Windows Presentation Foundation Using C#

Windows Presentation Foundation Using C#

Student Guide

Revision 4.7

Object Innovations Course 4135

Windows Presentation Foundation Using C#
Rev. 4.7

Student Guide

™ is a trademark of Object Innovations.

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_47.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 .NET Framework 4.6.**
- **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 2015.**
- **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?

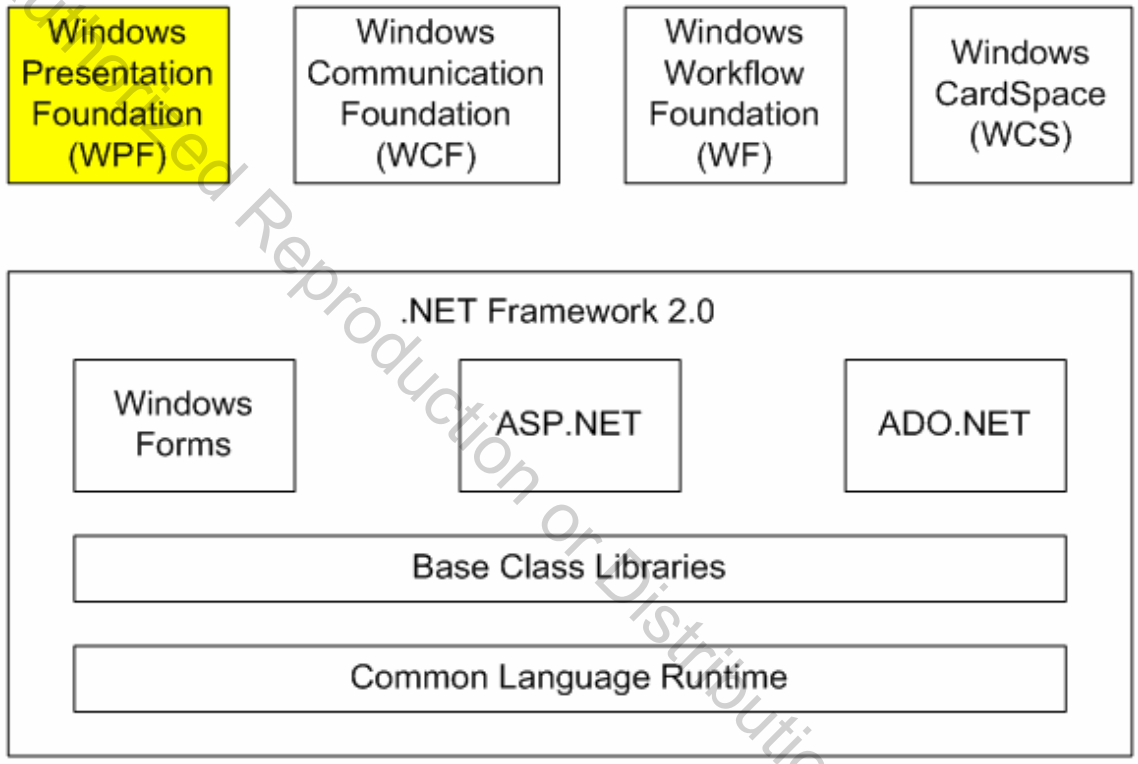
- **The various technologies support development of sophisticated graphics and GUI programs, but there are several different, complex technologies a programmer may need to know.**
- **The goal of Windows Presentation Foundation is to provide a unified framework for creating modern user experiences.**
  - 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 2015 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 a significant 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 4.0/4.6

