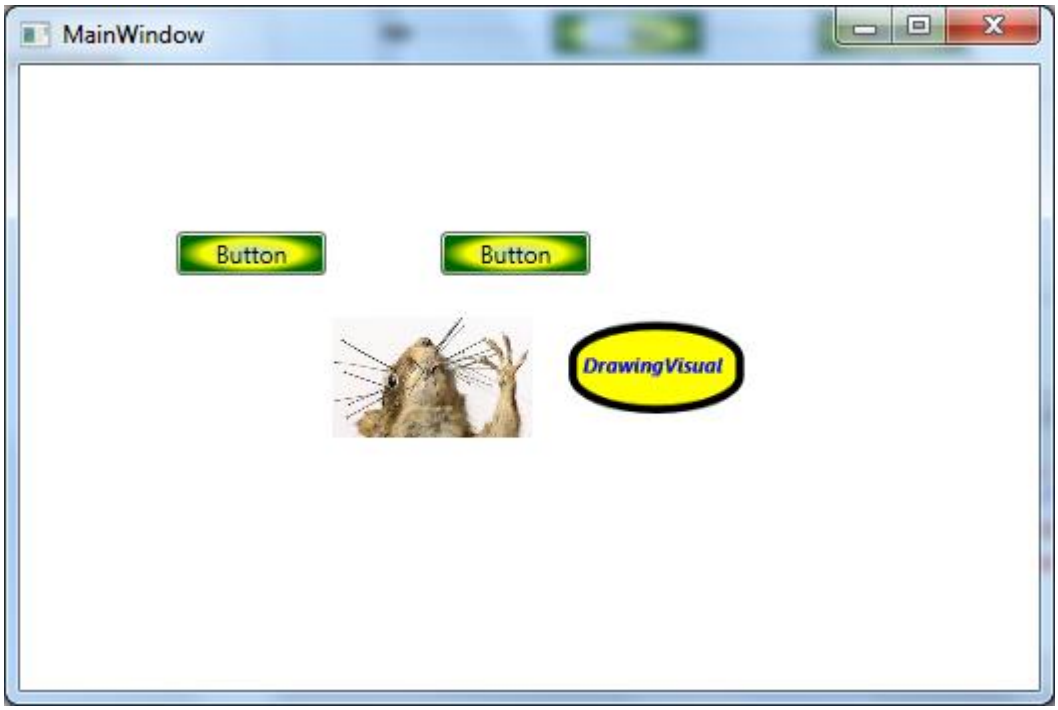
- Convert to new resource (knap1)



- On knap2



- Result



XAML:

```
<Window x:Class="Visuals.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <RadialGradientBrush x:Key="FarvetBaggrund">
            <GradientStop Color="LightBlue" Offset="0"/>
            <GradientStop Color="DarkGreen" Offset="1"/>
            <GradientStop Color="Yellow" Offset="0.669"/>
        </RadialGradientBrush>


    </Window.Resources>
    <Grid Loaded="Grid_Loaded_1">
        <Image x:Name="MitBillede" HorizontalAlignment="Left" Height="100"
Margin="272,126,0,0" VerticalAlignment="Top" Width="100"/>
        <Image x:Name="imag" HorizontalAlignment="Left" Height="62" Margin="156,126,0,0"
VerticalAlignment="Top" Width="100" Source="Images/KiroMouse.png"/>
        <Button x:Name="knap1" Content="Button" HorizontalAlignment="Left"
Margin="78,83,0,0" VerticalAlignment="Top" Width="75" Background="{DynamicResource
FarvetBaggrund}"/>
        <Button Content="Button" HorizontalAlignment="Left" Margin="210,83,0,0"
VerticalAlignment="Top" Width="75" Background="{DynamicResource FarvetBaggrund}"/>

    </Grid>
</Window>
```

- It is also possible to use {DynamicResource}

The {DynamicResource} Markup Extension

It is also possible for a property to use the DynamicResource markup extension. To see the difference, change the markup for the Cancel Button to the following:

```
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel"
                FontSize="20" Background="{DynamicResource myBrush}" Click="Cancel_OnClick"/>
```

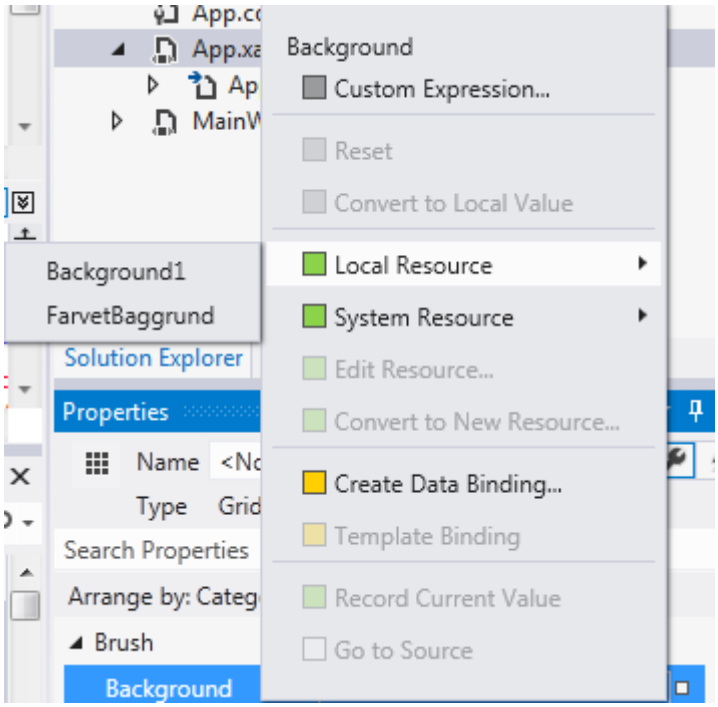
This time, when you click the Cancel Button, the background for the Cancel Button changes, but the background for the OK Button remains the same. This is because the {DynamicResource} markup extension is able to detect whether the underlying keyed object has been replaced with a new object. As you might guess, this requires some extra runtime infrastructure, so you should typically stick to using {StaticResource} unless you know you have an object resource that will be swapped with a different object at runtime and you want all items using that resource to be informed.

