Noteset #11 WPF

WPF is the **Windows Presentation Framework**, Microsoft's current API for building GUIs for Windows .NET based apps. WPF is an evolutionary replacement for Windows Forms. The release of WPF has caused .NET developers to ask if Windows Forms study and development should be discontinued in favor of WPF. The answer at the current time is no. WPF introduces many technical improvements compared to Windows Forms, but Windows Forms is not dead, so it is worthwhile to learn them both. Now that WPF has been released for a few years, some blog writers ask what is the future even of WPF itself. Currently WPF seems alive and well, to speculate about Microsoft's strategic vision over the next decade for any of its products is outside the scope of this course.

Some Notable Differences between WPF and WinForms

- **Code + Markup:** WPF permits (but does not require) the coding of the GUI to be divided into markup (using the XAML language) and "code-behind" (using C#). Markup is used to define layout, and code is used for anything that is not directly layout -- in practice this means all the event handler delegates. There is nothing to prevent the same person from being responsible for composing both the markup and the code, but a significant goal of WPF is to allow a development team to assign these tasks to different specialists – a graphic designer for the markup and a traditional code developer for the code.
- **DirectX:** Graphics in WPF is based on DirectX, while graphics in WinForms is based on the older and much more limited GDI standard.
- **RoutedEvent:** a generalization of earlier approaches to handling events. WPF supports two event routing strategies: regular "bubbling" event routing as in WinForms, and the newer "tunneling" event routing. The difference in the strategies has to do with the arrangement of GUI elements in the visual hierarchy. Elements sit on top of each other along the z-axis with different z-order values. Tunneling events, also called preview events, propagate back-to-front (from root to leaf), while bubbling events propagate front-to-back (from leaf to root). An event propagates from one object to another until it is marked "handled."
- **Dependency Properties and Attached Properties:** enhancement of regular class properties that allow notification of property changes and other new features. Attached properties are needed to solve some layout issues with interacting components when expressed in XAML. The problem would not come up in code.

Build Process for WPF

A WPF app starts off as a mix of XAML and C# source files. Under the hood, the XAML is converted to C#, then combined with the other C# files, compiled, and linked into an executable. Since this build process is complex, it can't be done as a command line task. A code-only WPF app, one that does not use XAML, can be built from the command line. The main problem is that the compiler must be notified of several additional DLLs. See this article for more info:

http://msdn.microsoft.com/en-us/library/aa970678%28VS.90%29.aspx

Suggestion: create a file named wpfcsc.cmd containing the following:

```
set prefix=C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0
csc.exe /out:%~n1.exe /target:winexe %1 ^
        /reference:"%prefix%\presentationframework.dll" ^
        /reference:"%prefix%\windowsbase.dll" ^
        /reference:"%prefix%\presentationcore.dll"
```

From the SDK command prompt, type

```
% wpfcsc MyApp.cs
```

to produce

```
MyApp.exe
```

which can now be executed as a command

```
% MyApp.exe
```

PowerShell Command Line

The code above is for a traditional Batch file to be executed by the Command Prompt. A better approach would be to write a PowerShell PS1 script for execution by PowerShell. If you have some experience with PowerShell, you could try it, but it's not simple.

First, you will need to make sure that PowerShell runs the most recent version of the .NET CLR. You can find out which versions of various assemblies PowerShell is using by running:

```
$psversiontable
```

Then to make sure PowerShell uses the latest CLR, add these two registry settings:

```
reg add hklm\software\microsoft\.netframework /v OnlyUseLatestCLR /t REG_DWORD /d 1
reg add hklm\software\wow6432node\microsoft\.netframework /v OnlyUseLatestCLR /t REG_DWORD /d 1
```

On my system, this updated the CLR from 2.X to 4.X. By default, for backward compatibility, PowerShell runs an earlier version of the CLR.

Next, you will need to import some definitions of Visual Studio variables to get the C# compiler to be recognized from PowerShell. The easiest way is to create a PS1 like this and place it into your PowerShell configuration directory, which for me is C:\Users\Rick\Documents\WindowsPowerShell:

```
pushd 'c:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Tools\'
cmd /c "vsvars32.bat&set" |
foreach {
  if ($_ -match "=") {
    $v = $_.split("="); set-item -force -path "ENV:\$($v[0])"  -value "$($v[1])"
  }
}
popd
```

This finds a file called **vsvars.bat** provided by your Visual Studio install and adds these variables to your PowerShell environment. After this command, the csc.exe command should be available from the PowerShell prompt.

