# Windows Presentation Foundation Using C#

**Windows Presentation Foundation Using C#**
**Rev. 4.8**

**Student Guide**



**Authors:** Robert J. Oberg and Ernani Junior Cecon

**Special Thanks:** Dana Wyatt



Object Innovations
877-558-7246
www.objectinnovations.com

Published in the United States of America.

# Table of Contents (Overview)

# Directory Structure

- **Install the course software by running the self-extractor *Install_WpfCs_48.exe*.**
- **The course software installs to the root directory *C:\OIC\WpfCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02** and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for performing in-class demonstrations led by the instructor.
- **Data files install to the directory *C:\OIC\Data*.**

# Table of Contents (Detailed)

# Chapter 1

# Introduction to WPF

# Introduction to WPF

## Objectives

***After completing this unit you will be able to:***

- **Discuss the rationale for WPF.**
- **Describe what WPF is and its position in the classic .NET Framework and in .NET Core.**
- **Give an overview of the main features of WPF.**
- **Describe the role of the fundamental Application and Window classes.**
- **Implement a "Hello, World" Windows application using WPF.**
- **Create, build and run simple WPF programs using Visual Studio 2017.**
- **Use simple brushes in your WPF programs.**
- **Use panels to lay out Windows that have multiple controls.**

# History of Microsoft GUI

- **WPF is an extremely sophisticated and complex technology for creating GUI programs.**
- **Why has Microsoft done this when Windows Forms and Web Forms in .NET are relatively new themselves?**
- **To understand, let's take a look back at various technologies Microsoft has employed over the years to support GUI application development:**
  - Windows 1.0 was the first GUI environment from Microsoft (ignoring OS/2, which is no longer relevant), provided as a layer on top of DOS, relying on the GDI and USER subsystems for graphics and user interface.
  - Windows has gone through many versions, but always using GDI and USER, which have been enhanced over the years.
  - DirectX was introduced in 1995 as a high-performance graphics system, targeting games and other graphics-intensive environments.
  - Windows Forms in .NET used a new enhanced graphics subsystem, GDI+.
  - DirectX has gone through various versions, with DirectX 9 providing a library to use with managed .NET code,

# Why WPF?

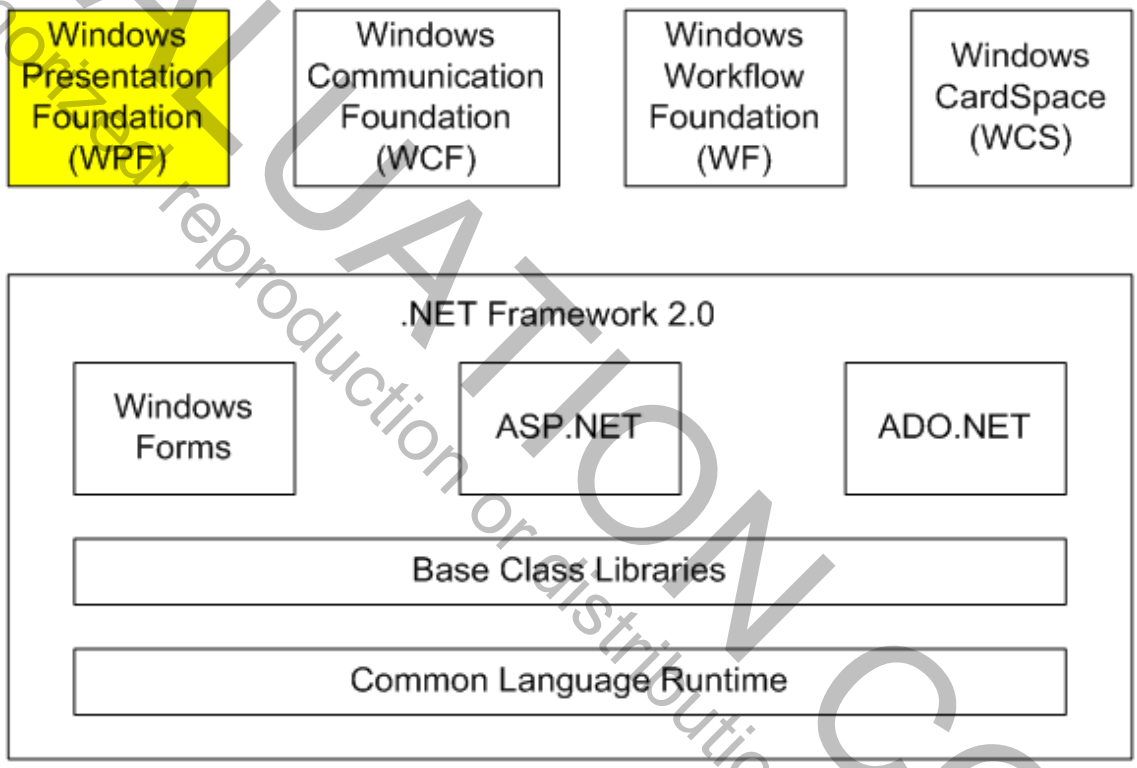
- **The various technologies support development of sophisticated graphics and GUI programs, but there are several different, complex technologies a programmer may need to know.**
- **The goal of Windows Presentation Foundation is to provide a unified framework for creating modern user experiences.**
  - It is built on top of .NET, providing all the productivity benefits of the large .NET class library.
- **Benefits of WPF include:**
  - Integration of 2D and 3D graphics, video, speech, and rich document viewing.
  - Resolution independence, spanning mobile devices and 50 inch televisions.
  - Easy use of hardware acceleration when available.
  - Declarative programming of objects in the WPF library through a new Extensible Application Markup Language, or XAML.
  - Easy deployment through Windows Installer, ClickOnce, or by hosting in a Web browser.

# When Should I Use WPF?

- **DirectX can still provide higher graphics performance and can exploit new hardware features before they are exposed through WPF.**
  - But DirectX is a low-level interface and *much* harder to use than WPF.
- **WPF is better than Windows Forms for applications with rich media, but what about business applications with less demanding graphics environments?**
  - Initially, WPF lacks some Windows Forms controls.
  - But future development at Microsoft will be focused on WPF rather than Windows Forms, so the long range answer is clearly to migrate to WPF development.
  - Visual Studio 2017 provides strong tool support for WPF.
- **Is WPF a replacement for Adobe "Flash" for Web applications with a rich user experience?**
  - Viewing rich WPF Web content requires Windows and .NET Framework 3.0 or higher.
  - Microsoft Silverlight, a small lightweight subset of the WPF runtime, does offer an alternative to Flash.

# WPF and .NET Framework 3.0

- **WPF originated as a component of a group of new .NET technologies, formerly called *WinFX* and later called .NET Framework 3.0.**
- **It layers on top of .NET Framework 2.0.**




- **WPF provides a unified programming model for creating rich user experiences incorporating UI, media and documents.**

# .NET Framework 4.0 and Higher

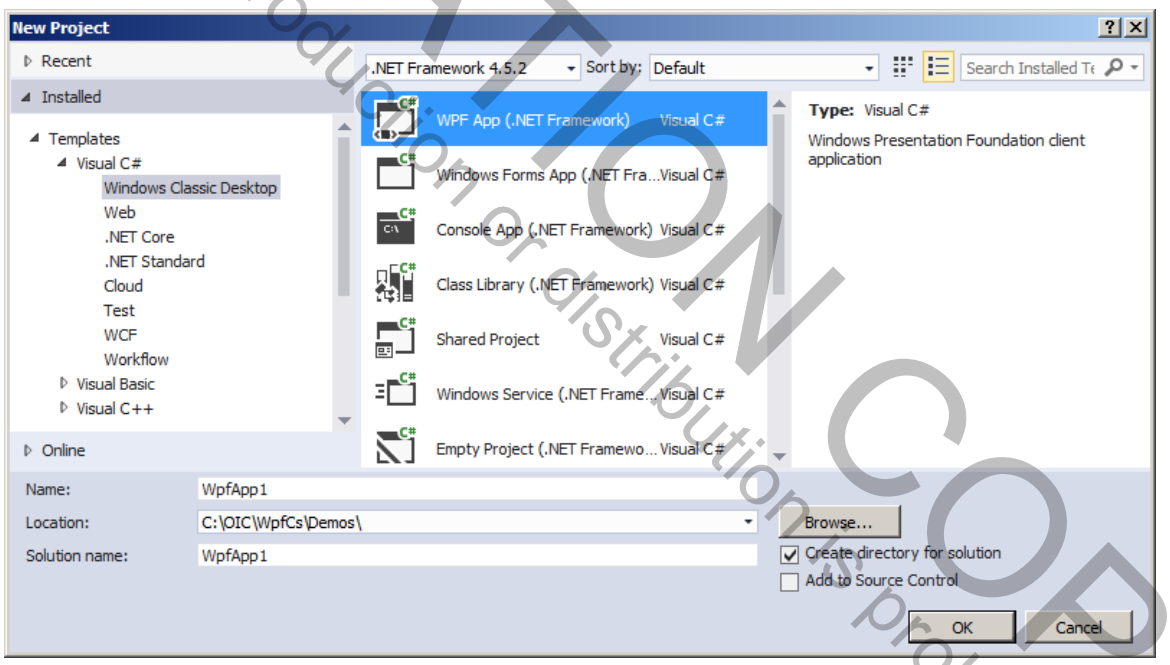
- **The .NET Framework 3.5 added a number of important features beyond those of .NET 3.0.**
  - Notable was integration with the tooling support provided by Visual Studio 2008.
  - Language Integrated Query (LINQ) extends query capabilities to the syntax of the C# and Visual Basic programming languages.
  - Enhancements to the C# programming language, largely to support LINQ.
  - Integration of ASP.NET AJAX into the .NET Framework.
- **.NET 3.5 still layered on top of the .NET 2.0 runtime.**
- **.NET 4.0 and higher provides a new runtime and many new features, such as:**
  - New controls and other enhancements to WPF.
  - New bindings, simplified configuration and other enhancements to WCF.
  - A dynamic language runtime supporting dynamic languages such as IronRuby and IronPython.
  - ASP.NET MVC 6 for Web development.
  - A new programming model for parallel programming.
  - And much more!