Application level resources

- Problem above:
  - Will only work inside **the same Window**

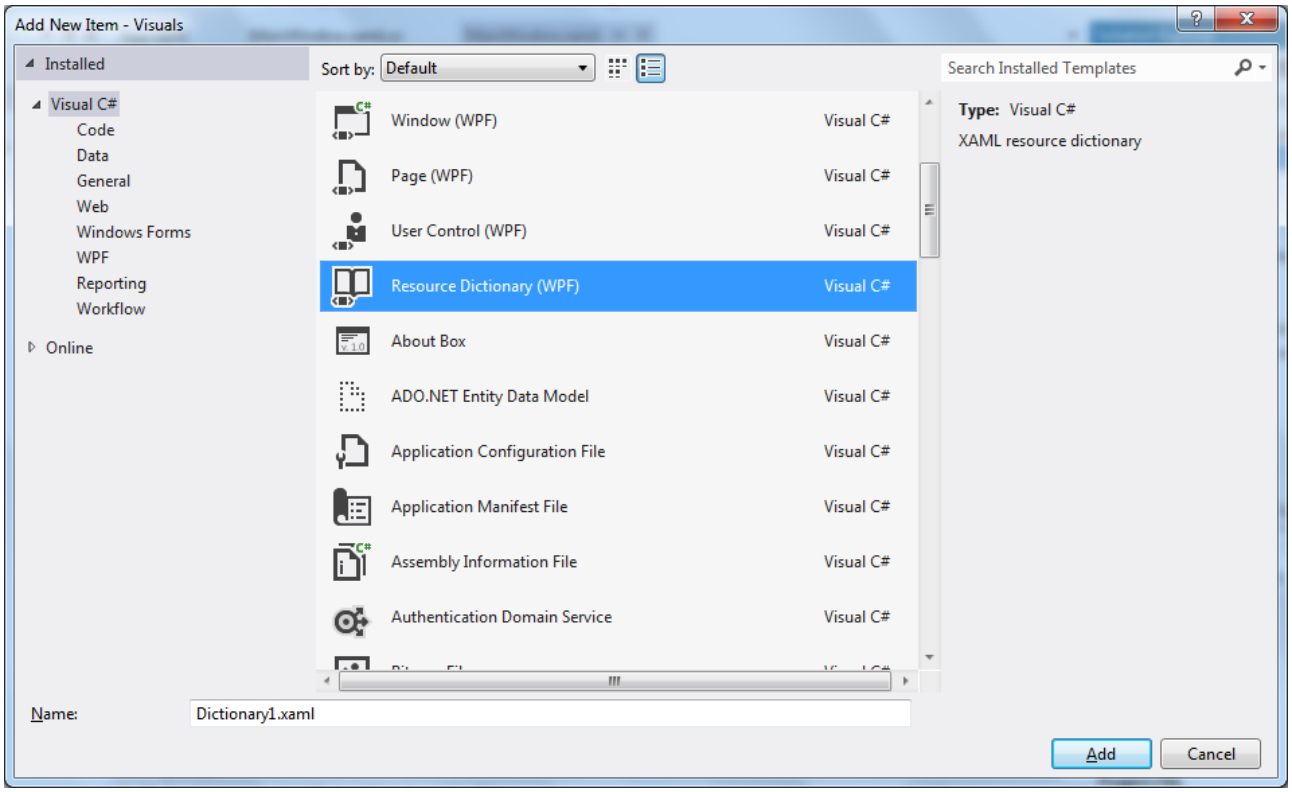
- Can be made **Application Specific**
- Placed in the App.xaml file

```
<Application x:Class="RenderingsEksempler.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <RadialGradientBrush x:Key="Background1">
            <GradientStop Color="LightBlue" Offset="0" />
            <GradientStop Color="DarkGreen" Offset="1" />
            <GradientStop Color="Yellow" Offset="0.669" />
        </RadialGradientBrush>

    </Application.Resources>
</Application>
```



Merged Resource Dictionary

- Is simply a .xaml file with a collection of object resources
- One single project can have several of those files
    - Notice: <ResourceDictionary> tag
- ALL resource dictionary files is added to a <ResourceDictionary> on application level




```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<RadialGradientBrush x:Key="Background1">
        <GradientStop Color="LightBlue" Offset="0" />
        <GradientStop Color="DarkGreen" Offset="1" />
        <GradientStop Color="Yellow" Offset="0.669" />
    </RadialGradientBrush>
</ResourceDictionary>
```

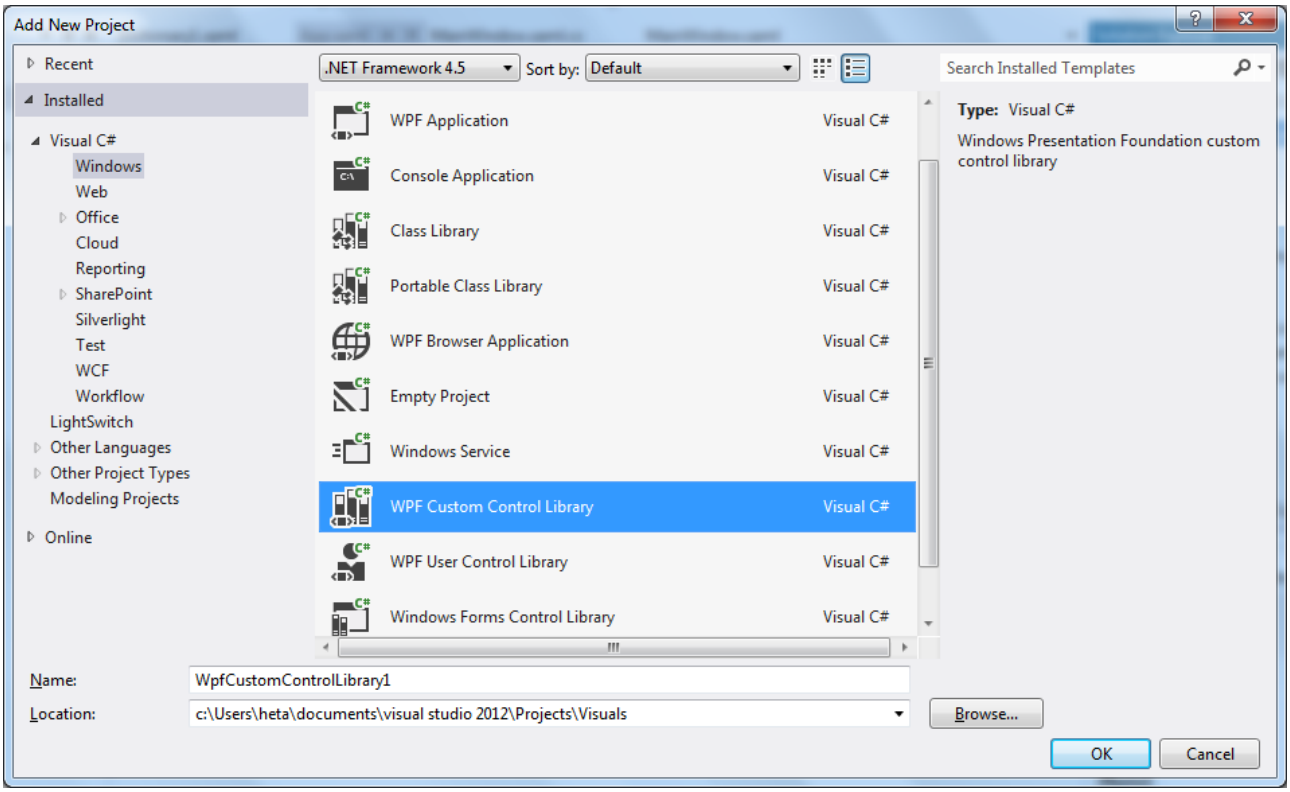
Dictionary1.xaml

```
<Application x:Class="Visuals.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
<ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source = "Dictionary1.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>


    </Application.Resources>
</Application>
```

App.xaml

An assembly only with resources

- Create a WPF user Control library



- Drag the .xaml file to this project
- Compile
- Create a reference
    - (Add Reference)

Compile your User Control Library project. Next, reference this library from the ObjectResourcesApp project using the Add Reference dialog box. Now, merge these binary resources into the application-level resource dictionary of the ObjectResourcesApp project. Doing so, however, requires some rather funky syntax, shown here:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <!-- The syntax is /NameOfAssembly;Component/NameOfXamlFileInAssembly.xaml -->
      <ResourceDictionary Source = "/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

First, be aware that this string is space-sensitive. If you have extra white space around your semicolon or forward slashes, you will generate errors. The first part of the string is the friendly name of the external library (no file extension). After the semicolon, type in the word Component followed by the name of the compiled binary resource, which will be identical to the original XAML resource dictionary.