Finally, you need a script to compile your C# files and link in the required WPF DLLs. Create a PS1 file, say wpfcsc.ps1, like this:

```
param(
      [string] $fn1=$null
)

$prefix = "C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0"

csc.exe /out:wpfapp.exe /target:winexe $fn1 `
      /reference:"$prefix\PresentationFramework.dll" `
      /reference:"$prefix\WindowsBase.dll" `
      /reference:"$prefix\PresentationCore.dll"
```

The param block defines a command line argument named $fn1, which by default will be the first param on the command line. Then it runs the csc.exe command with the required WPF-specific DLLs referenced with the /reference modifier.

Note that you will need to check your installation for the exact install directories for both Visual Studio variables and WPF DLLs, the directory names can vary by a bit from one version to the next. You will also have to run PowerShell as administrator and use the PowerShell cmdlet

```
Set-ExecutionPolicy remotesigned
```

to turn on ability to run PowerShell scripts in the first place.

Realistically, it's probably not worth the effort to use a command line build, just create a WPF project in Visual Studio and let it manage the build for you. And if the app includes a mix of XAML and C#, a command line build is no longer an option.

Console Output

No console output from a WPF application is not supported. Calls to **Console.WriteLine()** compile but do nothing at runtime. Also, a WPF application launched from the command prompt is not a child process of the command prompt shell that launched it. If you need to terminate it, you will have to go to the list of running processes, get its process id, and kill it by process id.

Code + Markup

- Some developers feel strongly that the concept of GUI layout is best documented in markup.
- XAML (eXtensible Application Markup Language) based on XML is defined for this purpose.
- A WPF application can be written in code-only or in code+markup.
- Markup-only is possible for extremely simple GUIs.
- Layout defined in markup files: MyApp.xaml
- Code associated with that markup is defined in an associated "code-behind" file: MyApp.xaml.cs

Namespaces

For WPF, use

System	
System.Windows	**🡐 class Window here**
System.Windows.Controls	**🡐 most GUI building block classes here**
System.Windows.Input	
System.Windows.Media	
System.Windows.Drawing	

In comparison, for WinForms, use

System	
System.Drawing	
System.Windows.Forms	**🡐 most GUI building block classes here**

Separating vs. Mixing WinForm and WPF Classes

WPF is a different API from WinForms, but there is a strong similarity. There is a similar relationship between WinForms and WPF as between Java AWT and Java Swing. But in the .NET case, there is more possibility for confusion, since in some cases, the names of controls were not modified between the two APIs. In Java, Swing added a "J" to the name of many components introduced by AWT to produce a Swing version: **java.awt.Button** vs. **javax.swing.JButton**, for example.
In .NET, new namespaces were introduced but many controls have the same name in the two namespaces: **System.Windows.Forms.Button** for WinForms, and **System.Windows.Controls.Button** for WPF, for example. Even though they have the same name **Button**, they belong to different namespaces and have different properties and behaviors.

Since WPF is a newer API than WinForms, the full complement of GUI-building controls has in the past been more complete in WinForms that in WPF. WPF re-implementation of controls introduced in WinForms has been slow. For that reason, there is some motivation for WPF app developers to reach over into the WinForms namespaces to use a WinForm control in the middle of an otherwise WPF app if an equivalent WPF version of the control has not yet been released.

My recommendation for starters is to not mix classes.

Organization of a Code-Only WPF App

Top-Level Container

Window (not **Form**), plus **Application** (must be instantiated, not simply used statically)

Boilerplate for Code-Only App

- Create a Window object
- Call its Show() method.
- Create an Application object.
- Call its Run() method.

```
Window win = new Window();
win.Show();
Application app = new Application();      // Application object
app.Run();                                // non-static method Run()
```

Alternatively

```
Window win = new Window();
Application app = new Application();
app.Run(win);                             // non-static method Run()
                                          // Run() calls Show()
```

A few rules:

- A WPF application can create multiple windows
- It can only create one Application object.
- The call to Run() must be last. It does not return until the window is closed.

Compare this to WinForms

```
Form form = new Form();
Application.Run(form);                    // static method Run()
```

Static Properties of class Application

Current	static of type Application
MainWindow	non-static of type Window

For example, the main window of the current application is available as

Application.Current.MainWindow

Required [STAThread] Directive for Main Method
If you do a manual build and create your own Main() method, use the directive **[STAThread]** just as for a Windows Form app.

Relationship Between Windows and Applications
The Application object for the app keeps track of some global information such as a list of windows and shutdown mode.

Properties of class Application

Windows	of type WindowCollection
MainWindow	of type Window
ShutdownMode	of type ShutdownMode enumeration (OnMainWindowClose, OnExplicitShutdown, etc.)

The main window is the window that must be closed to terminate the program.