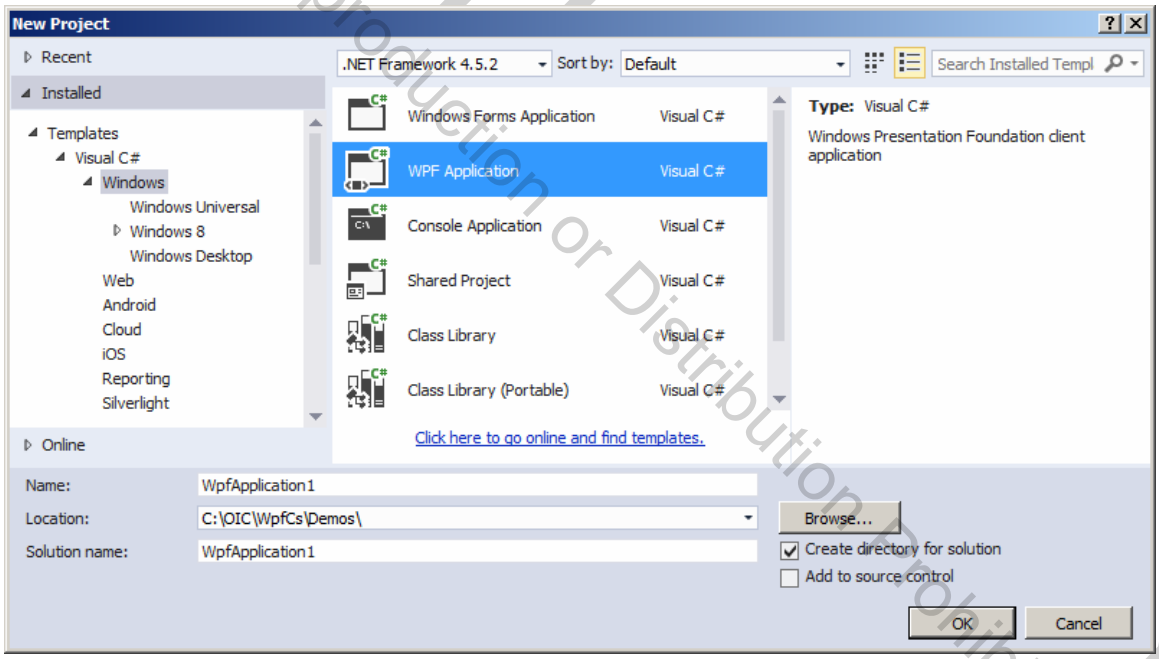
- **The .NET Framework 3.5 added a number of important features beyond those of .NET 3.0.**
  - Notable was integration with the tooling support provided by Visual Studio 2008.
  - Language Integrated Query (LINQ) extends query capabilities to the syntax of the C# and Visual Basic programming languages.
  - 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 4.0/4.6 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 2015

- **Visual Studio 2015 provides effective tooling support for .NET Framework 4.6.**
  - Early support for WinFX involved add-ons to Visual Studio, but now there is a fully integrated environment.
- **Visual Studio 2015 has a new IDE with an attractive new graphical appearance.**
  - VS 2015 is implemented using WPF.
- **Features in Visual Studio 2015 include:**
  - Improvements in the Integrated Development Environment (IDE), such as better navigation and easier 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, .NET 4.5.1, NET 4.5.2 and .NET 4.6.
- **There are many project templates, including:**
  - WPF projects
  - WCF projects
  - WF projects
  - Reporting projects
- **There are a number of designers, including WPF/Silverlight Designer, an object/relational designer, and a workflow designer.**

Visual Studio Community 2015

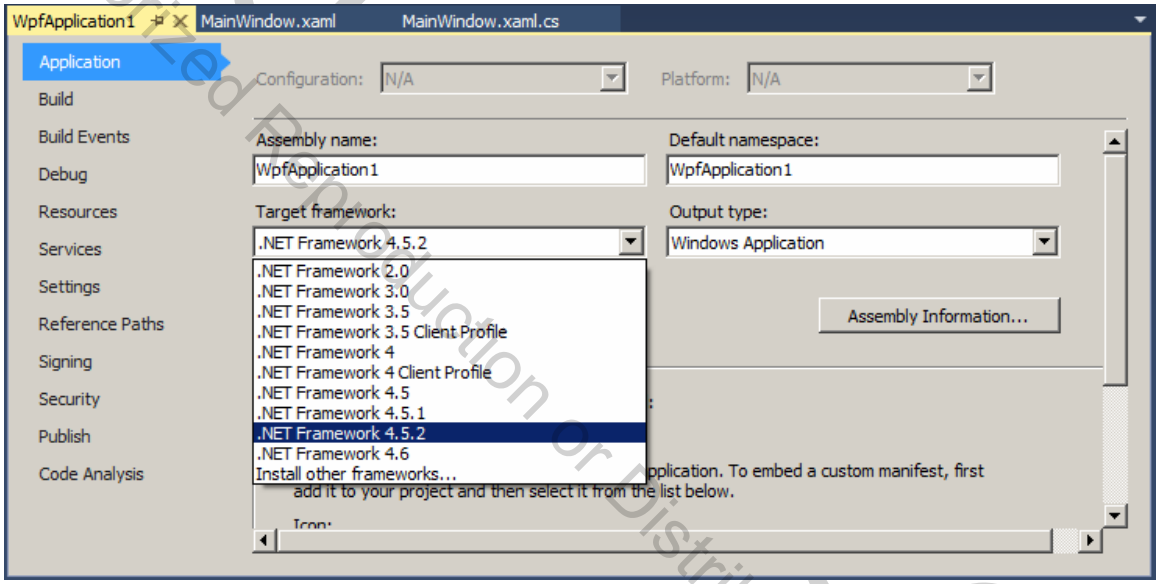
- **A noteworthy aspect of Visual Studio 2015 is a strong free Communnity version of the tool.**
- **In this course we will rely on Visual Studio Community 2015.**
  - It supports multiple language development (C#, Visual Basic, and C++).
  - It supports the creation of WPF projects.
  - It also supports 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/.NET 4.6.**

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 2015 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.