Animations

- Animation is (for instance)
  - Rotating pictures, moving texts, video and much more
  - In general a way to make a better and more interesting UI

Animation class types

- More than 100 classes dedicated to the purpose __Animation

To and From properties

- To: Represent end value
- From: Represent start value

TimeLine class

- Common base class - TimeLine

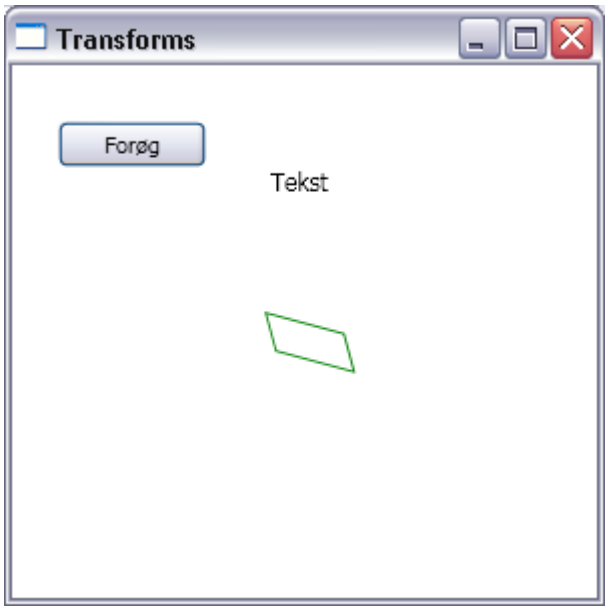
Table 30-1. *Key Members of the* Timeline *Base Class*

Properties	Meaning in Life
AccelerationRatio, DecelerationRatio, SpeedRatio	These properties can be used to control the overall pacing of the animation sequence.
AutoReverse	This property gets or sets a value that indicates whether the timeline plays in reverse after it completes a forward iteration (the default value is false).
BeginTime	This property gets or sets the time at which this timeline should begin. The default value is 0, which begins the animation immediately.
Duration	This property allows you to set a duration of time to play the timeline.
FillBehavior, RepeatBehavior	These properties are used to control what should happen once the timeline has completed (repeat the animation, do nothing, etc.).

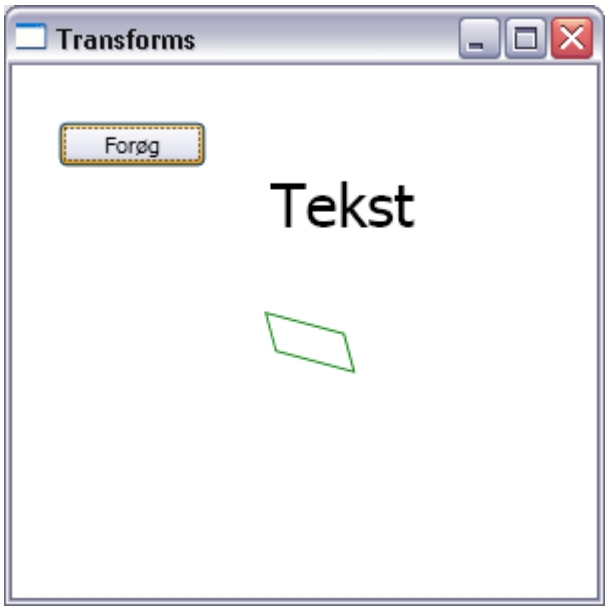
Animation example 1

- Font size goes slowly
  - from 12 to 30

START:



End:



C# Code

```
...
   private void button1_Click(object sender, RoutedEventArgs e)
        {
            DoubleAnimation da = new DoubleAnimation();
            da.From = 12;
            da.To = 30;
```

```
            l1.BeginAnimation(Label.FontSizeProperty, da);
        }
...
```

- It is a double that is incremented

Animation example 2

- Circle and rectangle moves from 0 to 45 degree and back again 5 times

```
<Window x:Class="AnimationsEksempel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Animation" Height="23" HorizontalAlignment="Left"
          Margin="201,35,0,0" Name="Animation" VerticalAlignment="Top" Width="75"
                                                    Click="Animation_Click" />
        <Rectangle Height="75" HorizontalAlignment="Left" Margin="291,136,0,0"
                   Name="rectangle1" Stroke="Black" VerticalAlignment="Top" Width="146" />
        <Ellipse Height="75" HorizontalAlignment="Left" Margin="87,136,0,0" Name="ellipse1"
                                    Stroke="Black" VerticalAlignment="Top" Width="153" />
    </Grid>
</Window>
```

.xaml

C# Code

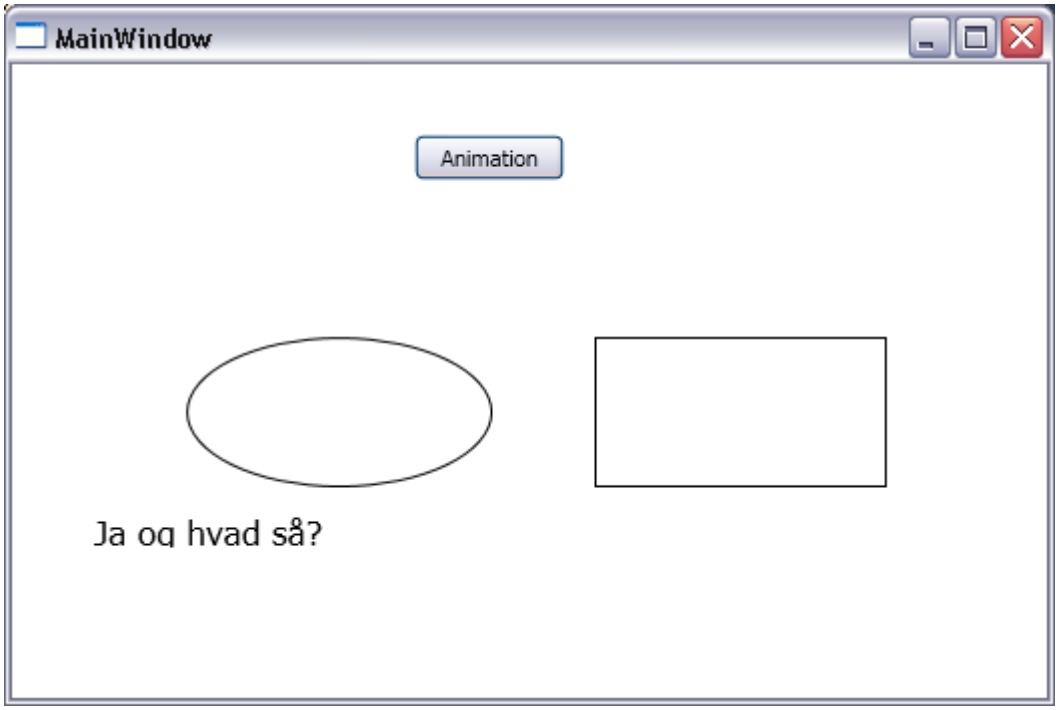
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
```

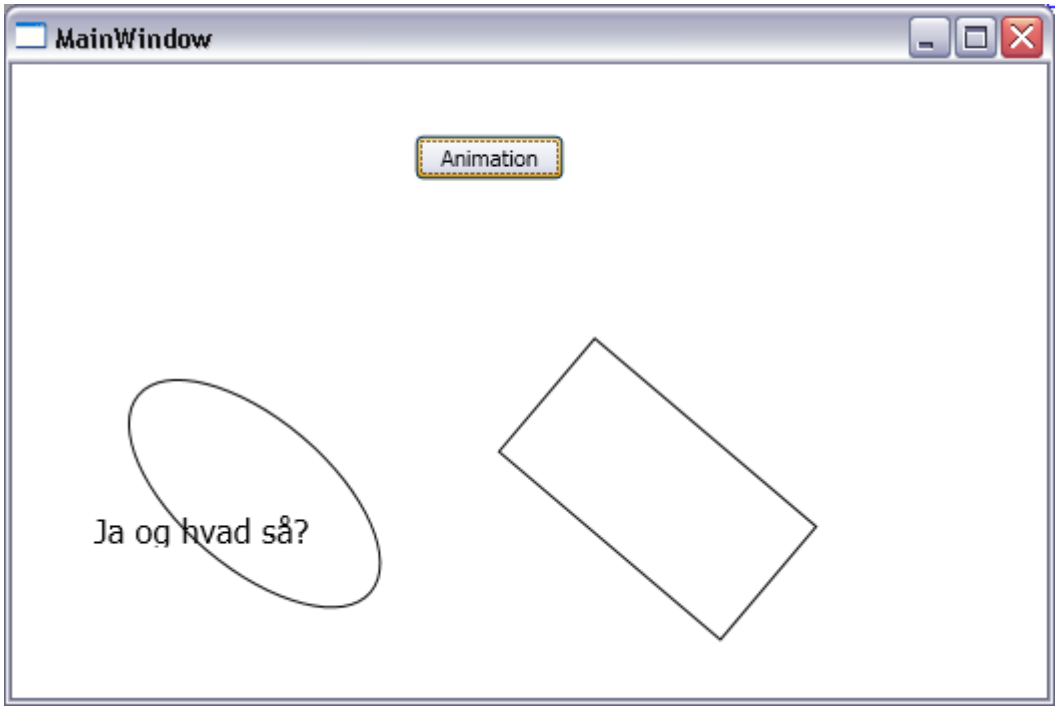
```
using System.Windows.Media.Animation;