# Visual Studio 2017

- **Visual Studio 2017 provides effective tooling support for the .NET Framework.**
  - Early support for WinFX involved add-ons to Visual Studio, but now there is a fully integrated environment.
  - .NET 4.7 is not installed with Visual Studio 2017 but is a separate download. We do not use .NET 4.7 in this course.
  - In fact, the default Framework is .NET 4.5.2.
- **Visual Studio 2017 has an IDE with an attractive graphical appearance, implemented using WPF.**
- **Features in Visual Studio 2017 include:**
  - Easy navigation and docking.
  - Automatic settings migration from earlier versions of Visual
  - Multi-targeting to .NET 2.0, .NET 3.0, .NET 3.5, .NET 4.0, .NET 4.5.x, .NET 4.6.x and .NET 4.7 (if installed).
- **There are many project templates, including:**
  - WPF projects
  - WCF projects
  - WF projects
- **There are a number of designers, including WPF Designer, an object/relational designer, and a workflow designer.**

# Visual Studio Community 2017

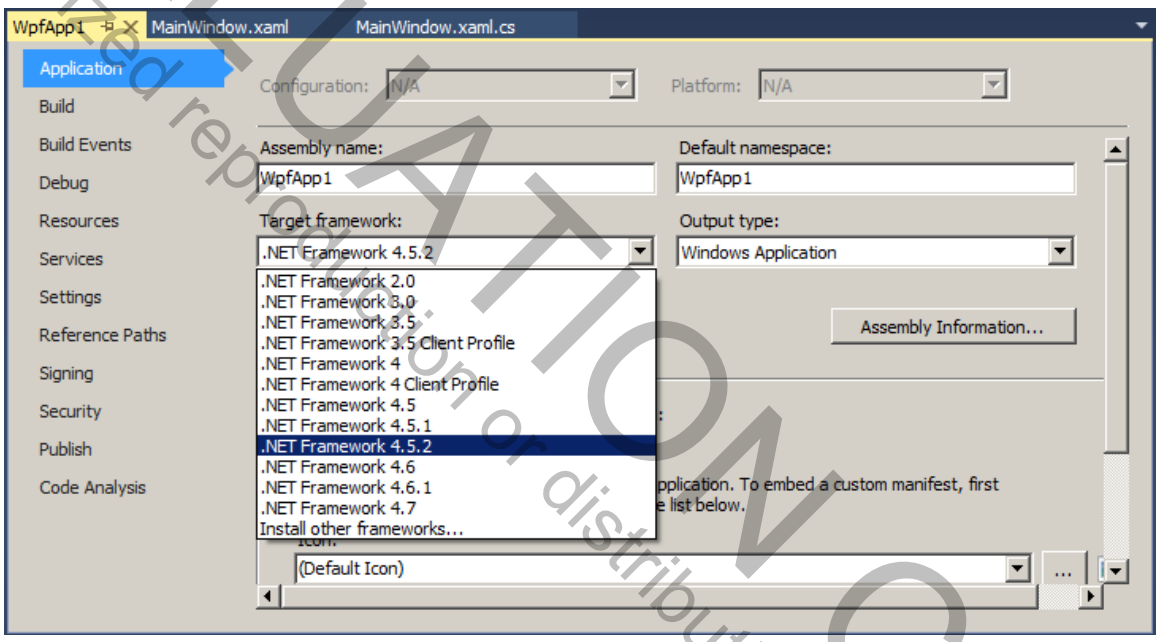
- **A noteworthy aspect of Visual Studio 2017 is a strong free Communnity version of the tool.**
- **In this course we will rely on Visual Studio Community 2017.**
  - It supports multiple language development (C#, Visual Basic, and C++).
  - It supports the creation of WPF projects.
  - It also supports unit testing.




- **Moreover, the Community edition provides features present in higher editions, such as support of WCF and Workflow projects.**

# Target Framework

- **You can specify the version of .NET Framework that your application targets both at the time the project is created and later by bringing up the properties for your project.**
  - Right-click over the project in Solution Explorer and choose Properties.




- **Many example programs in this course were originally targeted for an earlier version of the .NET Framework, but will run fine under .NET 4.5.2 or higher.**

# WPF and .NET Core

- **.NET Core is a smaller, modular version of the .NET Framework that is portable across multiple platforms.**
- **Rather than one large assembly, .NET Core is released through NuGet in smaller feature-specific assembly packages.**
- **.NET Core provides key functionality used in applications regardless of platform.**
  - This common functionality provides for shared code that can be used across platforms.
  - Your application then links in additional platform-specific code.
- **Microsoft platforms you can target include traditional desktop Windows and Windows phones.**
  - Other platforms include Mac and several versions of Linux.
- **The original .NET Framework targeting only Window is referred to as the "classic .NET Framework."**
- **However, WPF is *not* supported by .NET Core.**
  - WPF is built on top of DirectX and thus targets *only* Windows platforms. WPF is part of the classic .NET Framework.

# WPF Core Types and Infrastructures

- **A great many classes in WPF inherit from one of four different classes:**
  - UIElement
  - FrameworkElement
  - ContentElement
  - FrameworkContentElement
- **These classes, often called *base element classes*, provide the foundation for a model of composing user interfaces.**
- **WPF user interfaces are composed of elements that are assembled in a *tree hierarchy*, known as an *element tree*.**
- **The element tree is both an intuitive way to lay out user interfaces and a structure over which you can layer powerful UI services.**
  - The **dependency property system** enables one element to implement a property that is automatically shared by elements lower in the element tree hierarchy.
  - **Routed events** can route events along the element tree, affording event handlers all along the traversed path to handle the event.

# XAML

- **Extensible Application Markup Language (XAML, pronounced "zammel") provides a declarative way to define user interfaces.**
- **Here is the XAML definition of a simple button.**

```
<Button
  FontSize="16"
  HorizontalAlignment="Center"
  VerticalAlignment="Center"
  >
  Say Hello
</Button>
```

- **To see this button displayed, we'll need some more program elements, which we'll discuss later.**
- **XAML has many advantages, and we'll study it beginning in the next chapter.**
  - Using XAML facilitates separating front-end appearance from back-end logic.
  - XAML is the most concise way to represent user interfaces.
  - XAML is defined to work well with tools.

# Controls

- **WPF comes with many useful controls, and more should come as the framework evolves:**
  - Editing controls such as TextBox, CheckBox, RadioButton.
  - List controls such as ListBox, ListView, TreeView.
  - User information such as Label, ProgressBar, ToolTip.
  - Action such as Button, Menu and ToolBar.
  - Appearance such as Border, Image and Viewbox.
  - Common dialog boxes such as OpenFileDialog and PrintDialog.
  - Containers such as GroupBox, ScrollBar and TabControl.
  - Layout such as StackPanel, DockPanel and Grid.
  - Navigation such as Frame and Hyperlink.
  - Documents such as DocumentViewer.
  - WPF 4.5 includes a new Ribbon control that can be used to customize the UI for Microsoft Office applications.
- **The appearance of controls can be customized without programming with styles and templates.**
- **If necessary, you can create a custom control by deriving a new class from an appropriate base class.**

# Data Binding

- **WPF applications can work with many different kinds of data:**
  - Simple objects
  - Collection objects
  - WPF elements
  - ADO.NET data objects
  - XML objects
  - Objects returned from Web services
- **WPF provides a data binding mechanism that binds these different kinds of data to user interface elements in your application.**
  - Data binding can be implemented both in code and also declaratively using XAML.
  - Visual Studio 2017 provides drag and drop data binding for WPF.

