Table of Contents (Overview)

Directory Structure

- **Install the course software by running the self-extractor *Install_WpfCs_46.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 4.5.1.**
- **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 2013.**
- **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?

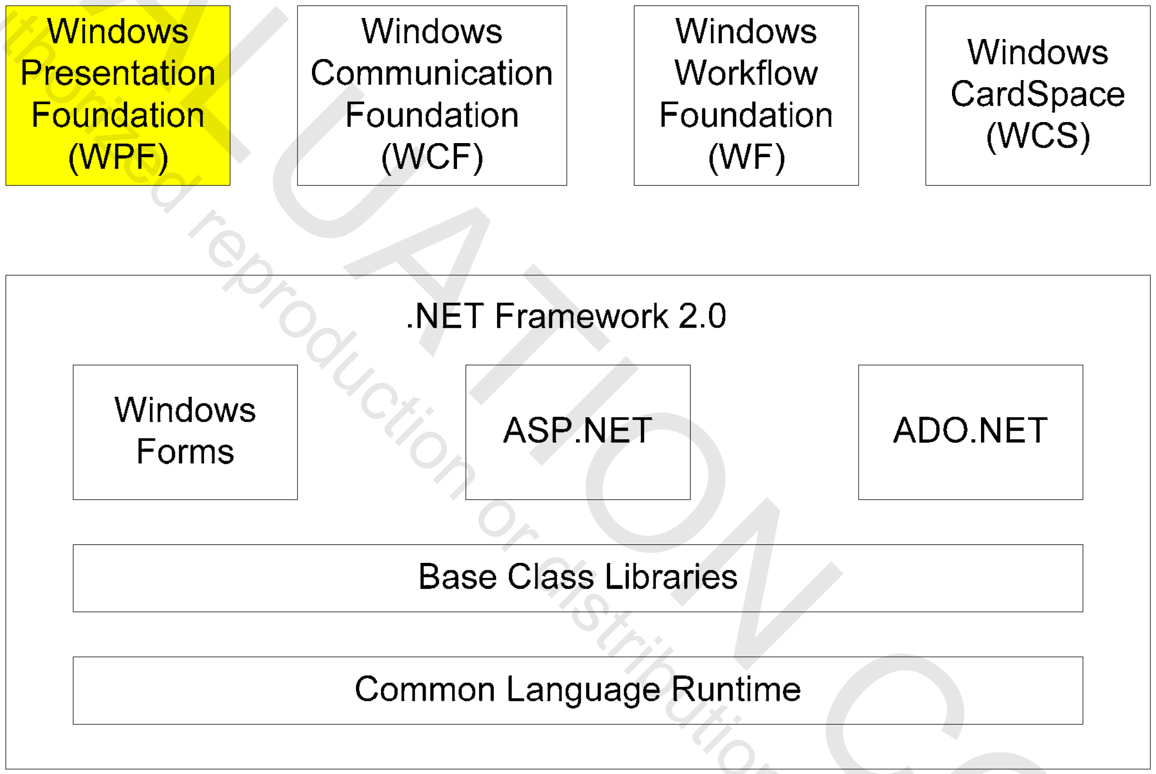
- **The various technologies support development of sophisticated graphics and GUI programs, but there are several different, complex technologies a programmer may need to know.**
- **The goal of Windows Presentation Foundation is to provide a unified framework for creating modern user experiences.**
  - 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 2013 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 4.0/4.5.1