Some class Application Events

Startup	shortly after Run() is called
Exit	shortly before Run() returns
SessionEnding	OS shutdown (only received for specific compilation options)

Properties of class Window

Title	of type String
Owner	of type Window
ShowInTaskbar	of type bool

Regarding WPF Application in Visual Studio: Where is the Main() Method?
- In Solution Explorer, click "Show All Files" button.
- Go to "obj ➔ x86 ➔ Debug ➔ App.g.cs"

Inheritance
Application classes can inherit from class Application or class Window

If the app inherits from Window the boilerplate is as before. This is more common for code-only projects.

If the app inherits from application, the boilerplate is
```
MyApp app = new MyApp();
app.Run();
```

You can also write subclasses for both the Application and the Window.
```
class Driver {
      [STAThread]
      public static void Main() {
            MyApplication app = new MyApplication();
            app.Run();
      }
}
class MyApplication : Application {
      protected override void OnStartup(StartupEventArgs args) {
            base.OnStartup(args);
            MyWindow win = new MyWindow();
            win.Show();
      }
}
class MyWindow : Window {
      public MyWindow() {
            Title = "…";
      }
      protected override void OnMouseDown(MouseButtonEventArgs args) {
            …
      }
}
```

Window Properties

Left	of type double (device independent units)
Top	of type double (device independent units)
Width	double
Height	double

Initially undefined, remain undefined until you set them.

ActualWidth	double
ActualHeight	double

The actual width and height, once the Window is visible.

Use Left and Right to put the Window at a particular position on the desktop.

Other Properties

The SystemParameters class contains some static properties that are useful for establishing values of some global application properties.

SystemParameters.PrimaryScreenWidth
SystemParameters.PrimaryScreenHeight
SystemParameters.WorkArea.Width
SystemParameters.WorkArea.Height

Layout and the Content Property

Window is an example of a control that uses a **Content** property. The Content property can be set to only a single control. It is similar to the content pane of a JFrame container in Java Swing. If the Window needs to display multiple controls with layout, then use one of the panels with a built-in layout and set this panel to be the Content property of the Window.

- StackPanel (similar to a JPanel with a BoxLayout in Java Swing)
- WrapPanel (similar to a JPanel with the default FlowLayout)
- DockPanel (similar to a JPanel with BorderLayout)
- Grid (similar to a JPanel with a GridBagLayout)

A typical simple layout would be

- Create a panel
- Assign it to the Content property of the window.
- Create other controls.
- Add them to the panel.

```
public class Counter : Window {

   Button btn;
   TextBox text;

   public Counter() : base() {
         WrapPanel wp = new WrapPanel();

         btn = new Button();
         btn.Content = "Inc";
         btn.Margin = new Thickness(5);

         text = new TextBox();
         text.Width = 60;
         text.Text = "0";

         wp.Children.Add(btn);
         wp.Children.Add(text);

         Width = 100;
         Height = 100;
         Content = wp;
   }
}
```

Canvas for Absolute Positioning
Because Window uses the Content property, there is no meaningful way to attach a control to the Window at specific coordinates. And with the layout-based panels, the location of the controls is managed automatically and not set programmatically. Letting a layout panel do layout tasks is preferable for most tasks, but how to do absolute positioning if you really need it? This is where the Canvas control comes in. Create a Canvas control, add elements to the Canvas at specific positions, then set the Canvas object to be the Content property of the Window that displays it.

```
        rect = new Rectangle();
        rect.Stroke = Brushes.Black;
        Canvas.SetLeft(rect, 30);
        Canvas.SetTop(rect, 50);
        rect.Width = 200;
        rect.Height = 100;
        cnv.Children.Add(rect);
```

This also works for controls like buttons.

```
public class CanvasDemo : Window {

        public Canvas cnv;
        public Button btn1,btn2,btn3;

        public CanvasDemo() : base() {
                cnv = new Canvas();
                btn1 = CreateButton(cnv, "Button 1", 10, 10);
                btn2 = CreateButton(cnv, "Button 2", 10, 50);
                btn3 = CreateButton(cnv, "Button 3", 10, 90);
                this.Content = cnv;
        }

        public Button CreateButton(Canvas c, string title, double left, double top) {
                Button b = new Button();
                b.Content = title;
                b.FontSize = 18;
                c.Children.Add(b);
                Canvas.SetLeft(b,left);
                Canvas.SetTop(b,top);
                return b;
        }

        [STAThread]
        public static void Main(string[] args) {
                CanvasDemo window = new CanvasDemo();
                Application app = new Application();
                app.Run(window);
        }
}
```

The controls may or may not overlap, the Canvas container does not enforce any layout.

In this example, the buttons assume their default size, although Width and Height properties could be set via assignment. But position relative to Canvas origin, uses the static methods Canvas.SetLeft() and Canvas.SetTop().