# Appearance

- **WPF provides extensive facilities for customizing the appearance of your application.**
- **UI *resources* allow you to define objects and values once, for things like fonts, background colors, and so on, and reuse them many times.**
- ***Styles* enable a UI designer to standardize on a particular look for a whole product.**
- ***Control templates* enable you to replace the default appearance of a control while retaining its default behavior.**
- **With *data templates*, you can control the default visualization of bound data.**
- **With *themes*, you can enable your application to respect visual styles from the operating system.**

# Layout and Panels

- ***Layout* is the proper sizing and positioning of controls as part of the process of composing the presentation for the user.**
- **The WPF layout system both simplifies the layout process through useful classes and provides adaptability of the UI appearance in the face of changes:**
  - Window resizing
  - Screen resolution and dots per inch
- **The layout infrastructure is provided by a number of classes:**
  - StackPanel
  - DockPanel
  - WrapPanel
  - Grid
  - Canvas
- **The flexible layout system of WPF facilitates globalization of user interfaces.**

# Graphics

- **WPF provides an improved graphics system.**
- ***Resolution and device-independent graphics*: WPF uses device-independent units, enabling resolution and device independence.**
  - Each pixel, which is device-independent, automatically scales with the dots-per-inch setting of your system.
- ***Improved precision*: WPF uses *double* rather than *float* and provides support for a wider array of colors.**
- ***Advanced graphics and animation support.***
  - You can use animation to make controls and elements grow, spin, and fade, and so on. You create interesting page transitions, and other special effects.
- ***Hardware acceleration*: The WPF graphics engine is designed to take advantage of graphics hardware where available.**

# Media

- **WPF provides rich support for media, including images, video and audio.**
- **WPF enables you to work with images in a variety of ways. Images include:**
  - Icons
  - Backgrounds
  - Parts of animations
- **WPF provides native support for both video and audio.**
  - The **MediaElement** control makes it easy to play both video and audio.

# Documents and Printing

- **WPF provides improved support in working with text and typography.**
- **WPF includes support for three different types of documents:**
  - **Fixed documents** support a precise WYSIWYG presentation.
  - **Flow documents** dynamically adjust and reflow their content based on run-time variables like window size and device resolution.
  - **XPS documents** (XPS Paper Specification) is a paginated representation of electronic paper described in an XML-based format. XPS is an open and cross-platform document format.
- **WPF provides better control over the print system, including remote printing and queues.**
  - XPS documents can be printed directly without conversion into a print format such as Enhanced Metafile (EMF), Printer Control Language (PCL) or PostScript.
- **WPF provides a framework for annotations, including "Sticky Notes."**

# Plan of Course

- **As you can see, Windows Presentation Foundation is a large, complex technology.**
- **In a short course such as this one, the most we can do is to provide you with an effective orientation to this large landscape.**
- **We provide a step-by-step elaboration of the most fundamental features of WPF and many small, complete example programs.**
- **We follow this sequence:**
  - In the rest of this chapter we introduce you to several, small "Hello, World" sample WPF applications.
  - The second chapter introduces XAML.
  - The third chapter covers a number of simple WPF controls.
  - We discuss layout in more detail.
  - We then cover common user interface features in Windows programming, including dialogs, menus and toolbars.
  - Resources and dependency properties are discussed.
  - The course concludes with chapters on data binding and styles and interop with Windows Forms.

# Application and Window

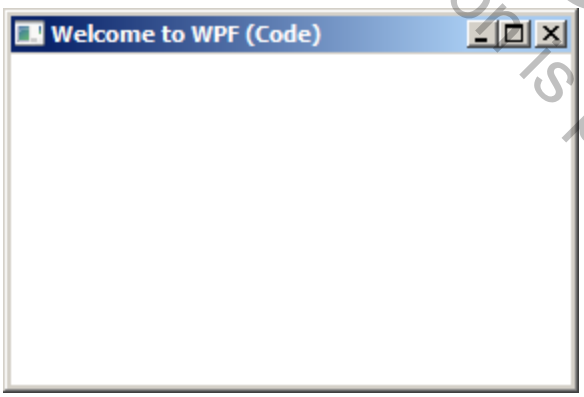
- **The two most fundamental classes in WPF are *Application* and *Window*.**
  - A WPF application usually starts out by creates objects of type **Application** and **Window**.
  - For an example, see the file **Program.cs** in the folder **FirstWpf\Step1** in the chapter directory for Chapter 1.

```
using System;
using System.Windows;

namespace FirstWpf
{
   public class MainWindow : Window
   {
      [STAThread]
      static void Main(string[] args)
      {
         Application app = new Application();
         app.Run(new MainWindow());
      }
      public MainWindow()
      {
         Title = "Welcome to WPF (Code)";
         Width = 288;
         Height = 192;
      }
   }
}
```

- **A program can create only one Application object, which is invisible. A Window object is visible, corresponding to a real window.**

# FirstWpf Example Program

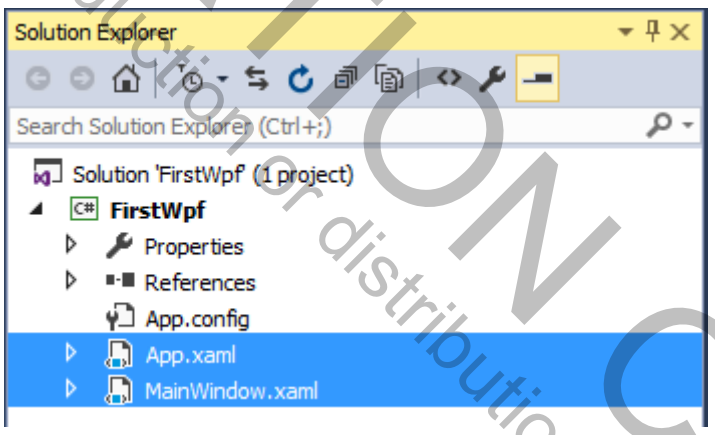
- **Our example program has the following features:**
  - Import the **System.Windows** namespace. This namespace includes the fundamental WPF classes, interfaces, delegates, and so on, including the classes Application and Window.
  - Make your class derive from the **Window** class.
  - Provide the attribute **[STAThread]** in front of the **Main()** method. This is required in WPF and ensures interoperability with COM.
  - In **Main()**, instantiate an **Application** object and call the **Run()** method.
  - In the call to **Run()** pass a new instance of your Window-derived class.
  - In the constructor of your Window-derived class, specify any desired properties of your Window object. We set the **Title**, **Width** and **Height**.
- **Build and run. You'll see:**



# Demo – Using Visual Studio 2017

- **Although you can compile WPF programs at the command-line, for simplicity we will use Visual Studio 2017 throughout this course.**
  - To make clear all the details in creating a WPF application, we'll create our sample program from scratch in the **Demos** directory.

1. Use the New Project dialog (File | New Project) to create a new WPF Application called **FirstWpf** in the **Demos** directory.
2. In Solution Explorer, delete the files **App.xaml** and **MainWindow.xaml**.




3. Add a new code file **Program.cs** to your project.
4. Enter the code shown two pages back. If you like, to save typing, you may copy/paste from the **FirstWpf\Step1** folder.
5. Build and run. You are now at Step 1. That's all there is to creating a simple WPF program using Visual Studio 2017!

# Creating a Button

6. Continuing the demo, let's add a button to our main window. Begin with the following code addition.

```
public HelloWorld()
{
    Title = "First WPF C# Program";
    Width = 288;
    Height = 192;

    Button btn = new Button();
    btn.Content = "Say Hello";
    btn.FontSize = 16;

    Content = btn;
}
```

7. Build the project. You'll get a compile error, because you need an additional namespace, **System.Windows.Controls**.

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

8. Build and run. You'll se the button fills the whole client area of the main window.

9. Add the following code to specify the horizontal and vertical alignment of the button.

```
btn.HorizontalAlignment =
   HorizontalAlignment.Center;
btn.VerticalAlignment = VerticalAlignment.Center;
```

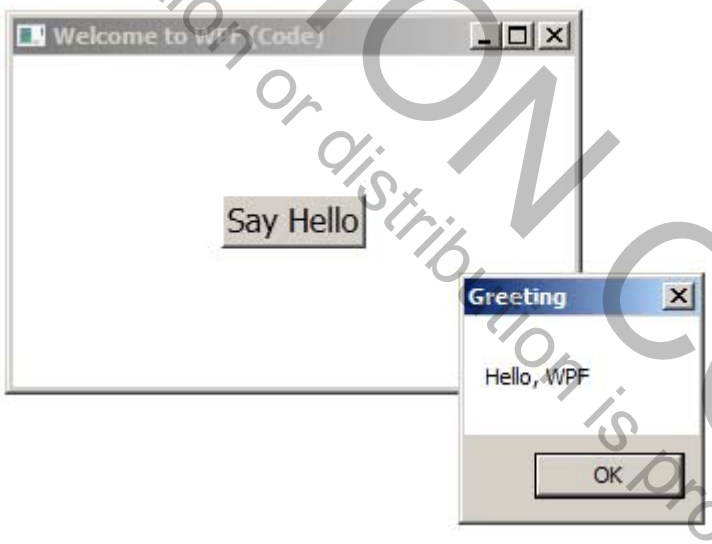
10. Build and run. Now the button will be properly displayed, sized just large enough to contain the button's text in the designated font.

# Providing an Event Handler

11. Continuing the demo, add the following code to specify an event handler for clicking the button.

```
    btn.Click += ButtonOnClick;

    Content = btn;
}

void ButtonOnClick(object sender, RoutedEventArgs
args)
{
    MessageBox.Show("Hello, WPF", "Greeting");
}
```

12. Build and run. You will now see a message box displayed when you click the “Say Hello” button



# Specifying Initial Input Focus

13. You can specify the initial input focus by calling the **Focus()** method of the **Button** class (inherited from the **UIElement** class).

```
btn.Focus();
```

14. Build and run. The button will now have the initial input focus, and hitting the Enter key will invoke the button's Click event handler. You are now at Step 2.