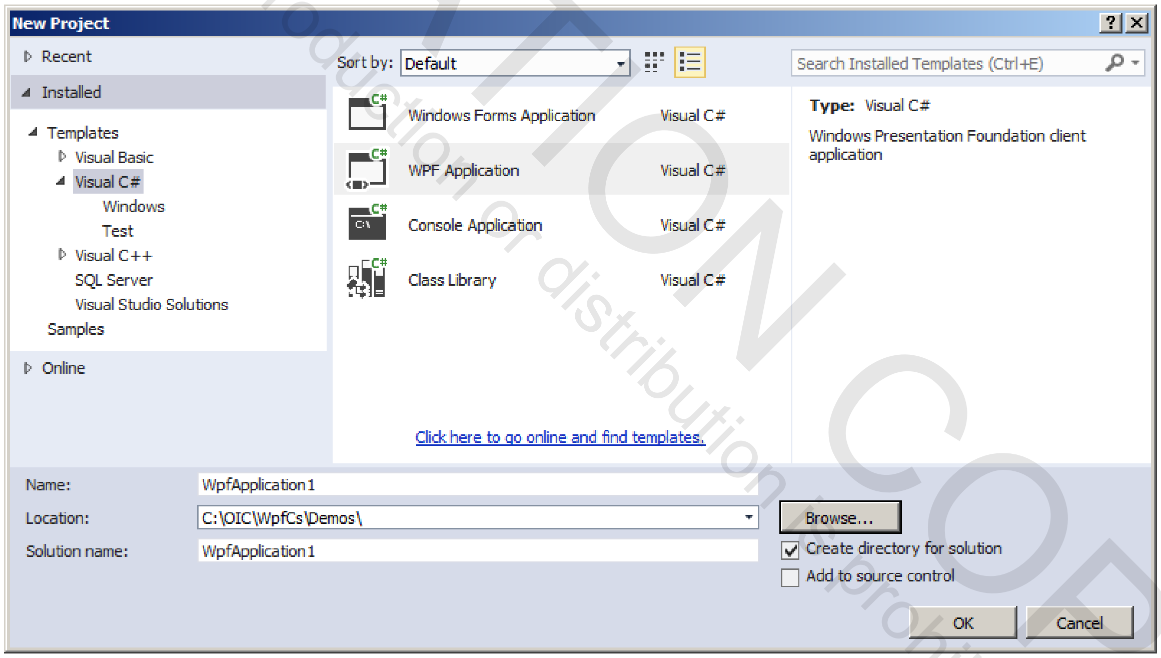
- **The .NET Framework 3.5 added a number of important features beyond those of .NET 3.0.**
  - Notable was integration with the tooling support provided by Visual Studio 2008.
  - Language Integrated Query (LINQ) extends query capabilities to the syntax of the C# and Visual Basic programming languages.
  - 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 4.0/4.5.1 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 5 for Web development.
  - A new programming model for parallel programming.
  - And much more!

Visual Studio 2013

- **Visual Studio 2013 provides effective tooling support for .NET Framework 4.5.1.**
  - Early support for WinFX involved add-ons to Visual Studio, but now there is a fully integrated environment.
- **Visual Studio 2013 has a new IDE with an attractive new graphical appearance.**
  - VS 2013 is implemented using WPF.
- **Features in Visual Studio 2013 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 4.5 or .NET 4.5.1.
- **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 Express 2013

- **A noteworthy aspect of Visual Studio 2013 is strong free Express versions of the tool.**
- **In this course we will rely on Visual Studio Express 2013 for Windows Desktop.**
  - It supports multiple language development (C#, Visual Basic, and C++).
  - It supports the creation of WPF projects.
  - It also supports unit testing.