Properties such as "Left" and "Top" for class Button are called **attached properties**. This is a special feature of WPF. An attached property is a property that is not a native property of a class. But if an object needs extra properties in order to interact with another class, then the extra properties can be attached or "glued on" and assigned a value. But the properties can only be accessed by a special static method as shown, they cannot be assigned values using the usual syntax for native properties.

Canvas and MouseEvents
A Canvas will not receive mouse events unless the Background property is set.

Simple Graphics
In WinForms, graphics primitives for drawing lines and shapes can be added to an OnPaint() method for a container like a Form. WPF has no Paint event and OnPaint() handler. It has a similar pair Render and OnRender(), but these are used much less frequently than Paint/OnPaint() are used in WinForms. Instead, WPF uses a different approach, graphics are constructed as a collection of objects which derive from class Shape, and which know how to draw themselves, like any other control. Create the shape as an instantiated visual object, attach it to a container, and it will automatically display itself, there is no need to write your own low level graphics primitives to draw the shapes.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

public class Drawing : Window {
      public Drawing() : base() {
            SolidColorBrush br = new SolidColorBrush();
            br.Color = Color.FromArgb(255,255,255,0);

            Ellipse e = new Ellipse();
            e.Fill = br;
            e.StrokeThickness = 2;
            e.Stroke = Brushes.Black;
            e.Width = 200;
            e.Height = 100;

            StackPanel panel = new StackPanel();
            panel.Children.Add(e);

            Content = panel;
            Width = 300;
            Height = 300;
      }
}
```

Delegates
WPF events and delegates are structured similarly to Windows Forms events and delegates. But there has been some reorganization and renaming. So the events and delegates are similar but not identical in the two cases.

MouseEventHandler (System.Windows.Input)

This delegate is used with the following attached events:
Mouse.MouseEnter
Mouse.MouseLeave
Mouse.MouseMove
Mouse.PreviewMouseMove

This delegate is used with the following routed events. These routed events forward the previously listed attached events to make them more accessible to the general element model in WPF.
UIElement.MouseEnter
UIElement.MouseLeave
UIElement.MouseMove
UIElement.PreviewMouseMove

MouseButtonEventHandler (System.Windows.Input)

This delegate is used with the following attached events.
Mouse.MouseDown
Mouse.MouseUp
Mouse.PreviewMouseDown
Mouse.PreviewMouseUp

This delegate is used with the following routed events. These routed events forward the previously listed attached events to make them more accessible to the general element model in WPF.
UIElement.MouseDown
UIElement.MouseUp
UIElement.MouseLeftButtonDown
UIElement.MouseLeftButtonUp
UIElement.MouseRightButtonDown
UIElement.MouseRightButtonUp

Class System.Windows.Window
Class Window recognizes the usual set of events. Here are some of the mouse related ones for example:

- MouseDown
- MouseEnter
- MouseLeave
- MouseLeftButtonDown
- MouseLeftButtonUp
- MouseMove
- MouseRightButtonDown
- MouseRightButtonUp
- MouseUp

Plus Preview versions of these events.

Event Handling Methods

The process for responding to events in WPF is very parallel to the process in WinForms. There has been some changes in the names of event args and delegates. The various classes and delegates are found mostly in the System.Windows and the System.Windows.Input namespaces.

Method	**Args**	**Delegate**
OnMouseUp	MouseButtonEventArgs	MouseButtonEventHandler
OnMouseMove	MouseEventArgs	MouseEventHandler
OnClick	RoutedEventArgs	RoutedEventHandler

Think of RoutedEventArgs and RoutedEventHandler as the “default” types for event args and delegates. More specific kinds of events have their own.

Canvas and Geometric Shapes

Rendering instead of Painting a Control

The "OnPaint" method from WinForms is replaced by "OnRender()", defined in base class UIElement:

```
        protected virtual void OnRender(DrawingContext drawingContext);
```

Note that there is no corresponding "Render" event, so the method behaves more like a callback than a delegate that responds to an event.

Subclassing Canvas and overriding OnRender() is possible (visual layer drawing), but not the default approach. In WPF, geometric shapes are created and added to a Canvas and know how to draw themselves. OnRender() will not be overridden as frequently as OnPaint() in WinForms or paintComponent() in Java Swing.

See MSDN article http://msdn.microsoft.com/en-us/library/ms745163.aspx for Canvas examples

Shape-related classes are defined in namespace System.Windows.Shapes.

For example, see class Rectangle.