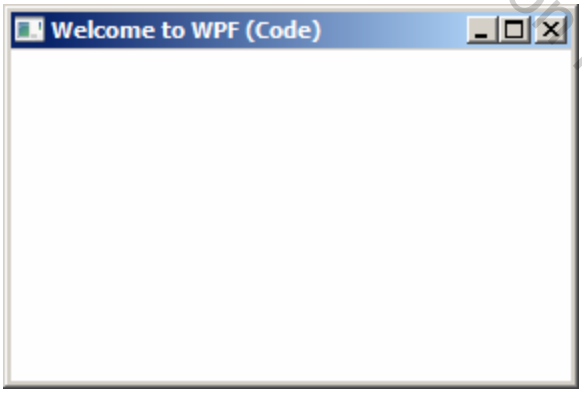
```
using System;
using System.Windows;

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

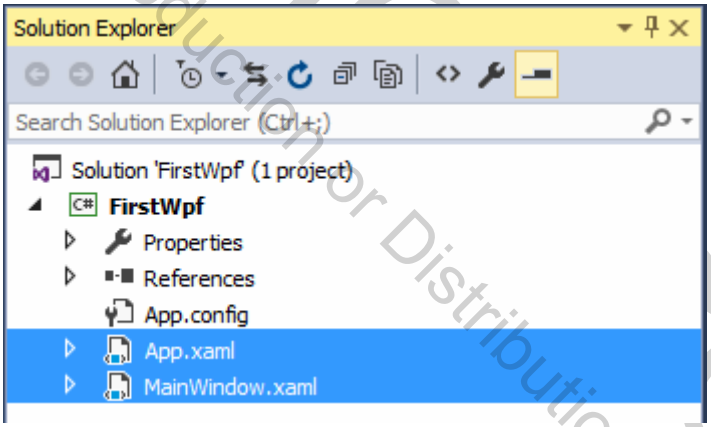
- **Our example program has the following features:**
  - Import the **System.Windows** namespace. This namespace includes the fundamental WPF classes, interfaces, delegates, and so on, including the classes Application and Window.
  - Make your class derive from the **Window** class.
  - Provide the 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 2015

- **Although you can compile WPF programs at the command-line, for simplicity we will use Visual Studio 2015 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 2015!

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*.**

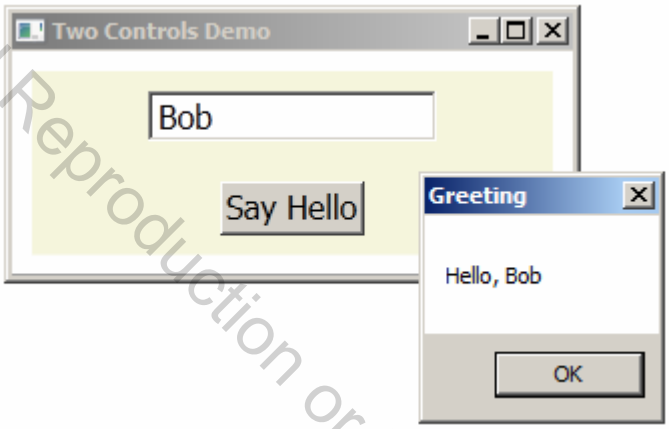
```
StackPanel panel = new StackPanel();
...

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 .NET Framework.**
  - In .NET 3.0/3.5, it is layered on top of .NET Framework 2.0.
  - In .NET 4.0/4.6 there is a new 4.0 runtime.
- **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 2013, 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 9

Resources

Resources

Objectives

After completing this unit you will be able to:

- **Understand what resources are and how you can use them throughout the application.**
- **Take advantage of using logical resources, a new type of resource supported by WPF.**
- **Use resources from XAML and from procedural code.**
- **Understand the difference between the StaticResource and DynamicResource markup extensions.**