namespace AnimationsEksempel
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Animation_Click(object sender, RoutedEventArgs e)
        {
            DoubleAnimation da = new DoubleAnimation();

            da.To = 45;
            da.From = 0;


            RotateTransform rt1 = new RotateTransform();
            rectangle1.RenderTransform = rt1;

            RotateTransform rt2 = new RotateTransform();
            ellipse1.RenderTransform = rt2;
            da.AutoReverse = true;                      // Reverse
            da.RepeatBehavior = new RepeatBehavior(5); // 5 times

            rt1.BeginAnimation(RotateTransform.AngleProperty, da);
            rt2.BeginAnimation(RotateTransform.AngleProperty, da);
        }
    }
}
```

Storyboards and event triggers

- Here is pure XAML used

```
...
<Label Content="Ja og hvad så?" Height="28" HorizontalAlignment="Left" Margin="36,218,0,0"
Name="label1" VerticalAlignment="Top" >
            <Label.Triggers>
                <EventTrigger RoutedEvent="Label.Loaded">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard TargetProperty="FontSize">
                                <DoubleAnimation From="12" To="50" Duration="0:0:4"
                                                                    RepeatBehavior="Forever">

                                </DoubleAnimation>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Label.Triggers>

        </Label>


...
```

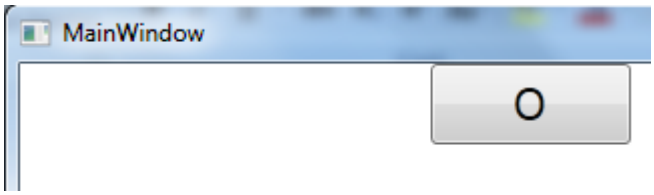


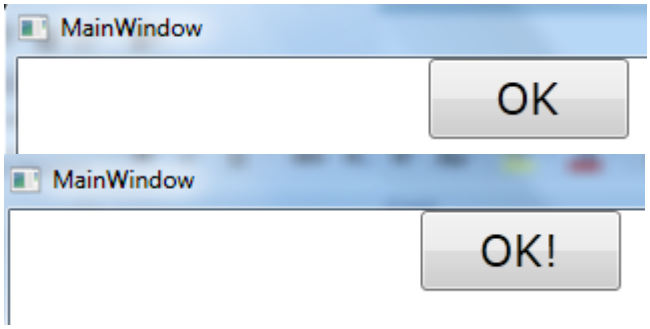


Discrete KeyFrames

```
<Window x:Class="Anima.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

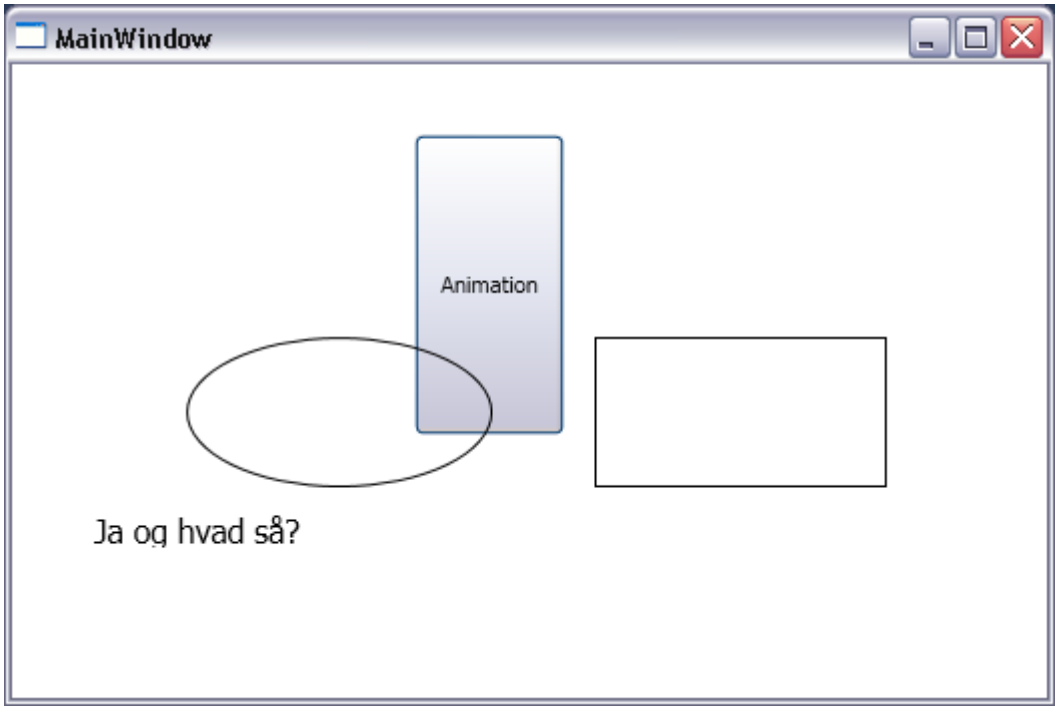
```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <Button Name="myButton" Height="40"
                        FontSize="16pt" FontFamily="Verdana" Width = "100">
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.Loaded">
                    <BeginStoryboard>
                        <Storyboard>
                            <StringAnimationUsingKeyFrames RepeatBehavior = "Forever"
                                                           Storyboard.TargetName="myButton"
                                                           Storyboard.TargetProperty="Content"
                                                           Duration="0:0:3">
                                <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                                <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                                <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                                <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
                            </StringAnimationUsingKeyFrames>
                            </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Button.Triggers>
</Button>
</StackPanel>
</Window>
```





WPF Styles

- A style is general
  - Use App.xaml to set up a style



- Button Height is now controlled from App.xaml
- App.xaml contains a <Style> element

```
<Application x:Class="AnimationsEksempel.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <Style x:Key="MyAppStyle">
            <Setter Property="Control.Height" Value="150"></Setter>
        </Style>
    </Application.Resources>
</Application>
```

- Now the style can be attached to a Button (just **REMOVE** the Height property on the Button) and **refer** to MyAppStyle

```
<Button Content="Animation" HorizontalAlignment="Left" Margin="201,35,0,0" Name="Animation"
VerticalAlignment="Top" Width="75" Click="Animation_Click" Style="{StaticResource
MyAppStyle}" />
```

”Override”

- If the property in Control (Button) is placed the resource is override

```
<Button Content="Animation" HorizontalAlignment="Left" Margin="201,35,0,0" Name="Animation"
VerticalAlignment="Top" Width="75" Click="Animation_Click" Height="50"
Style="{StaticResource MyAppStyle}" />
```

- Or by using x:Null

```
<!-- The default style for all text boxes. -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
</Style>
```

You can now define any number of `TextBox` controls, and they will automatically get the defined look. If a given `TextBox` does not want this default look and feel, it can opt out by setting the `Style` property to `{x:Null}`. For example, `txtTest` will get the default unnamed style, while `txtTest2` is doing things its own way.