```
        Rectangle r = new Rectangle();      // this is the only constructor
        r.Width = …;
        r.Height = …;
        r.Stroke = Brushes.Black;           // must set, no default
        r.StrokeThickness = 2;              // optional, default = 1
```

Attaching a shape to a canvas and rendering it is handled differently in WPF from WinForms and from Java Swing.

1 Create a Canvas and a Shape
```
        Canvas c = new Canvas();
        Rectangle r = new Rectangle();
```

2 Set Shape Properties Width, Height, Stroke, …
```
        … // see example above
```

3 Attach Shape to Canvas
```
        c.Children.Add(r);
```

4 Define the Shape's position relative to the Canvas that it is attached to
```
        Canvas.SetLeft(r,…);
        Canvas.SetTop(r,…);
```

Note that Canvas.SetLeft() and Canvas.SetTop() are static methods of class Canvas. The methods are not applied to either the canvas or shape object directly. This is because Left and Top are attached properties for class Canvas.

5 The remaining code is boilerplate
```
        Window win = new Window();
        win.Content = c;
        Application app = new Application();
        app.Run(win);
```

Panels and Controls

Layout is provided by a series of panel-like containers that maintain a collection of child objects in a layout arrangement that is built in (and fixed) for that type of container.

- Panel
- StackPanel
- WrapPanel
- DockPanel
- Grid

The panel becomes the Content of the window.
Controls are added to the Children property of the panel.

```
Window win = new Window();
StackPanel p = new StackPanel();
win.Content = p;

Button b1 = new Button();
p.Children.Add(b1);
Button b2 = new Button();
p.Children.Add(b2);
```

Grid
Similar to GridLayout or GridBagLayout in Java Swing, or TableLayoutPanel in Windows Forms
Example from MSDN entry for Class Grid:

```
Grid myGrid = new Grid();
myGrid.Width = 250;
myGrid.Height = 100;
myGrid.HorizontalAlignment = HorizontalAlignment.Left;
myGrid.VerticalAlignment = VerticalAlignment.Top;
myGrid.ShowGridLines = true;

// Define the Columns for 3 cols
ColumnDefinition colDef1 = new ColumnDefinition();
myGrid.ColumnDefinitions.Add(colDef1);
...

// Define the Rows for 4 rows
RowDefinition rowDef1 = new RowDefinition();
myGrid.RowDefinitions.Add(rowDef1);
```

```
// Add the first text cell to the Grid
TextBlock txt1 = new TextBlock();
txt1.Text = "2005 Products Shipped";
txt1.FontSize = 20;
txt1.FontWeight = FontWeights.Bold;
Grid.SetColumnSpan(txt1, 3);
Grid.SetRow(txt1, 0);

// Add the second text cell to the Grid
TextBlock txt2 = new TextBlock();
txt2.Text = "Quarter 1";
txt2.FontSize = 12;
txt2.FontWeight = FontWeights.Bold;
Grid.SetRow(txt2, 1);
Grid.SetColumn(txt2, 0);

// Add the third text cell to the Grid
TextBlock txt3 = new TextBlock();
txt3.Text = "Quarter 2";
txt3.FontSize = 12;
txt3.FontWeight = FontWeights.Bold;
Grid.SetRow(txt3, 1);
Grid.SetColumn(txt3, 1);