Resources in .NET

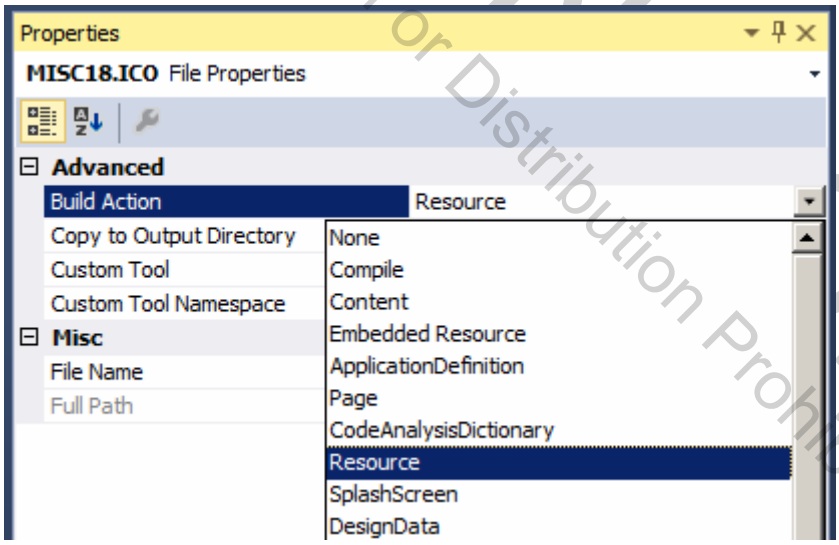
- **Resources are parts of a program that aren't code.**
- **You can think of resources as images, fonts, audio files and video files.**
- **Resources can even be just a collection of strings.**
  - This is often used to add multi-language support to an application.
  - Instead of writing a literal directly into the code, you just make a reference to a string in a resource file.
  - Then you can use .NET framework classes to load different resource files for each culture / language.

Resources in WPF

- **The core .NET resources system is supported by WPF.**
- **Additionally, WPF supports a number of new features.**
- **There are two types of resources in WPF: binary resources and logical resources.**
  - Binary resources are common items like images or audio files.
  - Logical resources can be any arbitrary .NET object.

Binary Resources

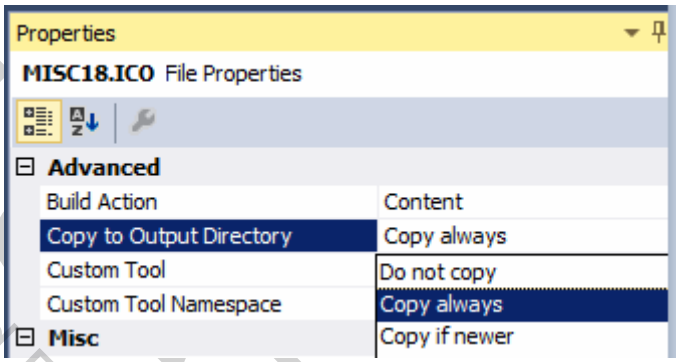
- **Binary resources are everything that was known as resources in .NET framework so far.**
- **WPF stores compiled XAML files as binary resources behind the scenes.**
- **This type of resources can be put into two categories:**
  - **Localizable resources** which can be different depending on culture.
  - **Language-neutral resources** that do not depend on culture.
- **A typical way of defining a resource is to add the file to the Visual Studio project and set a build action for it.**



  - By doing this, the resource will be packaged into the project assembly.

Loose Files as Resources

- **An alternative to this is to use the Content build action.**



  - But if you do so, you should change the **Copy to Output Directory** property to "Copy always" or "Copy if newer", because the resource won't be embedded in the assembly.
  - As the resource will be a loose file in the output directory, you can easily update the file without needing to rebuild or deploy the application again.

- **You can still use a resource that has not been added to the project (despite this being not satisfactory).**
  - You can reference the resource using the full path.

```
<Button Margin="10">
   <Image
      Source="c:\OIC\Data\Graphics\MISC20.ICO"/>
</Button>
```

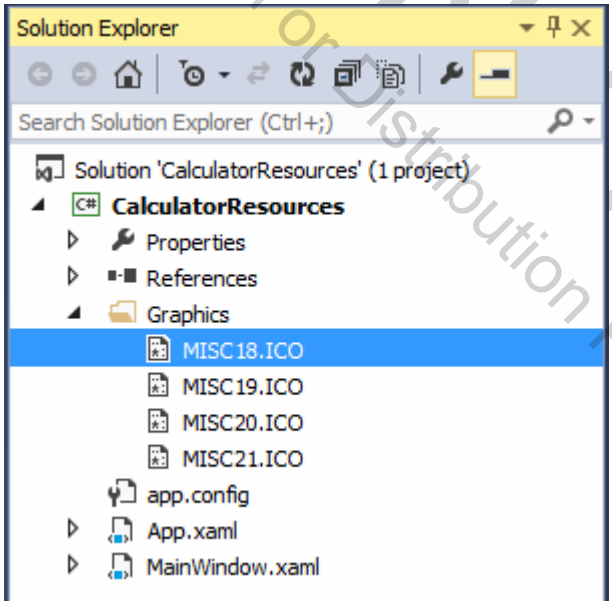
- **It's still possible to reference a resource programmatically.**
  - This is useful when the resource is generated at run-time.