- **Note that specifying the focus programmatically in this manner is deprecated, because it violates accessibility guidelines.**
  - When run for the visually impaired, setting the focus will cause the text of the button to be read out.

# Complete First Program

- **See *FirstWpf\Step2*.**

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

namespace FirstWpf
{
   public class MainWindow : Window
   {
      [STAThread]
      static void Main(string[] args)
      {
         Application app = new Application();
         app.Run(new MainWindow());
      }
      public MainWindow()
      {
         Title = "Welcome to WPF (Code)";
         Width = 288;
         Height = 192;

         Button btn = new Button();
         btn.Content = "Say Hello";
         btn.FontSize = 16;
         btn.HorizontalAlignment =
            HorizontalAlignment.Center;
         btn.VerticalAlignment =
            VerticalAlignment.Center;

         btn.Click += ButtonOnClick;
         // Setting focus is deprecated for
         // violating accessibility guidelines
         btn.Focus();

         Content = btn;
      }
```

# Complete First Program (Cont'd)

```
        void ButtonOnClick(object sender,
        RoutedEventArgs args)
        {
           MessageBox.Show("Hello, WPF",
              "Greeting");
        }
     }
  }
```

# Device-Independent Pixels

- **The *Width* and *Height* properties for the main window are specified in *device-independent pixels* (or units).**
  - Each such unit is 1/96 inch.
  - Values of 288 and 192 thus represent a window that is 3 inches by 2 inches.
- **If you get a new monitor with a much higher resolution, the window will still be displayed with a size of 3 inches by 2 inches.**
- **Note that this mapping to inches assumes that your monitor is set to its "natural" resolution.**
  - Any differences will be reflected in a different physical size.

# Class Hierarchy

- **The key classes *Application*, *Window* and *Button* all derive from the abstract class *DispatcherObject*.**

Object
    DispatcherObject (abstract)
        Application
        DependencyObject
            Visual (abstract)
                UIElement
                FrameworkElement
                Control
                ContentControl
                Window
                ButtonBase
                Button

# Content Property

- **The key property of *Window* is *Content*.**
  - The **Content** property also applies to all controls that derive from **ContentControl**, including **Button**.
- **You can set *Content* to any *one* object.**
  - This object can be anything, such as a string, a bitmap, or any control.
  - In our example program, we set the Content of the main window to the Button that we created.

```
Button btn = new Button();
...

Content = btn;
```

- **We will see a little later how we can overcome the limitation of one object to create a window that has multiple controls in it.**

# Simple Brushes

- **You may specify a foreground or background of a window or control by means of a *Brush*.**
  - We will look at the simplest brush class, **SolidColorBrush**.
- **You can specify a color for a SolidColorBrush in a couple of ways:**
  - By using the **Colors** enumeration.
  - By using the **FromRgb()** method of the **Color** class.
- **The program *SimpleBrush* illustrates setting foreground and background properties.**

```
public SimpleBrush()
{
   Title = "Simple Brushes";
   Width = 288;
   Height = 192;
   Background = new SolidColorBrush(Colors.Beige);

   Button btn = new Button();
   ...
   btn.Background = new SolidColorBrush(
      Color.FromRgb(0, 255, 0));
   btn.Foreground = new SolidColorBrush(
      Color.FromRgb(0, 0, 255));
   Content = btn;
}
```

# Panels

- **As we have seen, the *Content* of a window can be set only to a *single* object.**
- **What do we do if we want to place multiple controls on a window?**
- **We use a *Panel*, which is a single object and can have multiple children.**
- **Panel is an abstract class deriving from *FrameworkElement*. There are several concrete classes representing different types of panels.**

UIElement
    FrameworkElement
        Panel (abstract)
            Canvas
            DockPanel
            Grid
            StackPanel
            UniformGrid
            WrapPanel

- **Rather than specify precise size and location of controls in a window, WPF prefers *dynamic layout*.**
  - The panels are responsible for sizing and positioning elements.
  - The various classes deriving from **Panel** each support a particular kind of layout model.

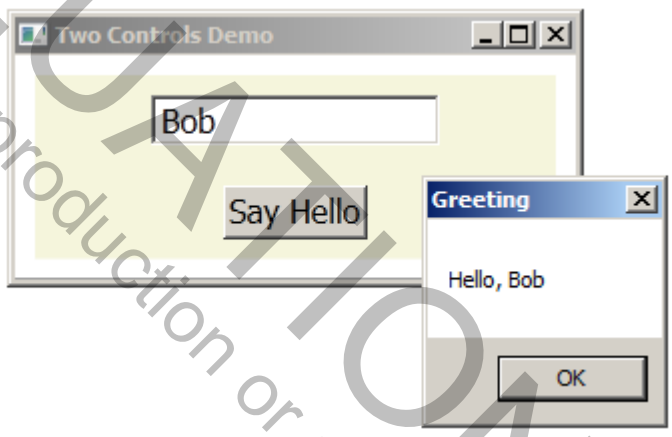
# Children of Panels

- ***Panel* has a property *Children* that is used to store child elements.**
  - **Children** is an object of type **UIElementCollection**.
  - **UIElementCollection** is a collection of UIElement objects.
- **There is a great variety of elements that can be stored in a panel, including any kind of control.**
- **You can add a child element to a panel via the *Add()* method of *UIElementCollection*.**

```
StackPanel panel = new StackPanel();
...

Button btnGreet = new Button();
...

panel.Children.Add(btnGreet);
```

# Example – TwoControls

- **The example program *TwoControls* illustrates use of a *StackPanel*, whose children are a TextBox and a Button.**
  - See Step2.
  - We provide a beige brush for the panel to help us see the extent of the panel in the window.




  - The program also illustrates various automatic sizing features of WPF.

# TwoControls – Code

- **The *TwoControls* class derives from Window in the usual manner.**
- **A private member *txtName* is defined in the class, because we need to reference the TextBox in both the constructor and in the event handler.**

```
class TwoControls : Window
{
   [STAThread]
   static void Main(string[] args)
   {
      Application app = new Application();
      app.Run(new TwoControls());
   }

   private TextBox txtName;

   public TwoControls()
   {
      Title = "Two Controls Demo";
      Width = 288;
      const int MARGINSIZE = 10;
```

- **A *StackPanel* is created and the *Content* of the main window is set to this new StackPanel.**

```
      StackPanel panel = new StackPanel();
      Content = panel;
```

# Automatic Sizing

- **Only the width of the main window is specified.**
- **The height of the main window is sized to its content, which is a panel containing two controls.**

```
public TwoControls()
{
   Title = "Two Controls Demo";
   Width = 288;
   const int MARGINSIZE = 10;

   StackPanel panel = new StackPanel();
   Content = panel;

   SizeToContent = SizeToContent.Height;

   panel.Background = Brushes.Beige;
   panel.Margin = new Thickness(MARGINSIZE);
```

  – Note that we are specifying a brush for the panel, and we are specifying a margin of 10 device-independent pixels.

- **The TextBox specifies its width and horizontal alignment, and also a margin.**

```
   txtName = new TextBox();
   txtName.FontSize = 16;
   txtName.HorizontalAlignment =
      HorizontalAlignment.Center;
   txtName.Margin = new Thickness(MARGINSIZE);
   txtName.Width = Width / 2;
   panel.Children.Add(txtName);
```

# TwoControls – Code (Cont'd)

- **The Button also specifies its horizontal alignment and a margin.**

```
Button btnGreet = new Button();
btnGreet.Content = "Say Hello";
btnGreet.FontSize = 16;
btnGreet.Margin = new Thickness(MARGINSIZE);
btnGreet.HorizontalAlignment =
   HorizontalAlignment.Center;
btnGreet.Click += ButtonOnClick;
panel.Children.Add(btnGreet);
```

- **Both the TextBox and the Button are added as children to the panel.**

```
txtName = new TextBox();
...
panel.Children.Add(txtName);

Button btnGreet = new Button();
...
panel.Children.Add(btnGreet);
```

- **The Click event of the Button is handled.**