…

// Add the TextBlock elements to the Grid Children collection
myGrid.Children.Add(txt1);
myGrid.Children.Add(txt2);
myGrid.Children.Add(txt3);
myGrid.Children.Add(txt4);
myGrid.Children.Add(txt5);
myGrid.Children.Add(txt6);
myGrid.Children.Add(txt7);
myGrid.Children.Add(txt8);
```

Text

- TextBlock: text with no visual control boundary
- TextBox
- RichTextBox

Routed Events

WPF introduces the concept of routed events that interact with the visual hierarchy in a more general way than in WinForms.

System.Windows.RoutedEvent

is the base class of WPF events. The “routing” in the title refers to how the event will be processed by the objects in the visual hierarchy for a given app. Specifically, a routed event will have an opportunity to interact with multiple elements in the hierarchy. The two possibilities are

- **Bubbling**: event is handled by the leaf node first (the event source), and is then passed to other objects in the direction of the root of the hierarchy (or front-to-back in z-order terms). This is the “usual” event behavior.
- **Tunneling**: event is handled by the root node first, and is then passed to other objects in the direction of the leaf node in the hierarchy (back-to-front in z-order terms). This is not the usual approach, but is a useful and compact way to get the GUI to respond to events in some cases. You could say the contribution of WPF compared to WinForms is to expand the existing set of bubbling events to include a tunneling event equivalent.

The words “bubbling” and “tunneling” do not appear in the names of events. Instead, regular events are assumed to be bubbling, and are paired with a “preview” event that is assumed to be tunneling. For example

MouseDown (bubbling) ⇔ PreviewMouseDown (tunneling)

Note: some controls use events in an idiosyncratic way. A Button can respond to a Click event, but you may not see a response to a MouseDown event for a Button, because this event gets incorporated into the Click event.

Example #1
Imagine a simple WPF with the following visual hierarchy:
    Window ➔ StackPanel ➔ Grid ➔ Button/TextBox
We will define delegates for two events KeyDown and PreviewKeyDown for the Window, StackPanel, and Grid objects.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;

public class EventDemo : Window {

        public Button b1;
        public TextBox tb1;
        public Grid grid;
        public StackPanel sp;

        public EventDemo() : base() {
                b1 = new Button();
                b1.Content = "Click";
                b1.MaxWidth = 100;

                tb1 = new TextBox();
                tb1.Text = "text";
                tb1.MaxWidth = 100;

                grid = new Grid();
                grid.RowDefinitions.Add(new RowDefinition());
                grid.ColumnDefinitions.Add(new ColumnDefinition());
                grid.ColumnDefinitions.Add(new ColumnDefinition());
                grid.Children.Add(b1);
                Grid.SetRow(b1,0);
                Grid.SetColumn(b1,0);
                grid.Children.Add(tb1);
                Grid.SetRow(tb1,0);
                Grid.SetColumn(tb1,1);

                sp = new StackPanel();
                sp.Children.Add(grid);

                this.Content = sp;
                this.Width = 300;
                this.Height = 150;

                this.PreviewKeyDown += wPKD;
                sp.PreviewKeyDown   += spPKD;
                grid.PreviewKeyDown += gPKD;

                this.KeyDown        += wKD;
                sp.KeyDown          += spKD;
                grid.KeyDown        += gKD;
        }

        public void wPKD (Object obj, KeyEventArgs args) { MessageBox.Show("wPKD");  }
        public void spPKD(Object obj, KeyEventArgs args) { MessageBox.Show("spPKD"); }
        public void gPKD (Object obj, KeyEventArgs args) { MessageBox.Show("gPKD");  }
        public void wKD  (Object obj, KeyEventArgs args) { MessageBox.Show("wKD");   }
        public void spKD (Object obj, KeyEventArgs args) { MessageBox.Show("spKD");  }
        public void gKD  (Object obj, KeyEventArgs args) { MessageBox.Show("gKD");   }
}
```

After building and running the app, the window will be displayed. Before any keys are pressed the window receives the focus. At this point a key down event will generate the series:
    Preview Key Down for Window, Key Down for Window
Using the mouse, click on the Button to give it the focus, then press a key. The series is now
    PKD Window, PKD StackPanel, PKD Grid, KD Grid, KD StackPanel, KD Window
Inside any of the delegates the code
    args.Handled = true;
will terminate the chain of events by marking the event "handled" at that point.

Note the following before experimenting with other components and events:

- Any control responding to a "Click" event (such as Button) will absorb lower level events such as MouseDown and MouseUp.
- A control like TextBox processes key events in a unique way.
- Using a MessageBox to track the processing of a click event may not work, because the MessageBox itself requires a click to dismiss it, possibly interfering with the state of the event processing that was being monitored.

Example #2

Use the code from example #1, but set the text box to not be focusable:

```
tb1.Focusable = false;
```

change the event delegate assignment as follows

```
this.PreviewMouseDown += wPMD;
sp.PreviewMouseDown   += spPMD;
grid.PreviewMouseDown += gPMD;

this.MouseDown        += wMD;
sp.MouseDown          += spMD;
grid.MouseDown        += gMD;
```

and use the following definitions for the delegates:

```
public void wPMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "1";
      MessageBox.Show("1");
}

public void spPMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "2";
      MessageBox.Show("2");
}

public void gPMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "3";
      MessageBox.Show("3");
}

public void wMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "4";
      MessageBox.Show("4");
}

public void spMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "5";
      MessageBox.Show("5");
}

public void gMD(Object obj, MouseButtonEventArgs args) {
      //tb1.Text += "6";
      MessageBox.Show("6");
}
```

Run the application by clicking the mouse inside the text box. The delegates have been defined so that the execution can be tracked either by updating the text box on the GUI, or by displaying a message box dialog. Initially, the text box version has been commented out, leaving the message box version. Try the program for both versions and note the effect.

By making the text box not focusable, we ensure that both the PreviewMouseDown and MouseDown events are individually seen by the text box. In the message box version of the program, you will note that the message box event handling will interfere with the event handling for the original application that we are trying to observe.

Code + Markup With XAML

Simple Example Built With Visual Studio

WPF applications that include markup cannot be built easily from the command prompt. The XAML markup is converted to C#, and the result is compiled along with the code-behind to create the final application. Visual Studio will perform all the necessary tasks to build the executable.