Binary Resources Example

- **Examine the code in the *CalculatorResources* folder in the chapter directory.**
  - There is a StackPanel with four buttons.
  - The buttons have an image as their content, referenced as a relative path inside the project.

```
<Button Margin="10"
        Name="btnAdd"
        Click="btnAdd_Click">
   <Image Source="Graphics/MISC18.ICO"/>
</Button>
```

  - The images were added to the project, with the build action set to Resource and they are not copied to the output directory. (They are embedded within the assembly.)



Logical Resources

- ***Logical resources*** **are a new type of resource specific to WPF.**
- **Any .NET object can be a logical resource.**
- **The object must be stored and named in a resources property (that is a collection) to be used along the code.**
  - The resources property can be set locally in a XAML element or in a higher scope.
  - The code below defines a Resources property in the Window element, with three different SolidColorBrush objects.

```
<Window x:Class="SimpleBrush.MainWindow"
   ...>
   <Window.Resources>
      <SolidColorBrush
         x:Key="windowBackgroundBrush">Beige
      </SolidColorBrush>
      <SolidColorBrush
         x:Key="buttonBackgroundBrush">LightGreen
      </SolidColorBrush>
      <SolidColorBrush
         x:Key="buttonForegroundBrush">Blue
      </SolidColorBrush>
   </Window.Resources>
   ...
```

Logical Resources (Cont'd)

- **The resources defined in the Window element are visible to any element that is below the Window in the element's tree hierarchy.**
  - The **x:Key** property defines a name under which the resource will be visible to other elements.
  - The resource is referenced using the StaticResource markup extension, which is responsible for finding the resource.

```
<Button Name="btnSayHello"
        FontSize="16"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Background="{StaticResource
buttonBackgroundBrush}"
        Foreground="{StaticResource
buttonForegroundBrush}"
        Click="BtnSayHello_Click">
```

- **We'll see more on the StaticResource markup extension later in this chapter.**

Logical Resources Demo

- **We'll illustrate the use of logical resources with the solution in the *Demos\SimpleBrush* folder, backed up in *SimpleBrush\Step1* in the chapter directory.**

1. Build and run the SimpleBrushXaml solution. You'll see a simple window with just one button as the content.

2. In the XAML code, the button and the window have some color formatting defined directly in the element's syntax.

```
<Window
   ...
   Title="Simple Brushes" Height="192" Width="288"
   Background="Beige">
   <Button Name="btnSayHello"
           FontSize="16"
           HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Background="LightGreen"
           Foreground="Blue"
           Click="BtnSayHello_Click">
      Say Hello
   </Button>
</Window>
```

Logical Resources Demo (Cont'd)

3. We have seen before that the colors used as background and foreground are brushes. Add these Brush objects to the Window's resource collection.

```
<Window
   ...
   Background="Beige">
   <Window.Resources>
      <SolidColorBrush
         x:Key="windowBackgroundBrush">Beige
      </SolidColorBrush>
      <SolidColorBrush
         x:Key="buttonBackgroundBrush">LightGreen
      </SolidColorBrush>
      <SolidColorBrush
         x:Key="buttonForegroundBrush">Blue
      </SolidColorBrush>
   </Window.Resources>
   ...
```

4. Now you can replace the current **Background** and **Foreground** values by a reference to the appropriate resource.

```
<Button Name="btnSayHello"
        FontSize="16"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Background="{StaticResource
buttonBackgroundBrush}"
        Foreground="{StaticResource
buttonForegroundBrush}"
        Click="BtnSayHello_Click">
   Say Hello
</Button>
```

Logical Resources Demo (Cont'd)

5. Build and run the application. You'll see that everything sill works as before, but now the button uses resources for the background and foreground colors.

6. Now you can replace the Background property value of the Window element.

```
<Window
    ...
    Title="Simple Brushes" Height="192" Width="288"
    Background="{StaticResource
windowBackgroundBrush}">
```

7. Build the application and run it in the debug mode (press F5). You'll hit an exception because windowBackgroundBrush isn't defined yet. (Notice that the resources are defined after this code section.)

8. As a workaround, we can change the way we set the Window's Background property. Remove the property from the Window tag and set this property after the resources are defined.

```
<Window
    ...
    Title="Simple Brushes" Height="192" Width="288">
    <Window.Resources>
       <SolidColorBrush
          x:Key="windowBackgroundBrush">Beige
       </SolidColorBrush>
       ...
    </Window.Resources>
    <Window.Background>
       <StaticResource
          ResourceKey="windowBackgroundBrush"/>
    </Window.Background>
```

Logical Resources in Code

9. Build and run the application. Now the application is using resources defined in XAML for brushes.

   – The application is saved at this point in the **SimpleBrush\Step2** folder in the chapter directory.