```
   btnGreet.Click += ButtonOnClick;
   panel.Children.Add(btnGreet);
}
void ButtonOnClick(object sender,
RoutedEventArgs args)
{
   MessageBox.Show("Hello, " + txtName.Text,
      "Greeting");
}
```

# Lab 1

## A Windows Application with Two Controls

In this lab you will implement the **TwoControls** example program from scratch. This example will illustrate in detail the steps needed to create a new WPF application using Visual Studio, and you will get practice with all the fundamental concepts of WPF that we've covered in this chapter.

Detailed instructions are contained in the Lab 1 write-up at the end of the chapter.

Suggested time: 30 minutes

# Summary

- **The goal of Windows Presentation Framework is to provide a unified framework for creating modern user experiences.**
- **WPF is a major component of the classic .NET Framework.**
  - In .NET 3.0/3.5, it is layered on top of .NET Framework 2.0.
  - In .NET 4.0 and higher there is a new 4.0 runtime.
- **However, since WPF relies on Windows DirectX graphics technology, WPF is not supported on the portable .NET Core.**
- **The most fundamental WPF classes are *Application* and *Window*.**
- **You can create, build and run simple WPF programs using Visual Studio.**
- **You may specify a foreground or background of a window or control by means of a *Brush*.**
- **You can use panels to lay out Windows that have multiple controls.**

# Lab 1

## A Windows Application with Two Controls

### Introduction

In this lab you will implement the **TwoControls** example program from scratch. This example will illustrate in detail the steps needed to create a new WPF application using Visual Studio, and you will get practice with all the fundamental concepts of WPF that we've covered in this chapter.

**Suggested Time:** 30 minutes

**Root Directory:** **OIC\WpfCs**

| | | |
|---|---|---|
| **Directories:** | **Labs\Lab1** | (do your work here) |
| | **Chap01\TwoControls\Step1** | (answer to Part 1) |
| | **Chap01\TwoControls\Step2** | (answer to Part 2) |

### Part 1. Create a WPF Application with a StackPanel

In Part 1 you will use Visual Studio to create a WPF application. You will go on to create a StackPanel that has as children a TextBox and a Button. This first version does not provide an event handler for the button. Also, it does not handle sizing very well!

1. Use Visual Studio to create a new WPF application **TwoControls** in the Lab1 folder.

2. In Solution Explorer, delete the files **App.xaml** and **MainWindow.xaml**.

3. Add a new code file **Program.cs** to your project.

4. In **Program.cs** enter the following code, which does the minimum of creating Application and Window objects.

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

namespace TwoControls
{
   class TwoControls : Window
   {
      [STAThread]
      static void Main(string[] args)
      {
         Application app = new Application();
         app.Run(new TwoControls());
      }
```

```
        public TwoControls()
        {
        }
    }
}
```

5. Build and run. You should get a clean compile. You should see a main window, which has no title and an empty client area.

6. Add the following code to the **TwoControls** constructor.

```
        public TwoControls()
        {
            Title = "Two Controls Demo";
            Width = 288;
        }
```

7. Build and run. Now you should see a title and the width as specified.

8. Now we are going to set the Content of the main window to a new StackPanel that we create. To be able to visually see the StackPanel, we will paint the background with a beige brush, and we'll make the Margin of the StackPanel 10 device-independent pixels.

```
        public TwoControls()
        {
            Title = "Two Controls Demo";
            Width = 288;
            const int MARGINSIZE = 10;

            StackPanel panel = new StackPanel();
            Content = panel;

            panel.Background = Brushes.Beige;
            panel.Margin = new Thickness(MARGINSIZE);
        }
```

9. Build. You'll get a compiler error because you need a new namespace for the **Brushes** class.

10. Bring in the **System.Windows.Media** namespace. Now you should get a clean build. Run your application. You should see the StackPanel displayed as solid beige, with a small margin.

11. Next we will add a TextBox as a child of the panel. Since we will be referencing the TextBox in an event-handler method as well as the constructor, define a private data member **txtName** of type **TextBox**.

```
        private TextBox txtName;
```

12. Provide the following code to initialize **txtName** and add it as a child to the panel.

```
            txtName = new TextBox();
```

```
txtName.FontSize = 16;
txtName.HorizontalAlignment = HorizontalAlignment.Center;
txtName.Width = Width / 2;
panel.Children.Add(txtName);
```

13. Build and run. Now you should see the TextBox displayed, centered, at the top of the panel.

14. Next, add code to initialize a Button and add it as a child to the panel.

```
Button btnGreet = new Button();
btnGreet.Content = "Say Hello";
btnGreet.FontSize = 16;
btnGreet.HorizontalAlignment = HorizontalAlignment.Center;
panel.Children.Add(btnGreet);
```

15. Build and run. You should now see the two controls in the panel. You are now at Step1.

**Part 2. Event Handling and Layout**

In Part 2 you will handle the Click event of the button. You will also provide better layout of the two controls.

1. First, we'll handle the Click event for the button. Provide this code to add a handler for the Click event.

```
btnGreet.Click += ButtonOnClick;
```

2. Provide this code for the handler, displaying a greeting to the person whose name is entered in the text box.

```
void ButtonOnClick(object sender, RoutedEventArgs args)
{
   MessageBox.Show("Hello, " + txtName.Text, "Greeting");
}
```

3. Build and run. The program now has its functionality, but the layout needs improving.

4. Provide the following code to size the height of the window to the size of its content.

```
SizeToContent = SizeToContent.Height;
```

5. Build and run. Now the vertical sizing of the window is better, but the controls are jammed up against each other.

6. To achieve a more attractive layout, provide the following statements to specify a margin around the text box and the button. You have a reasonable layout (Step2).

```
txtName.Margin = new Thickness(MARGINSIZE);
...
btnGreet.Margin = new Thickness(MARGINSIZE);
```

# Chapter 2

# XAML

# XAML

## Objectives

***After completing this unit you will be able to:***

- **Describe Extensible Application Markup Language (XAML) and its role in WPF in the classic .NET Framework.**
- **Explain the structure of XAML documents.**
- **Describe the XML namespaces you must use in your XAML documents.**
- **Use Visual Studio 2017 to create and edit XAML documents.**
- **Provide access key support in XAML.**
- **Explain the use of the content property in XAML.**
- **Use property elements to set values for complex properties in XAML documents.**
- **Explain the use of type converters in XAML.**

# What Is XAML?

- **XAML stands for Extensible Application Markup Language.**
- **XAML is a general-purpose declarative programming language that can be used to construct and initialize .NET objects.**
  - XAML is based on XML.
  - This piece of XAML will construct and initialize a **Button**.

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center" FontSize="16"
        Click="Button_Click">
  Say Hello
</Button>
```

- **You will recognize this fragment as XML:**
  - A start tag for the element **Button**
  - Four attributes, with the attribute values enclosed in quote marks (either double or single quote is OK)
  - Element content, in this case character data
  - An end tag

# Default Namespace

- **To ensure that the <*Button*> element gets mapped to the .NET *Button* class, we must ensure that the XML is in a suitable *namespace*.**
  - The namespace should be applied to the root element of the XML document.

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

    <Button HorizontalAlignment="Center"
            VerticalAlignment="Center"
            FontSize="16"
            Click="Button_Click">
        Say Hello
    </Button>
```

  - The attribute **xmlns** declares a default XML namespace, which applies to the element in which the namespace declaration appears as well as all child elements.
  - The URL **"http://schemas.microsoft.com/winfx/2006/xaml/presentation"** does not correspond to anything on the Web. Rather, WPF itself maps this XML namespace to common .NET namespaces, including **System.Windows**, **System.Windows.Controls**, and so on.
  - WPF will then recognize **Button** as a class and **FontSize**, **HorizontalAlignment** and so forth as properties of this class.

# XAML Language Namespace

- **A second namespace is the XAML language namespace, which defines special directives for the XAML compiler or parser.**
  - Its URL is **http://schemas.microsoft.com/winfx/2006/xaml**.
  - Since there is already a default namespace, it requires a prefix, which by convention is "x".

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

- **An example of a XAML directive is *x:Class*, which is used to specify a .NET class that is used in a code-behind file.**

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

# .NET Class and Namespace

- In this example, the .NET class is **MainWindow**, in the .NET namespace **OneButton**.

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

# Elements and Attributes

- **XAML specifies a mapping between XML namespaces, elements and attributes and .NET namespaces, types, properties and events.**
  - The declaration of an XML element (an **object element**) in XAML is equivalent to instantiating the corresponding .NET object.
  - Setting an attribute (called a **property attribute**) of such an element is equivalent to setting a property of the same name on the corresponding .NET object.
- **In our example we have the following correspondences:**

| | XML | .NET |
|---|---|---|
| Namespace | http://schemas.microsoft.com/winfx/2006/xaml/presentation | System<br>System.Windows<br>etc. |
| Element/ Type | <Button> | Button |
| Attribute/ Property | FontSize<br>HorizontalAlignment<br>etc. | FontSize<br>HorizontalAlignment<br>etc. |

# XAML in Visual Studio

- **Visual Studio will create projects for you that use XAML for creating WPF objects used in your program.**
  - The **Application** object is created in **App.xaml**.

```
<Application x:Class="WpfApplication1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/
       xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/
       2006/xaml"
    xmlns:local="clr-namespace:WpfApplication1"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

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

  - The **Window** object is created in **MainWindow.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="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

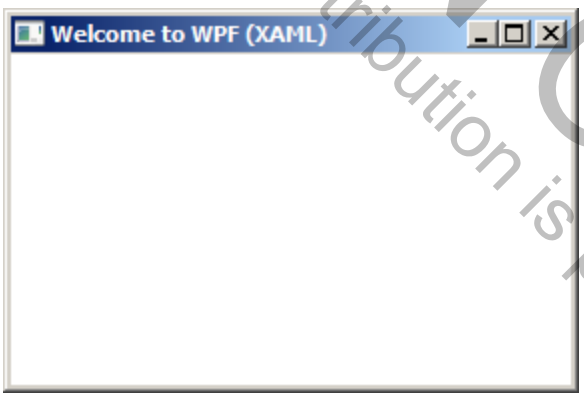
# Demo: One Button via XAML

1. Use Visual Studio to create a new WPF App called **OneButton** in the **Demos** folder.

2. Edit the file **MainWindow.xaml** to specify a title and the same height and width as in our procedural code version.