A new WPF project in Visual Studio creates four files:

- MainWindow.xaml
- MainWindow.xaml.cs
- App.xaml
- App.xaml.cs

The XAML is a Window tag with a Grid tag as its Content property.

```
<Window x:Class="WpfApplication1.MainWindow" …>   (edited/simplified)
      <Grid>
      </Grid>
</Window>
```

The code-behind is mostly empty, but is available for defining any event handling delegates.

```
namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

    }
}
```

The App.xaml defines the Application and points it to the MainWindow.xaml file through the StartupUri attribute:

Application.xaml

```
<Application x:Class="WpfApplication1.App" StartupUri="MainWindow.xaml">

</Application>
```

To summarize, the Code + Markup introduces a namespace specific for the project – **WpfApplication1**. And two classes in that namespace – class **MainWindow** and class **App**. Together, the two XAML tags are the equivalent of the programmatic app launch:

```
App app = new App();
MainWindow window = new MainWindow();
app.Run(window);
```

After opening a new WPF app, we will modify the XAML by hand to build a custom GUI consisting of a Window containing a WrapPanel, and the WrapPanel containing a TextBox and two Buttons.

```
<Window x:Class="WpfApplication12.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">
    <WrapPanel>
        <TextBox Name="tb1" Text="0" Width="50" Margin="5" Padding="5"> />
        <Button Name="incbtn" Click="doincclick" Margin="5" Padding="5">
            Inc
        </Button>
        <Button Name="decbtn" Click="dodecclick" Margin="5" Padding="5">
            Dec
        </Button>
    </WrapPanel>
</Window>
```

Margin and Padding Attributes
These are used for internal and external spacing of the components.

Name Attribute
Each tag should be given a Name attribute, with a value that represents the name of the object defined by that tag. The value of the Name attribute can now be used in the code-behind as if it were an object reference declared in C#.

Click Attribute
We need to define counter delegate functions to be called when the buttons are clicked. The values of the Click attributes refer to delegates for the Click event for those Button tags which must be defined in the code-behind file.

The code-behind looks like this:

```
namespace WpfApplication12
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void dodecclick(object sender, RoutedEventArgs e)
        {
            int value = int.Parse(tb1.Text);
            value--;
            tb1.Text = value.ToString();
        }
        private void doincclick(object sender, RoutedEventArgs e)
        {
            int value = int.Parse(tb1.Text);
            value++;
            tb1.Text = value.ToString();
        }
    }
}
```

In the code-behind, the tags in the XAML can be referred to using their Name attribute value as if they had been instantiated in code. This brief example illustrates how the programmer can easily make a connection between the XAML and the code-behind.

Here's a side-by-side look at the XAML from above and an approximation of the equivalent C# if it had been implemented as a code-only app:

First the XAML:

```
<Window x:Class="WpfApplication12.MainWindow" …
        Title="MainWindow" Height="350" Width="525">
    <WrapPanel>
        <TextBox Name="tb1" Text="0" Width="50" Margin="5" Padding="5"> />
        <Button Name="incbtn" Click="doincclick" Margin="5" Padding="5">
            Inc
        </Button>
        <Button Name="decbtn" Click="dodecclick" Margin="5" Padding="5">
            Dec
        </Button>
    </WrapPanel>
</Window>
```

And the equivalent C#:

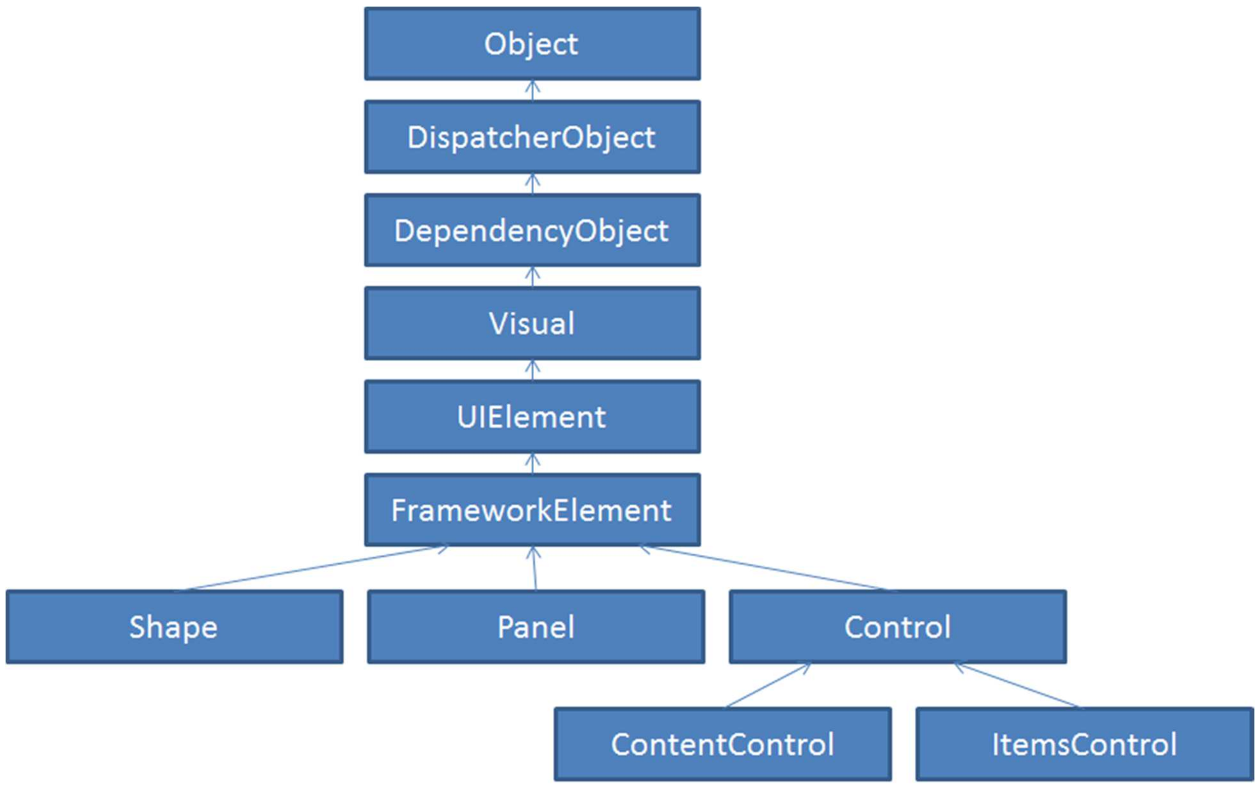
```
class MainWindow : Window {
        private TextBox tb1;
        private Button incbtn;
        private Button decbtn;
        public MainWindow() {
                WrapPanel p = new WrapPanel;
                this.Content = p;
                p.Parent = this;
                tb1 = new TextBox();
                tb1.Width = 50;
                tb1.Text = “0”;
                tb1.Margin = 5;
                tb1.Padding = 5;
                tb1.Parent = p;
                incbtn = new Button();
                incbtn.Content = “Inc”;
                incbtn.Margin = 5;
                incbtn.Padding = 5;
                incbtn.Click += doincclick;
                incbtn.Parent = p;
                decbtn = new Button();
                decbtn.Content = “Dec”;
                decbtn.Margin = 5;
                decbtn.Padding = 5;
                decbtn.Click += dodecclick;
        }
}
```

WPF Inheritance Hierarchy
As a WPF programmer researching how to best implement a given look or behavior, you will frequently find white papers, discussion forum and blog posts, and similar, providing example code, markup, or both. The example will often be written in terms of classes or tags that are not commonly used as explicit GUI elements, but which sit at important points in the inheritance hierarchy and which therefore represent the most general class possible for the example being coded.

A good MSDN paper is “WPF Architecture”
http://msdn.microsoft.com/en-us/library/ms750441.aspx

At the top of the inheritance hierarchy for WPF controls are the following classes:

- System.Object
  - The ultimate base class of all classes in the .NET Framework
- System.Windows.Threading.DispatcherObject
  - Defines the machinery for dealing with threads and concurrency
- System.Windows.DependencyObject
  - Defines the machinery for dealing with dependency properties (a useful expansion of .NET properties)
- System.Windows.Media.Visual
  - Supports building a tree of visual objects, the visual hierarchy, plus details on how to draw it to the screen
- System.Windows.UIElement
  - Layout, Input, Events
    - Layout proceeds in two phases: Measure + Arrange
  - Input
    - From signal on a kernel mode device driver to routed event (preview and actual)
- System.Windows.FrameworkElement
  - Adds features to UIElement
    - Layout system
    - Logical tree
    - Object lifetime events
    - Support for data binding and dynamic resource references
    - Styles
    - More animation support
  - Important derived classes
    - Shape
    - Panel
    - Control
- System.Windows.Controls.Control
  - Important Derived Control Classes
    - ContentControl: assign one item to its Content property
      - Label, Button, Window
    - ItemsControl: assign multiple items to its Items or ItemsSource property
      - ListBox, TreeView


Object
DispatcherObject
DependencyObject
Visual
UIElement
FrameworkElement
Shape
Panel
Control
ContentControl
ItemsControl