   – Going further, we can change this application to still use resources, but in procedural code.

10. Continuing with the demo, open the code-behind file **MainWindow.xaml.cs**.

11. Add the three brushes as resources just before the **InitializeComponent()** call.

```
public MainWindow()
{
    this.Resources.Add("windowBackgroundBrush",
       new SolidColorBrush(Colors.Beige));
    this.Resources.Add("buttonBackgroundBrush",
       new SolidColorBrush(Colors.LightGreen));
    this.Resources.Add("buttonForegroundBrush",
       new SolidColorBrush(Colors.Blue));

    InitializeComponent();
}
```

12. Now you can remove the Window.Resources tag from the XAML, as the resources are already defined in the code-behind file.

13. Build and run the application, and note that everything still works as before. The resources are being defined programmatically, but accessed through XAML.

Logical Resources in Code (Cont'd)

14. Now we'll change the code to set the properties accessing the resources programmatically. Do this just after the **InitializeComponent()** call.

```
public MainWindow()
{
   this.Resources.Add("windowBackgroundBrush",
      new SolidColorBrush(Colors.Beige));
   this.Resources.Add("buttonBackgroundBrush",
      new SolidColorBrush(Colors.LightGreen));
   this.Resources.Add("buttonForegroundBrush",
      new SolidColorBrush(Colors.Blue));

   InitializeComponent();

   this.Background = (Brush)this.FindResource(
      "windowBackgroundBrush");
   btnSayHello.Background =
      (Brush)this.FindResource(
         "buttonBackgroundBrush");
   btnSayHello.Foreground =
      (Brush)this.FindResource(
         "buttonForegroundBrush");
}
```

15. Remove the **Background** and **Foreground** properties from the Button tag, and the **Window.Background** tag.

16. Build and run the application. Everything works fine as before, using resources defined and accessed programmatically!

- **The application is saved at this point in the *SimpleBrush\Step3* folder in the chapter directory.**

Static Resources

- **As seen before, the StaticResource markup extension looks for a resource using a key, passed as a parameter.**

```
<Button Name="btnSayHello"
        ...
        Background="{StaticResource
buttonBackgroundBrush}"
        Foreground="{StaticResource
buttonForegroundBrush}"
        ...
</Button>
```

- **This key corresponds to an *x:Key* parameter of a resource defined somewhere in the application.**
- **The resource can be defined within many scopes in the application. It can be defined:**
  - Inside any element's resource property.
  - In the resource property of any parent element.
  - In the resource property of the Application object.
- **The StaticResource markup extension will look for the resource following the order above, and if not found, it throws an *InvalidOperationException*.**
  - Hence, we can improve the sharing capability of a resource if we define it in the root element.

Dynamic Resources

- **The StaticResource markup extension, seen in the previous page, applies the resource only once (actually, in the first time it's needed).**
- **If we use the DynamicResource markup extension instead, we're telling WPF that we want the resource to be applied every time it changes.**
  - Hence, it's important to consider an overhead on looking for resource updates.
- **StaticResource can be used almost anywhere in the code, and DynamicResource can only be used to set dependency property values.**
  - Dependency properties were discussed in the preceding chapter.

DynamicResource Example

- **See *SimpleBrushDynamic* in the chapter directory for a DynamicResource usage example.**
  - The first think to notice is that we can set a property using DynamicResource even if the resource will be defined later in the code. We've already seen that this is not possible when using StaticResource.

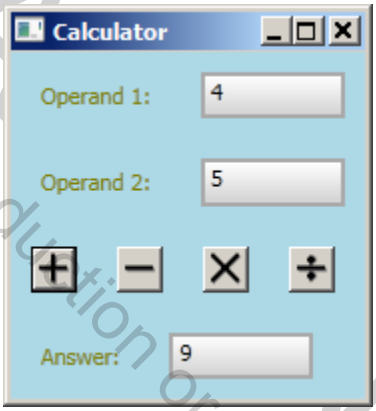
```
<Window
   ...
   Background="{DynamicResource
windowBackgroundBrush}">
   <Window.Resources>
      <SolidColorBrush
         x:Key="windowBackgroundBrush">Beige
      </SolidColorBrush>
      ...
    </Window.Resources>
```

- **The example contains two buttons, and its *Background* and *Foreground* properties are set to the same resources, but one uses StaticResource and the other uses DynamicResource.**
- **Additionally, there is a ComboBox that changes the resource *buttonBackgroundBrush*.**
- **Build and run the application.**
  - Notice that when you change the color of the SolidColorBrush resource using the ComboBox, only the button that uses DynamicResource has its Background updated.