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
       BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

Automatically adding style to a Control of a type (Button)

```
<Application x:Class="AnimationsEksempel.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <Style x:Key="MyAppStyle" TargetType="Button">
            <Setter Property="Control.Height" Value="150"></Setter>
        </Style>
    </Application.Resources>
</Application>
```

- Use: TargetType =”Button”
- Will now be used by all Buttons that binds with `Style="{StaticResource MyAppStyle}"`

BasedOn (Extension of a Style)

```
<!-- This style is based on BigGreenButton. -->
<Style x:Key ="TiltButton" TargetType="Button" BasedOn = "{StaticResource
BigGreenButton}">
<Setter Property = "Foreground" Value = "White"/>
<Setter Property = "RenderTransform">
<Setter.Value>
<RotateTransform Angle = "20"/>
</Setter.Value>
</Setter>
</Style>
```

Unnamed Styles

- Often we need the same ”Look and feel” inside the same scope (.xaml file)
- This can be done by using unnamed style

```
<Style TargetType="ListBox">
        <Setter Property="Background" Value="blue"></Setter>
</Style>
```

- Defines a ”local” style

Triggers and Style

```
<!-- The default style for all text boxes. -->
<Style TargetType="TextBox">
<Setter Property = "FontSize" Value ="14"/>
<Setter Property = "Width" Value = "100"/>
<Setter Property = "Height" Value = "30"/>
<Setter Property = "BorderThickness" Value = "5"/>
<Setter Property = "BorderBrush" Value = "Red"/>
<Setter Property = "FontStyle" Value = "Italic"/>
<!-- The following setter will be applied only when the text box is
in focus. -->
<Style.Triggers>
<Trigger Property = "IsFocused" Value = "True">
<Setter Property = "Background" Value = "Yellow"/>
</Trigger>
</Style.Triggers>
</Style>
```

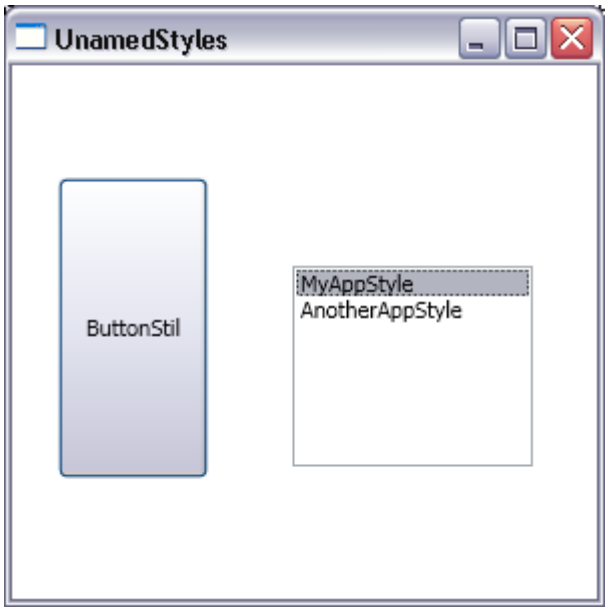
Multiple Triggers

```
<!-- The default style for all text boxes. -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- The following setter will be applied only when the text box is
  in focus AND the mouse is over the text box. -->
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
            <Condition Property = "IsFocused" Value = "True"/>
            <Condition Property = "IsMouseOver" Value = "True"/>
        </MultiTrigger.Conditions>
      <Setter Property = "Background" Value = "Yellow"/>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

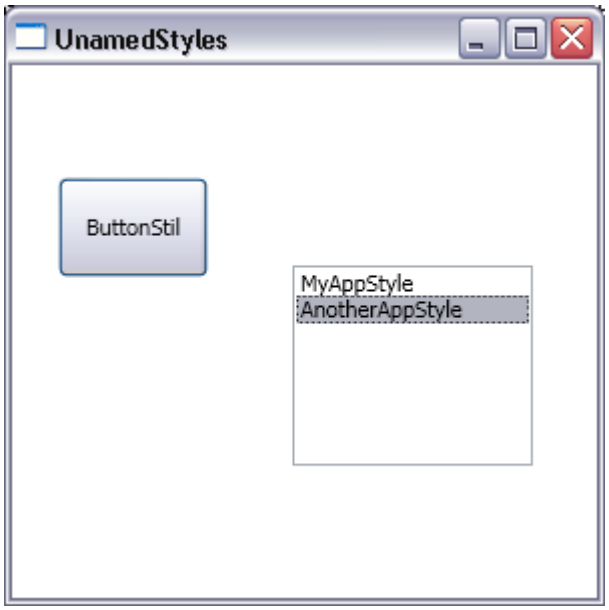
Set style using programming

- Start

- MyAppStyle



- AnotherAppStyle



- In App.xaml

```
<Application x:Class="AnimationsEksempel.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <Style x:Key="MyAppStyle" TargetType="Button">
            <Setter Property="Control.Height" Value="150"></Setter>
        </Style>

        <Style x:Key="AnotherAppStyle" TargetType="Button">
            <Setter Property="Control.Height" Value="50"></Setter>
        </Style>
    </Application.Resources>
</Application>
```

- C# code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
```

```
using System.Windows.Shapes;

namespace AnimationsEksempel
{
    /// <summary>
    /// Interaction logic for UnamedStyles.xaml
    /// </summary>
    public partial class UnamedStyles : Window
    {
        public UnamedStyles()
        {
            InitializeComponent();
            listBox1.Items.Add("MyAppStyle");
            listBox1.Items.Add("AnotherAppStyle");

        }

        private void listBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            Style curStyle = (Style)TryFindResource(listBox1.SelectedValue);
            if (curStyle != null)
                this.buttonStil.Style = curStyle;

        }
    }
}
```

Logical trees, Visual trees and default templates

- Mark-up is the logical tree
- Internal in WPF Visual trees are built
    - Uses templates and styles

Inspection of a logical tree

- XAML

```
<Window x:Class="WpfApplication1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Fun with Trees and Templates" Height="518"
Width="836" WindowStartupLocation="CenterScreen">
    <DockPanel LastChildFill="True">
        <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
            <StackPanel Orientation="Horizontal">
```

```
                <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
Click="btnShowLogicalTree_Click" Margin="4" BorderBrush="Blue" Height="40"/>
                <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
Click="btnShowVisualTree_Click" BorderBrush="Blue" Height="40" />
            </StackPanel>
        </Border>
        <TextBox x:Name="txtDisplayArea" Margin="10" Background="AliceBlue"
IsReadOnly="True"
BorderBrush="Red" VerticalScrollBarVisibility="Auto"
HorizontalScrollBarVisibility="Auto" />
    </DockPanel>
</Window>
```