```
<Window x:Class="OneButton.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/
       xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/
       2006/xaml"
    ...
    Title="Welcome to WPF (XAML)" Height="192"
    Width="288">
    <Grid>

    </Grid>
</Window>
```

3. Build and run. You should see the new title displayed, and the window will be the dimension you specified.

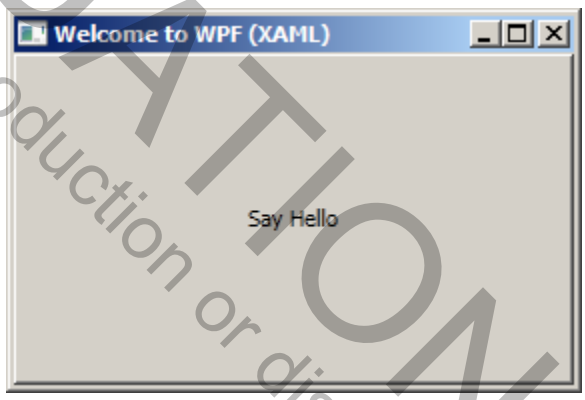


# One Button Demo (Cont'd)

4. Now let's edit **MainWindow.xaml** to specify a button control in place of a grid.

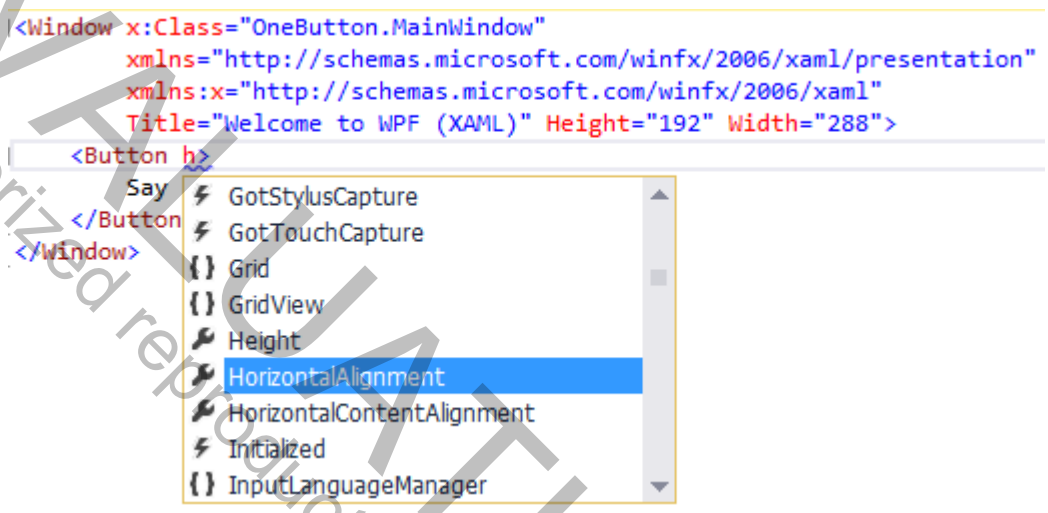
```
<Window x:Class="OneButton.MainWindow"
    ...
    <Button>
        Say Hello
    </Button>
</Window>
```

5. Build and run. The button fills the whole window!



# One Button Demo (Cont'd)

6. Edit the XAML file to specify some attributes, beginning with HorizontalAlignment, which we'll make Center (the default is Stretch). IntelliSense makes it easy.



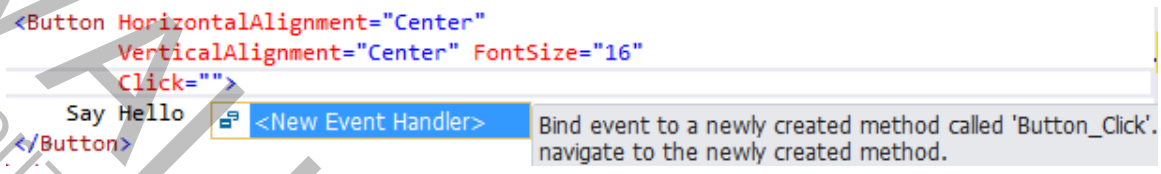

7. Also set VerticalAlignment to Center, and specify a larger font size.

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center" FontSize="16">
    Say Hello
</Button>
```

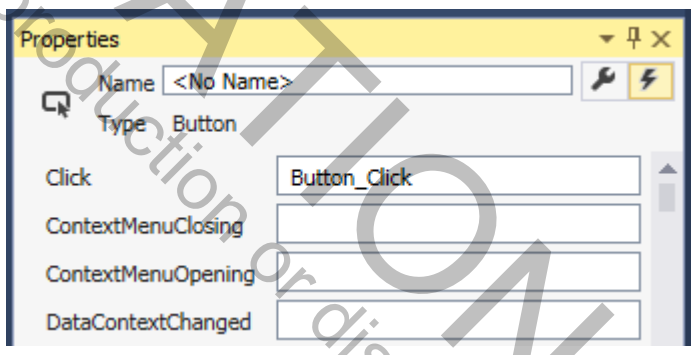
8. Build and run. We now see a nice centered button!

# Adding an Event Handler

- **There are three ways to add an event handler.**
  - Edit the XAML. Type in the event, right-click, and select Navigate to Event Handler. Or use IntelliSense.




  - Use the Events tab of the Properties window. Type in a name for a handler of the event you wish handled.




  - In Design view double-click the control to add a handler for the control's primary event (which is Click for a button).

9. Here is the final XAML for the button:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center" FontSize="16"
        Click="Button_Click">
    Say Hello
</Button>
```

# One Button Demo (Cont'd)

10. Provide the following code for the event handler, which will display a message box.

```
private void Button_Click(object sender,
RoutedEventArgs e)
{
   MessageBox.Show("Hello, WPF", "Greeting");
}
```

11. Build and run. Final project is in **Chap02\OneButton**.

# Layout in WPF

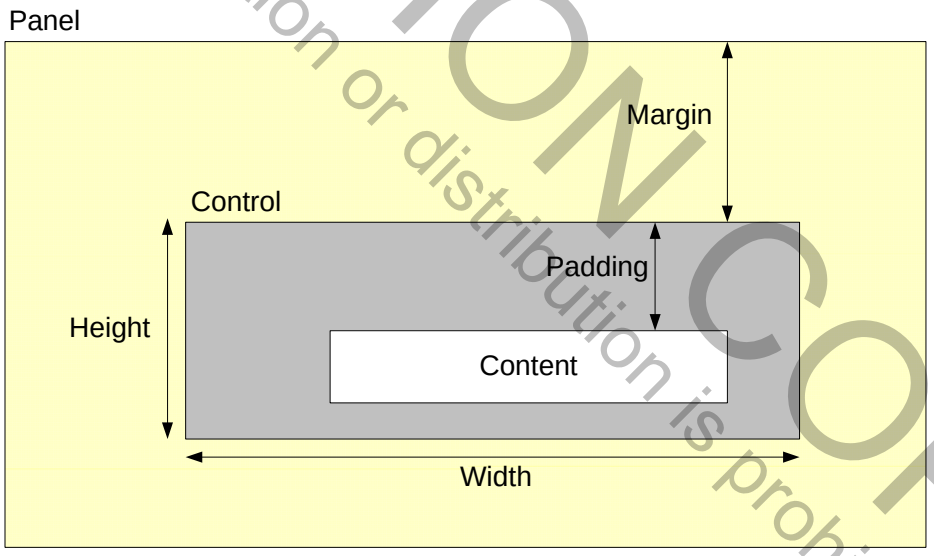
- **Ever since Visual Basic 1, it has been easy to lay out controls on windows.**
- **Traditional Windows applications tend to rely on fixed sizes and positions of controls.**
- **A major feature of WPF is a strong set of facilities for creating applications with dynamic sizing and positioning.**
  - Dynamic sizing and positioning is good when a program's user interface may be translated into a foreign language, or when the user of the program changes the size of the system font.
- **Layout in WPF relies on interaction between parent elements such as panels and child elements.**
  - Parents and children collaborate on determining layout, with the parent having the final say.

# Controlling Size

- **Whenever layout occurs for a window, for example by resizing, child elements request a desired size from their parent.**
- **Typically, WPF elements size to fit content.**
  - Even a Window can size to fit content by setting its **SizeToContent** property.
- **The size can be influenced by the child through various properties:**
  - Height and Width
  - Margin
  - Padding
- ***Height* and *Width* in WPF are of data type *double*.**
  - This provides a fine degree of granularity in sizing.
- **The base class where *Height* and *Width* are defined is *FrameworkElement*.**
  - The default value of **Height** and **Width** is **Double.NaN** ("not a number" in floating point terminology), which means that the element is sized to its content.
- **It is usually better not to set *Height* and *Width* explicitly.**
  - Changes, such as user choosing a larger system font, can cause part of the content to be truncated.

# Margin and Padding

- **The two common properties to facilitate dynamic sizing are *Margin* and *Padding*.**
  - All classes derived from **FrameworkElement** have a **Margin** property.
  - All classes derived from **Control** also have a **Padding** property.
- ***Margin* specifies the amount of extra space on the outside of an element.**
- ***Padding* specifies the amount of extra space on the inside of an element, around its content.**



# Thickness Structure

- **Margin and Padding are specified by means of a *Thickness* structure, which has four *Double* properties:**
  - Left
  - Top
  - Right
  - Bottom
- **You may initialize a *Thickness* structure in code via a constructor in two ways:**
  - Pass a single **Double**, which will apply a uniform value to all four sides.
  - Pass four **Double** values, which will apply separate values to left, top, right and bottom.
- **You may initialize *Thickness* via XAML in three ways:**