Lab 9

Adding Some Colors to the Calculator

In this lab you will use your knowledge on resources to create a colorful version of the Calculator program. You are provided with a starter solution that uses some brushes directly referenced in each element's properties, and there are buttons that use full path for its images. You will define resources for the brushes and configure the images as binary resources in the project.



Detailed instructions are contained in the Lab 9 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

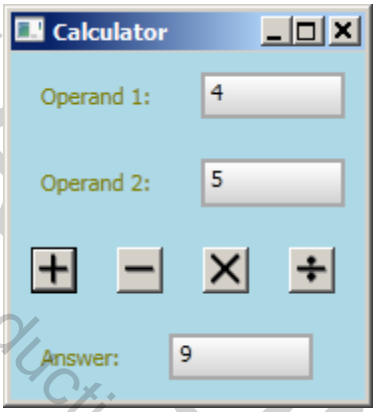
- **Resources are parts of a program that aren't code.**
- **Logical resources are a new type of resources supported by WPF, and it can store any .NET object.**
- **Resources can be used from XAML and from procedural code.**
- **You can use the StaticResource and DynamicResource markup extensions to reference a resource.**

Lab 9

Adding Some Colors to the Calculator

Introduction

In this lab you will use your knowledge on resources to create a colorful version of the Calculator program. You are provided with a starter solution that uses some brushes directly referenced in each element's properties, and there are buttons that use full path for its images. You will define resources for the brushes and configure the images as binary resources in the project.



Suggested Time: 30 minutes

Root Directory: **OIC\WpfCs**

Directories:

Labs\Lab9\CalculatorColors	(do your work here)
Chap09\CalculatorColors\Step1	(backup of starter code)
Chap09\CalculatorColors\Step2	(answer to Part 1)
Chap09\CalculatorColors\Step3	(answer to Part 2)

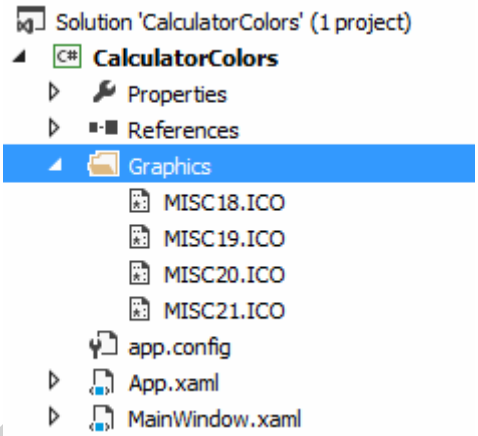
Part 1. Using Binary Resources

1. Build and run the starter program. Notice that this is exactly the Calculator program that you know, but the design was changed a bit to use some color brushes.

2. Examine the file **MainWindow.xaml**. You will find four buttons, with images that are references using full path.

```
<Button Margin="10"
        Name="btnAdd"
        Click="btnAdd_Click">
   <Image Source="C:\OIC\Data\Graphics\MISC18.ICO"/>
</Button>
```

3. This is not a good practice. Copy the image files from the **OIC\Data** folder to a new **Graphics** directory inside the project folder and add them to the project.

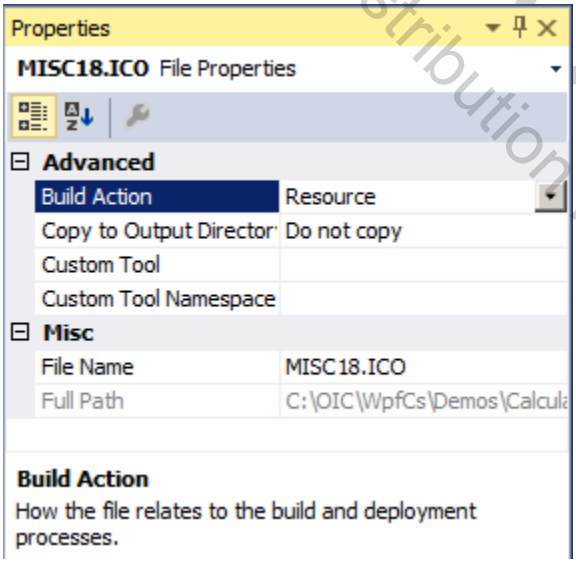


4. Now, change the references to the images to the new relative path inside the project. Do this for the four images.

```
<Button Margin="10"
        Name="btnAdd"
        Click="btnAdd_Click">
   <Image Source="Graphics/MISC18.ICO"/>
</Button>
```

5. Build and run your application. You'll see that the images are shown correctly as before, but now they are references as binary resources inside the project.