- Logical tree

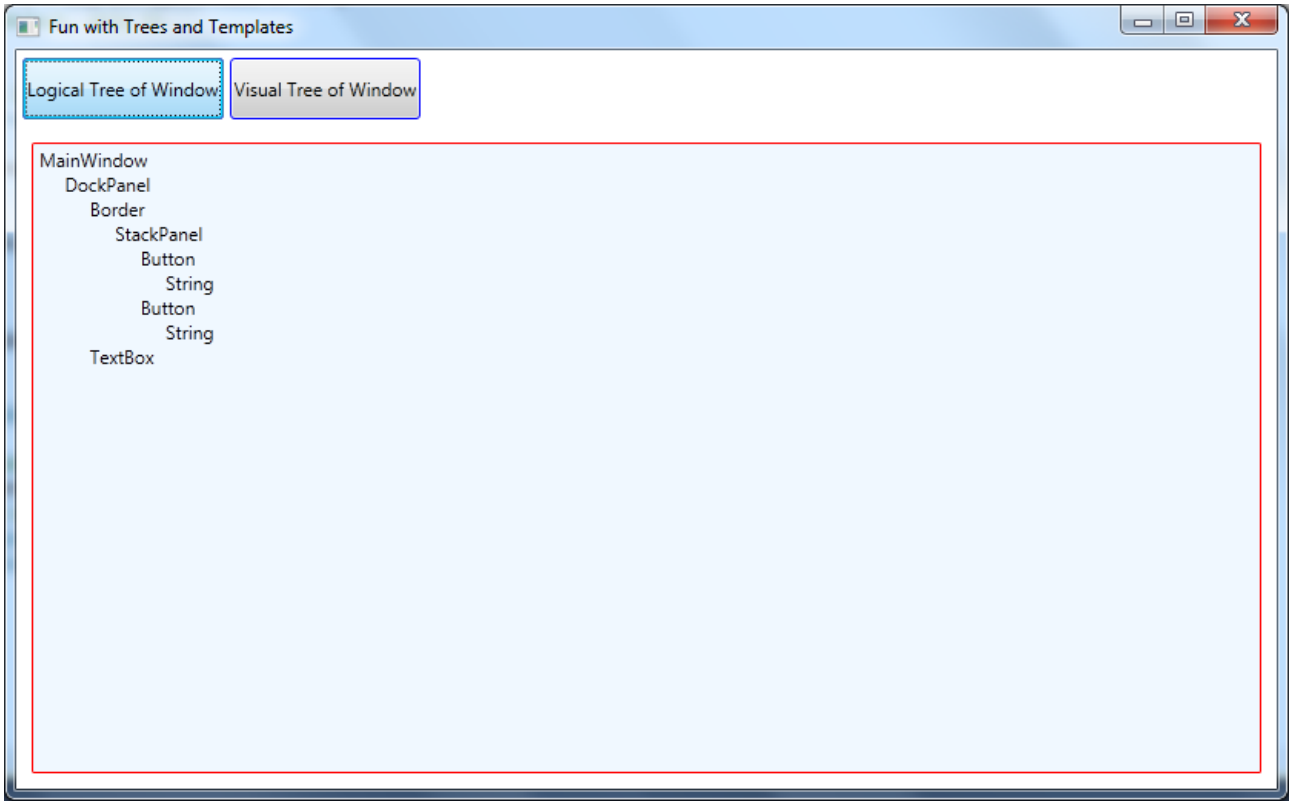
```
string dataToShow;
        public MainWindow()
        {
            //InitializeComponent();
        }

        protected void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
        {
            dataToShow = "";
            BuildLogicalTree(0, this);
            txtDisplayArea.Text = dataToShow;
        }

        void BuildLogicalTree(int depth, object obj)
        {
            // Add the type name to the dataToShow member variable.
            dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
            // If an item is not a DependencyObject, skip it.
            if (!(obj is DependencyObject))
                return;
            // Make a recursive call for each logical child.
            foreach (object child in LogicalTreeHelper.GetChildren(
            obj as DependencyObject))
                BuildLogicalTree(depth + 5, child);
        }
...
```

- Visual tree

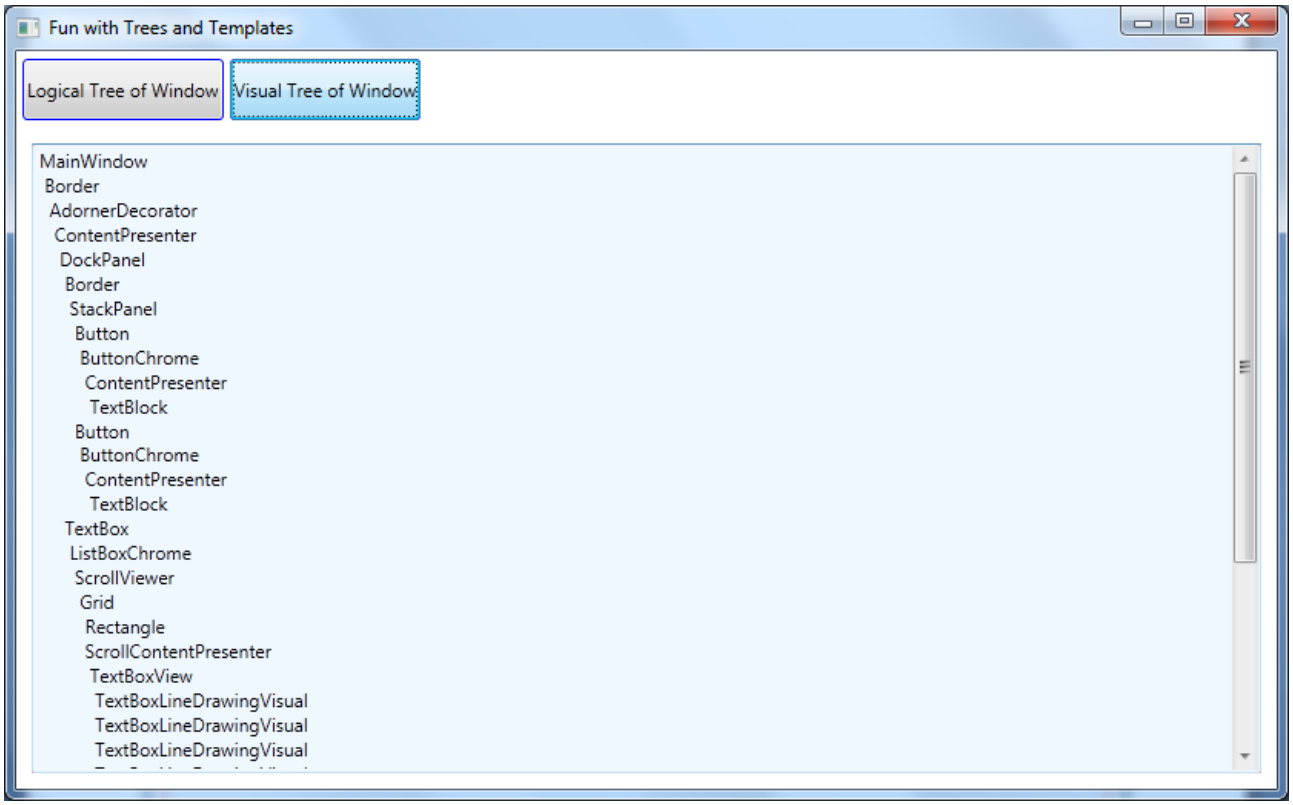
```
...
        protected void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
        {
            dataToShow = "";
            BuildVisualTree(0, this);
            this.txtDisplayArea.Text = dataToShow;
        }

        void BuildVisualTree(int depth, DependencyObject obj)
```

```
        {
            // Add the type name to the dataToShow member variable.
            dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
            // Make a recursive call for each visual child.
            for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
                BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
        }

...
```

- Output – logical tree:



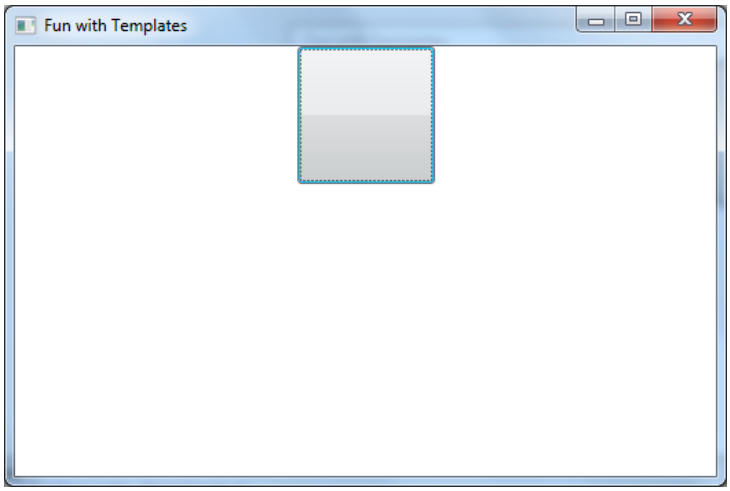
- Output – Visual tree:



- Notice: Low-level rendering

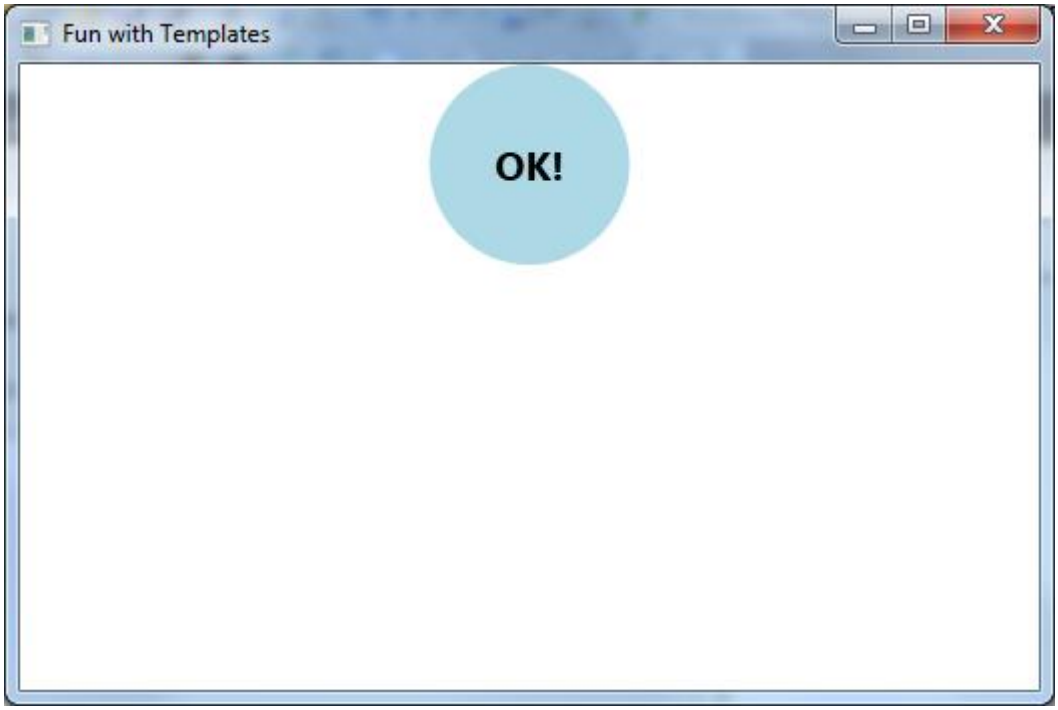
Templates

```
<Window x:Class="ButtonTemplate.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Fun with Templates" Height="350" Width="525">
    <StackPanel>
        <Button x:Name="myButton" Width="100" Height="100"
Click="myButton_Click"/>
    </StackPanel>
</Window>
```



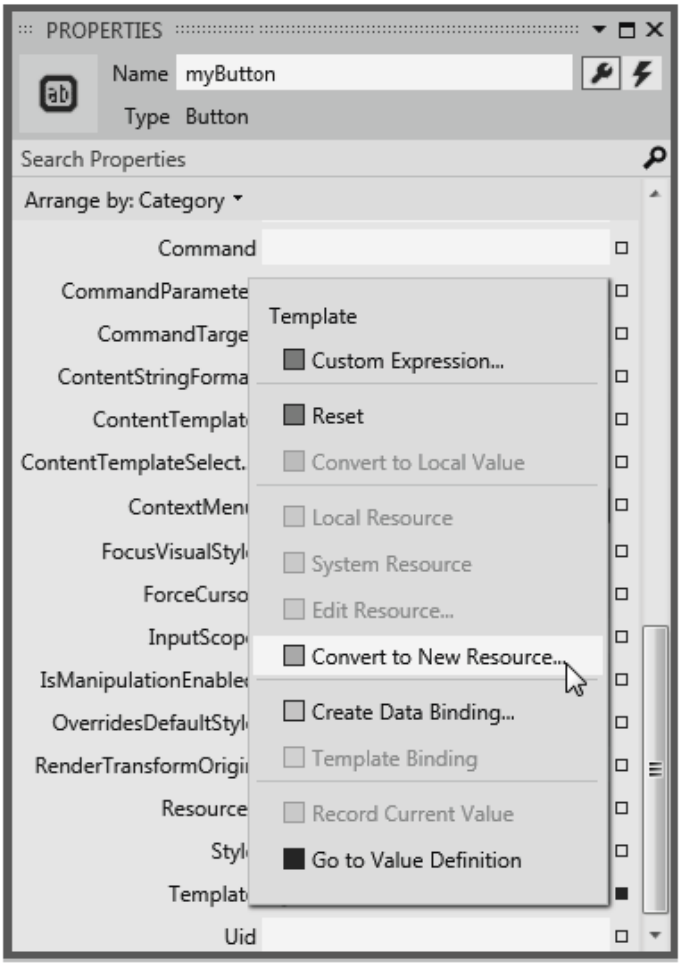
- New template (Button.Template)

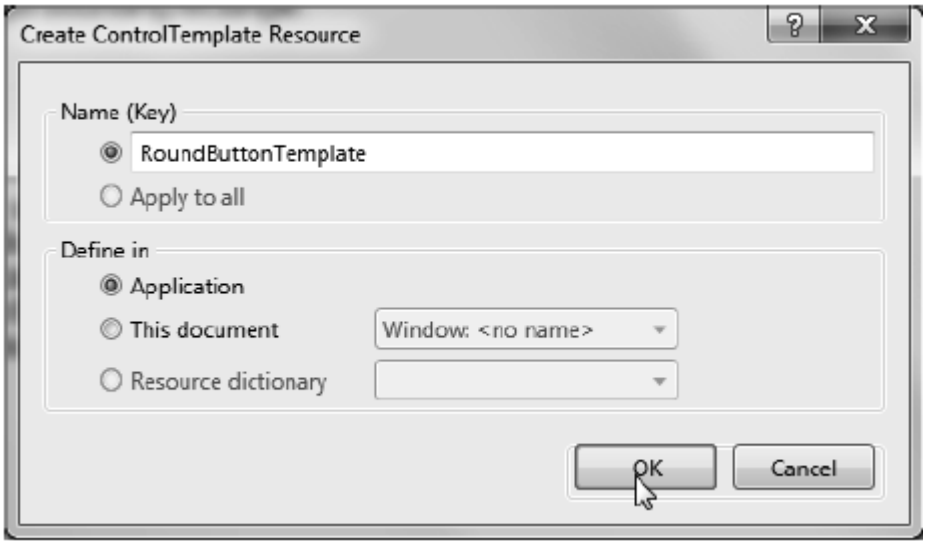
```
<Window x:Class="ButtonTemplate.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Fun with Templates" Height="350" Width="525">
    <StackPanel>
        <Button x:Name="myButton" Width="100" Height="100"
Click="myButton_Click">
            <Button.Template>
                <ControlTemplate>
                    <Grid x:Name="controlLayout">
                        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
                        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
                                                      HorizontalAlignment = "Center"
                               FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
                    </Grid>
                </ControlTemplate>
            </Button.Template>
        </Button>
    </StackPanel>
</Window>
```