```
<object property ="left" />

<object property ="left,top" />

<object property =" left,top,right,bottom"  />
```

  - The second option, not available in code, will provide symmetrical values for left/right and top/bottom.

# Children of Panels

- ***Panel* has a property *Children* that is used to store child elements.**
  - **Children** is an object of type **UIElementCollection**.
  - **UIElementCollection** is a collection of UIElement objects.
- **There is a great variety of elements that can be stored in a panel, including any kind of control.**
- **You can add a child element to a panel in code via the *Add()* method of *UIElementCollection*.**

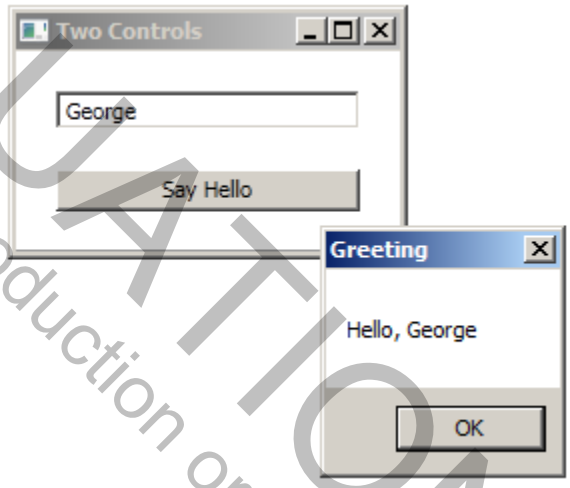
```
StackPanel panel = new StackPanel();
...

Button btnSayHello = new Button();
...

panel.Children.Add(btnSayHello);
```

- **You can add child elements to a panel in XAML by nesting them within the panel element.**

```
<StackPanel>
     <TextBox Margin="20"
              Name="txtName">
     </TextBox>
     <Button Margin="20, 0, 20, 20"
             Name="btnSayHello"
             Click="btnSayHello_Click">
         Say Hello
     </Button>
 </StackPanel>
```

# Example – TwoControlsXaml

- **The example program *TwoControlsXaml\Vertical* illustrates use of a *StackPanel*, whose children are a TextBox and a Button.**
  - The user can type in a name in the text box, which is used in the greeting message.




  - The program also illustrates various automatic sizing features of WPF.

- **This version of the program uses the StackPanel's default vertical orientation of child controls.**

# TwoControls – XAML

- **Here is the complete XAML:**

```
<Window x:Class="TwoControlsXaml.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/
        2006/xaml"
    Title="Two Controls"
    SizeToContent="Height"
    Width="200">
    <StackPanel>
        <TextBox Margin="20"
                 Name="txtName">
        </TextBox>
        <Button Margin="20, 0, 20, 20"
                Name="btnSayHello"
                Click="btnSayHello_Click">
            Say Hello
        </Button>
    </StackPanel>
</Window>
```

# Automatic Sizing

- **Only the width of the main window is specified.**
- **The height of the main window is sized to its content, which is a panel containing two controls.**
  - The **SizeToContent** property is used.

```
<Window x:Class="TwoControlsXaml.MainWindow"
    ...
    SizeToContent="Height"
    Width="200">
```

- **The TextBox and Button specify a margin in device-independent pixels.**

```
<TextBox Margin="20"
         Name="txtName">
</TextBox>
<Button Margin="20, 0, 20, 20"
        Name="btnSayHello"
        Click="btnSayHello_Click">
    Say Hello
</Button>
```

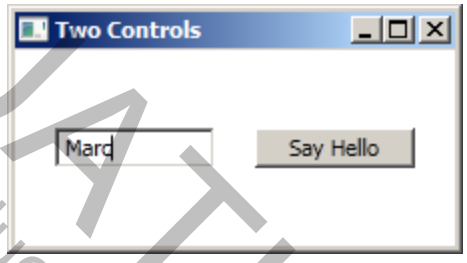
# TwoControls – Code

- **The event handler in the code-behind file extracts the value in the text box and uses it in the greeting.**

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

      private void btnSayHello_Click(object sender,
      RoutedEventArgs e)
      {
         MessageBox.Show("Hello, " + txtName.Text,
            "Greeting");
      }
   }
}
```

# Orientation

- **The StackPanel has a property *Orientation* that controls how child elements are laid out.**
  - The default **Orientation** is Vertical, and the other choice is Horizontal.
  - **Chap02\TwoControls\Horizontal** illustrates horizontal orientation.

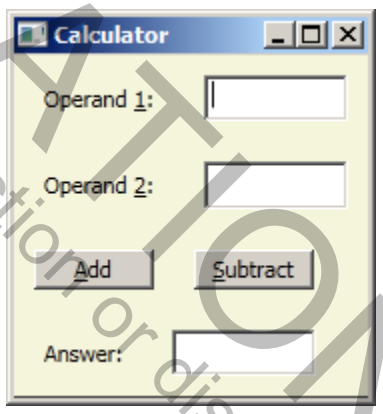



- **In the XAML we also change a few other properties to illustrate the layout of controls.**

```
<Window x:Class="TwoControlsXaml.MainWindow"
    ...
    SizeToContent="Width"
    Height="125">
    <StackPanel Orientation="Horizontal">
        <TextBox Margin="20"
                 Width="80"
                 VerticalAlignment="Center"
        ...
        <Button Margin="0,20,20,20"
                Width="80"
                VerticalAlignment="Center"
        ...
    </StackPanel>
</Window>
```

# Access Keys

- **WPF enables you to provide *access keys* as an alternative means of accessing UI elements.**
  - You can designate certain characters that will appear underlined when the user presses the Alt key.
  - Pressing the Alt key and the designated character produces special action, such as setting the focus or clicking a button.
- **See the *AccessKeyDemo* program in this chapter.**




  - Alt + 1 will set the focus to the first text box.
  - Alt + 2 will set the focus to the second text box.
  - Alt + A is equivalent to clicking the Add button.
  - Alt + S is equivalent to clicking the Subtract button.

# Access Keys in XAML

- **No code is required to make access keys work—you can do everything in XAML.**
  - Place an underscore in front of the character you want to serve as the access key. For a button, that is all you have to do.

```
<Button Margin="10"
        Width="60"
        Name="btnAdd"
        Click="btnAdd_Click">
    _Add
</Button>
```

  - In the case of a Label, you should also use the **Target** attribute to specify the associated control that will gain the focus.

```
<Label Margin="10"
       Target="{Binding ElementName=txtOp1}">
    Operand _1:
</Label>
<TextBox Margin="10"
   Width="72"
   Name="txtOp1"/>
```

# Content Property

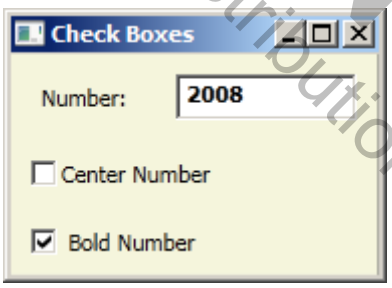
- **The content of a control in XAML may be placed between the start and end tags for the control.**

```
<CheckBox Margin="10" Name="chkCenter"
          ...>
   Center Number
</CheckBox>
```

- **You may also explicitly use the *Content* attribute.**

```
<CheckBox Margin="10" Name="chkBold"
          Content=" Bold Number" >
          ...>
</CheckBox>
```

- **Using the Content attribute gives you control over whitespace.**
  - Observe the better appearance of the "Bold Number" check box in the example **CheckBoxes**.




- **The content may typically be anything, not just a string.**

# Checked and Unchecked Events

- **Checking and unchecking a check box fires the events *Checked* and *Unchecked*.**
  - Our example provides handlers for these events.

```
<CheckBox Margin="10" Name="chkCenter"
          Checked="chkCenter_Checked"
          Unchecked="chkCenter_Checked">
   Center Number
</CheckBox>
<CheckBox Margin="10" Name="chkBold"
          Checked="chkBold_Checked"
          Unchecked="chkBold_Unchecked"
          Content=" Bold Number" >
</CheckBox>
```

  - Both events have a common handler.

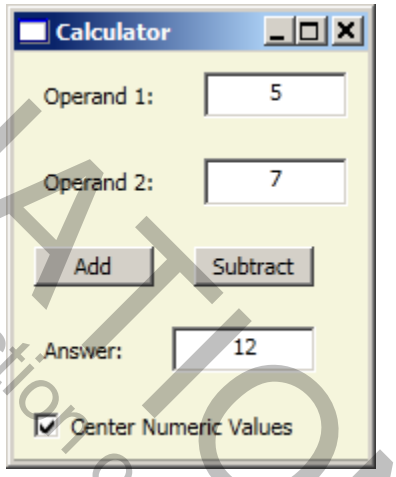
- **We distinguish the two states by the *IsChecked* property, which is of type *Nullable*<*Boolean*>.**
  - Cast it to **bool** to use it as a condition in an **if** statement.

```
private void chkCenter_Checked(object sender,
RoutedEventArgs e)
{
   if ((bool)chkCenter.IsChecked)
   {
      txtNumber.TextAlignment =
         TextAlignment.Center;
   }
   else
   {
      txtNumber.TextAlignment = TextAlignment.Left;
   }
}
```

# Lab 2

## Calculator Program via XAML

In this lab you will incrementally create a XAML version of a **Calculator** program. You'll first create the user interface. Then you'll implement the program's functionality. Finally, you'll add some enhancements to the program.




Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 45 minutes

# Property Element Syntax

- **Sometimes attribute syntax is not feasible, because the object necessary to provide the needed information cannot be expressed as a simple string.**
  - For example, suppose the content is an ellipse.
  - See the example program **EllipseDemo** in which there are several buttons whose content is an ellipse.
- **You could then use this notation:**

```
<Button Margin="10"
        Background ="LightGray">
  <Button.Content>
     <Ellipse Height="60" Width="120" Fill="Red" />
  </Button.Content>
</Button>
```

  - The syntax is <TypeName.Property>.

- **Or more compactly in special case of the *Content* property:**

```
<Button Margin="10,0,10,10"
        Background ="LightGray">
   <Ellipse Height="60" Width="120" Fill="Green" />
</Button>
```

# Type Converters

- **There is a subtlety in the way we commonly write attributes in XAML.**
  - Consider the **Background** attribute of **Button** and the manner we used it on the previous page.

```
<Button Margin="10,0,10,10"
        Background ="LightGray">
   <Ellipse Height="60" Width="120" Fill="Green" />
</Button>
```

  - The type of the **Background** property is **Brush**.
  - How does the string "LightGray" get converted to a particular kind of **Brush**?
- **XAML provides a number of *type converters* that will perform useful type conversions automatically.**
- **Without this feature, you would have to specify the *Background* property using more cumbersome property element syntax.**

```
<Button Margin="10,0,10,10">
   <Button.Background>
       <SolidColorBrush Color="LightGray" />
   </Button.Background>
   <Ellipse Height="60" Width="120" Fill="Blue" />
</Button>
```

# Summary

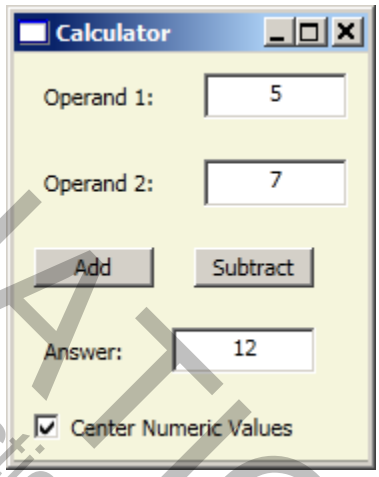
- **Extensible Application Markup Language (XAML) provides a declarative XML-based syntax for defining user interfaces in WPF.**
- **There are two XML namespaces that you must use in your XAML documents.**
- **Visual Studio 2017 makes it easy to create and edit XAML documents.**
- **Access key support can be provided in XAML without use of procedural code.**
- **The content property can be used as shorthand in lieu of the Content attribute.**
- **You can use property elements to set values for complex properties in XAML documents.**
- **Type converters can convert strings to other WPF types.**

# Lab 2

## Calculator Program via XAML

### Introduction

In this lab you will incrementally create a XAML version of a **Calculator** program. You'll first create the user interface. Then you'll implement the program's functionality. Finally, you'll add some enhancements to the program.




**Suggested Time:** 45 minutes

**Root Directory:** **OIC\WpfCs**

**Directories:**

| | |
|---|---|
| **Labs\Lab2** | (do your work here) |
| **Chap02\Calculator\Step1** | (answer to Part 1) |
| **Chap02\Calculator\Step2** | (answer to Part 2) |
| **Chap02\Calculator\Step3** | (answer to Part 3) |

### Part 1. Basic User Interface

1. Use Visual Studio to create a new WPF application **Calculator** in the **Lab2** folder.

2. Edit the starter XAML to make the title "Calculator", and replace the grid by a stack panel with beige background.

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

3. Provide XAML for the top pair of label and text box, for the first operand. They should be inside a nested stack panel with horizontal orientation.

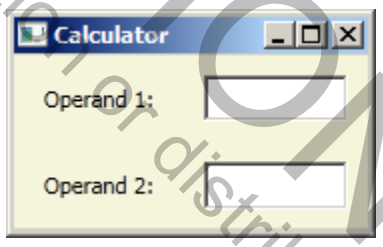
```
<StackPanel Background="Beige">
     <StackPanel Orientation="Horizontal">
         <Label Margin="10">
             Operand _1:
         </Label>
         <TextBox Margin="10"
                  Width="72"
                  Name="txtOp1"/>
     </StackPanel>
</StackPanel>
```

4. Provide the XAML for the second pair of label and text box. Build and run. You should see your controls, but also a lot of extra space in the window.

5. Replace the hardcoded width and height by specifying **SizeToContent** as "WidthAndHeight"..

```
<Window x:Class="Calculator.MainWindow"
    ...
    SizeToContent="WidthAndHeight">
```

6. Build and run. You should see the window with the controls you've laid out so far, reasonably sized.

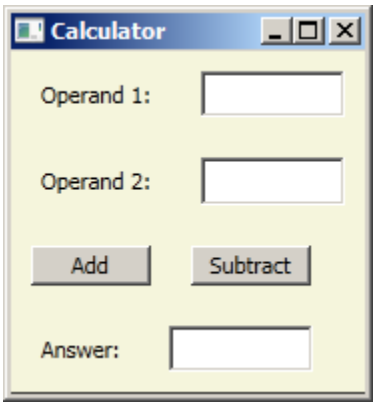



7. Add another horizontal stack panel with a pair of buttons for Add and Subtract.

```
<StackPanel Orientation="Horizontal">
   <Button Margin="10"
           Width="60"
           Name="btnAdd">
       Add
   </Button>
   <Button Margin="10"
           Width="60"
           Name="btnSubtract">
       Subtract
   </Button>
</StackPanel>
```

8. Add a fourth horizontal stack panel with another label and text box pair, for the answer. This time the text box should be read-only.

9. Build and run. Your basic user interface is now complete, and you are at Step 1.




**Part 2. Basic Functionality**

1. Add a handler for the Click event of the Add button. The simplest way to do this is to double-click on the Add button in Design view.

2. In the event handler provide code to convert the strings entered for the operands to numbers, add these numbers, and store the answer.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
   int num1 = Convert.ToInt32(txtOp1.Text);
   int num2 = Convert.ToInt32(txtOp2.Text);
   int answer = num1 + num2;
   txtAns.Text = answer.ToString();
}
```

3. In a similar manner provide a handler for the Subtract button.

4. Build and run. Do the Alt + 1 and Alt + 2 access keys work to position focus at the first and second text boxes, respectively? You are at Step 2.

**Part 3. Enhancements**

1. Although we see the 1 and 2 characters underlined, the Alt access keys do not work yet. Provide a Target attribute on the first label.

```
<Label Margin="10"
       Target="{Binding ElementName=txtOp1}">
    Operand _1:
</Label>
<TextBox Margin="10"
         Width="72"
         Name="txtOp1"/>
```

2. Provide a similar target for the second label.

3. Add access key support for the buttons simply by providing an underscore in front of the desired letter.

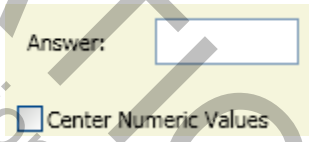
```
<Button Margin="10"
        Width="60"
        Name="btnAdd" Click="btnAdd_Click">
    _Add
</Button>
<Button Margin="10"
        Width="60"
        Name="btnSubtract" Click="btnSubtract_Click">
    _Subtract
</Button>
```

4. Build and run. The access keys should now work.

5. Add XAML for a check box indicating whether or not to center the numeric values.

```
<CheckBox Margin="10" Name="chkCenter">
   Center Numeric Values
</CheckBox>
```

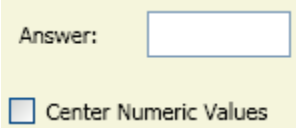
6. Build and run. The caption for the check box is jammed up against the box.




7. Achieve a better appearance by using the Content attribute notation and provide a leading space.

```
<CheckBox Margin="10" Name="chkCenter"
          Content=" Center Numeric Values">
</CheckBox>
```

8. Build and run. The appearance should be better!




9. Add handlers for the Checked and Unchecked events of the check box. When adding the handler for Unchecked, do not add a new handler, but rather choose the already existing handler for Checked. Note that by double-clicking the check box in Design view you will add the same handler for both events.

10. Implement this common handler. Note that you will need to cast **IsChecked** to **bool**.

```
private void chkCenter_Checked(object sender, RoutedEventArgs e)
{
   if ((bool)chkCenter.IsChecked)
```

```
    {
       txtOp1.TextAlignment = TextAlignment.Center;
       txtOp2.TextAlignment = TextAlignment.Center;
       txtAns.TextAlignment = TextAlignment.Center;
    }
    else
    {
       txtOp1.TextAlignment = TextAlignment.Left;
       txtOp2.TextAlignment = TextAlignment.Left;
       txtAns.TextAlignment = TextAlignment.Left;
    }
}
```

11. Build and run. Your little calculator should now be fully functional! You're at Step 3.