6. Close the application window and show the properties of the **MISC18.ICO** file.



7. By default, the image files have the Build Action property set to Resource and the Copy to Output Directory property set to Do not copy, so this way the files are embedded to the compiled assembly. You can check this by going to the **bin\Debug** folder in the project directory and see that the image files are not there, despite being correctly shown in the application window. Let's assume that we want the images to be easily replaced if necessary, without needing to rebuild the project. To achieve this, change each image's properties so that the Build Action is set to Content and the Copy to Output Directory is set to Copy if newer.

8. Build and run the application. The images are still being shown as expected, but now you can see them in the Graphics folder that was copied to the **bin\Debug** folder in the project directory. This gives you the flexibility we want, if we need to change the images easily. You are now at step 2.

Part 2. Using Logical Resources

1. Examine the file **MainWindow.xaml**. Note that the colors used in the interface are brushes that are directly set in the elements' properties.

```
<StackPanel Background="LightBlue">
   <StackPanel Orientation="Horizontal">
      <Label Margin="10"
             Target="{Binding ElementName=txtOp1}"
             Foreground="Olive"
             >
         Operand _1:
      </Label>
      <TextBox Margin="10"
               Width="72"
               Name="txtOp1"
               BorderBrush="DarkGray"
               >
         <TextBox.Background>
            <LinearGradientBrush StartPoint="1,0" EndPoint="1,1">
               <GradientStop Color="White" Offset="0"/>
               <GradientStop Color="LightGray" Offset="1"/>
            </LinearGradientBrush>
         </TextBox.Background>
      </TextBox>
      ...
```

2. These brushes could be set using resources. Let's start defining a resource for the root StackPanel's **Background** color. Add a **Window.Resources** element just before this StackPanel, and define a new SolidColorBrush for the LightBlue color and assign the key **windowBackground** to it.

```
...
<Window.Resources>
   <SolidColorBrush x:Key="windowBackground" Color="LightBlue"/>
</Window.Resources>
<StackPanel Background="LightBlue">
...
```

3. Now, change the background property to reference the new resource using the StaticResource markup extension.

```
<StackPanel Background="{StaticResource windowBackground}">
...
```

4. Build and run the application. Notice that the light blue background is there, but now referenced as a reusable resource.

5. Now let's do the same with all the labels' **Foreground** property and all the textboxes' **BorderBrush** property. The key for the labels' **Foreground** color should be **textForeground** and for the textboxes' **BorderBrush** color should be **boxBorders**. Don't forget to replace the references for these properties for all the three labels and textboxes!

```
<Window.Resources>
   <SolidColorBrush x:Key="windowBackground" Color="LightBlue"/>
   <SolidColorBrush x:Key="textForeground" Color="Olive"/>
   <SolidColorBrush x:Key="boxBorders" Color="DarkGray"/>
</Window.Resources>
<StackPanel Background="{StaticResource windowBackground}">
   <StackPanel Orientation="Horizontal">
      <Label Margin="10"
             Target="{Binding ElementName=txtOp1}"
             Foreground="{StaticResource textForeground}"
             >
         Operand _1:
      </Label>
      <TextBox Margin="10"
               Width="72"
               Name="txtOp1"
               BorderBrush="{StaticResource boxBorders}"
               >
      ...
```

6. Build the application to check that it's compiling.

7. Now let's add the last resource. It's the gradient used as background in the textboxes. You can just copy the LinearGradientBrush from one of the textboxes to the **Window.Resources** element and add an **x:Key** property to it. Set the key to **boxGradient**.

```
<Window.Resources>
   <SolidColorBrush x:Key="windowBackground" Color="LightBlue"/>
   <SolidColorBrush x:Key="textForeground" Color="Olive"/>
   <SolidColorBrush x:Key="boxBorders" Color="DarkGray"/>
   <LinearGradientBrush x:Key="boxGradient"
                        StartPoint="1,0"
                        EndPoint="1,1">
      <GradientStop Color="White" Offset="0"/>
      <GradientStop Color="LightGray" Offset="1"/>
   </LinearGradientBrush>
```

```
</Window.Resources>
...
```

8. Now you can remove the entire **TextBox.Background** element from all three textboxes and include the Background as a property, using the StaticResource markup extension to reference the newly added **boxGradient** resource.

```
<TextBox Margin="10"
         Width="72"
         Name="txtOp1"
         BorderBrush="{StaticResource boxBorders}"
         Background="{StaticResource boxGradient}"
         >
</TextBox>
```

9. Build and run your application. Now you are finished improving your application to use resources correctly!