Templates as resources

- Convert


PROPERTIES
Name myButton
Type Button
Search Properties
Arrange by: Category
Command
CommandParamete
CommandTarge
ContentStringForma
ContentTemplat
ContentTemplateSelect.
ContextMen
FocusVisualStyl
ForceCurso
InputScop
IsManipulationEnable
OverridesDefaultStyl
RenderTransformOrigi
Resource
Styl
Templat
Uid
Template
Custom Expression...
Reset
Convert to Local Value
Local Resource
System Resource
Edit Resource...
Convert to New Resource...
Create Data Binding...
Template Binding
Record Current Value
Go to Value Definition


Create ControlTemplate Resource
Name (Key)
RoundButtonTemplate
Apply to all
Define in
Application
This document
Window: <no name>
Resource dictionary
OK
Cancel

```
<Application x:Class="ButtonTemplate.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ControlTemplate x:Key="RoundButtonTemplate" TargetType="{x:Type Button}>
<Grid x:Name="controlLayout">
<Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
<Label x:Name="buttonCaption" VerticalAlignment = "Center"
                                              HorizontalAlignment = "Center"
                                    FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
            </Grid>
        </ControlTemplate>
    </Application.Resources>
</Application>
```

- Use the resource

```
<StackPanel>
    <Button x:Name="myButton" Width="100" Height="100"
Click="myButton_Click"
Template="{StaticResource RoundButtonTemplate}"></Button>
    <Button x:Name="myButton2" Width="100" Height="100"
Template="{StaticResource RoundButtonTemplate}"></Button>
    <Button x:Name="myButton3" Width="100" Height="100"
Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

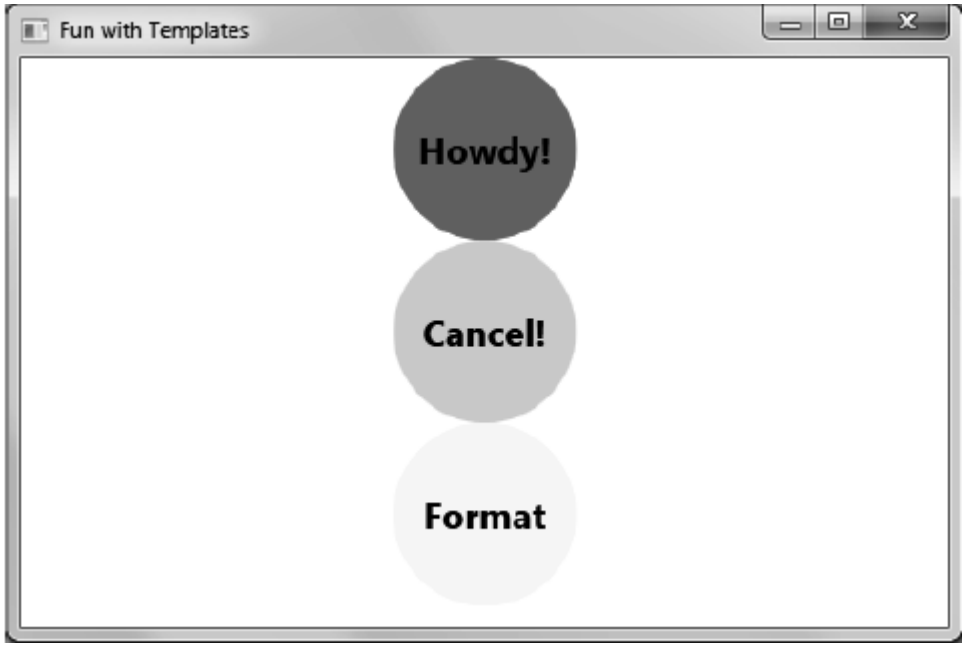
- One problem:

```
<StackPanel>
    <Button x:Name="myButton" Width="100" Height="100"
Background="Red" Content="Howdy!"
Click="myButton_Click"
Template="{StaticResource RoundButtonTemplate}" />
    <Button x:Name="myButton2" Width="100" Height="100"
Background="LightGreen" Content="Cancel!"
Template="{StaticResource RoundButtonTemplate}" />
    <Button x:Name="myButton3" Width="100" Height="100"
Background="Yellow" Content="Format"
Template="{StaticResource RoundButtonTemplate}" />
</StackPanel>
```

- All htree buttons are blue with the text: OK

- Solution: Use {templateBinding}

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button">
    <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
        <Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
FontSize="20" FontWeight="Bold"
HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
    <ControlTemplate.Triggers>
        ...
    </ControlTemplate.Triggers>
</ControlTemplate>
```



ContentPresenter

- An Ellipse does not have ”Content”
- Use ContentPresenter

The Role of ContentPresenter

When you designed your template, you used a `Label` to display the textual value of the control. Like the `Button`, the `Label` supports a `Content` property. Therefore, given your use of `{TemplateBinding}`, you could define a `Button` that contains complex content beyond that of a simple string.

However, what if you need to pass in complex content to a template member that does *not* have a `Content` property? When you want to define a generalized *content display area* in a template, you can use the `ContentPresenter` class as opposed to a specific type of control (`Label` or `TextBlock`). There is no need to do so for this example; however, here is some simple markup that illustrates how you could build a custom template that uses `ContentPresenter` to show the value of the `Content` property of the control using the template:

```
<!-- This button template will display whatever is set to the Content of the hosting button. -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}"/>
    <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```

Templates combined with Style

```
<!-- A style containing a template. -->

<Style x:Key ="RoundButtonStyle" TargetType ="Button">
    <Setter Property ="Foreground" Value ="Black"/>
    <Setter Property ="FontSize" Value ="14"/>
    <Setter Property ="FontWeight" Value ="Bold"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="100"/>
    <!-- Here is the template! -->
    <Setter Property ="Template">
        <Setter.Value>
            <ControlTemplate TargetType ="Button">
                <Grid x:Name="controlLayout">
                    <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
                    <Label x:Name="buttonCaption" Content ="{TemplateBinding Content}"
                           HorizontalAlignment="Center" VerticalAlignment="Center" />
                </Grid>
                <ControlTemplate.Triggers>
                    <Trigger Property = "IsMouseOver" Value = "True">
                        <Setter TargetName = "buttonSurface" Property = "Fill" Value =
                                                                        "Blue"/>
                        <Setter TargetName = "buttonCaption" Property = "Foreground" Value
                                                                        = "Yellow"/>
                    </Trigger>
                    <Trigger Property = "IsPressed" Value="True">
                        <Setter TargetName="controlLayout"
                                        Property="RenderTransformOrigin" Value="0.5,0.5"/>
                        <Setter TargetName="controlLayout" Property="RenderTransform">
                            <Setter.Value>
                                <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
                            </Setter.Value>
                        </Setter>
                    </Trigger>
```

```
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

- In XAML (use):

```
...
<Button x:Name="myButton" Background="Red" Content="Howdy!"
Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>

...
```