Type: Visual C#

Windows Presentation Foundation client application

Name: WpfApplication1

Location: C:\OIC\WpfCs\Demos\

Solution name: WpfApplication1

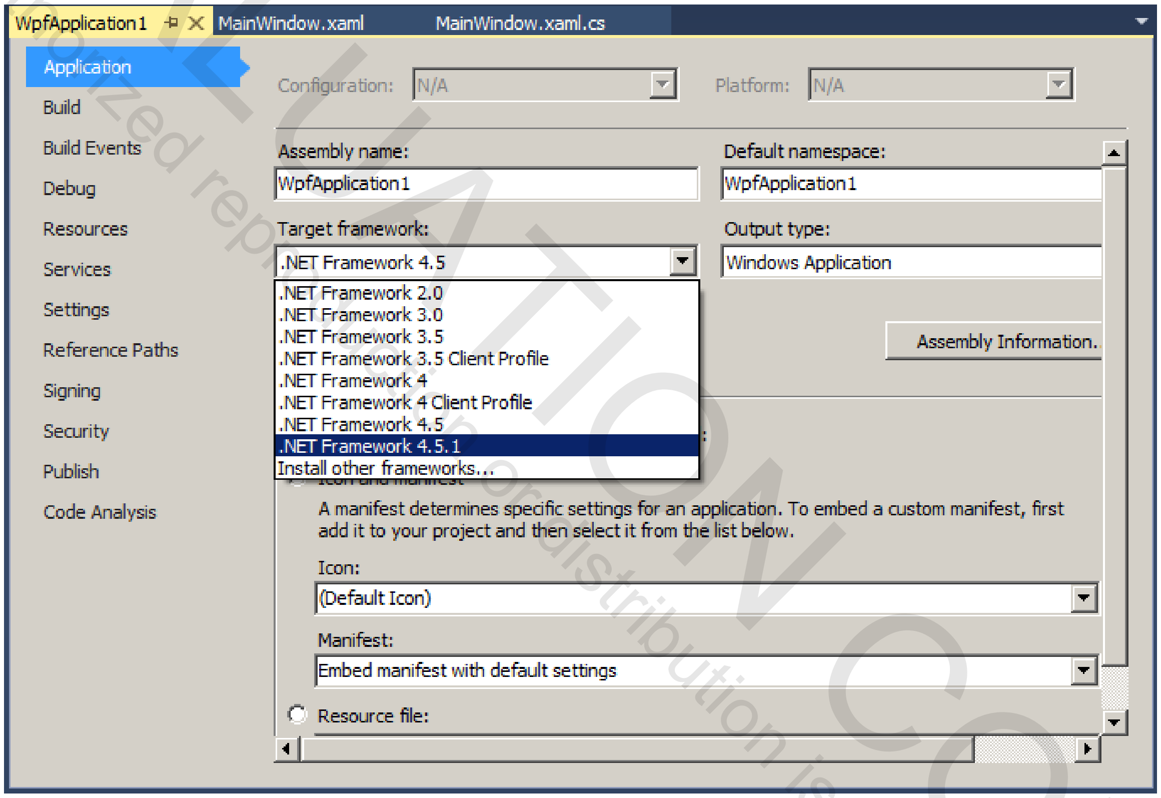
[✓] Create directory for solution

[ ] Add to source control

- **However, the Express edition does lack features present in higher editions, such as support of WCF and WF projects.**

Target Framework

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


WpfApplication1 MainWindow.xaml MainWindow.xaml.cs

Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis

Configuration: N/A Platform: N/A

Assembly name: WpfApplication1
Default namespace: WpfApplication1

Target framework: .NET Framework 4.5
Output type: Windows Application

.NET Framework 2.0
.NET Framework 3.0
.NET Framework 3.5
.NET Framework 3.5 Client Profile
.NET Framework 4
.NET Framework 4 Client Profile
.NET Framework 4.5
.NET Framework 4.5.1
Install other frameworks...

Assembly Information...

Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

Manifest: Embed manifest with default settings

Resource file:

- **Many example programs were originally targeted for an earlier version of the .NET Framework, but will run fine under .NET 4.5.1.**

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 2013 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.

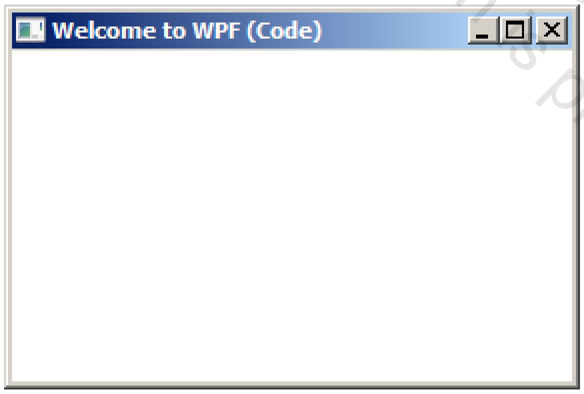
```
using System;
using System.Windows;

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

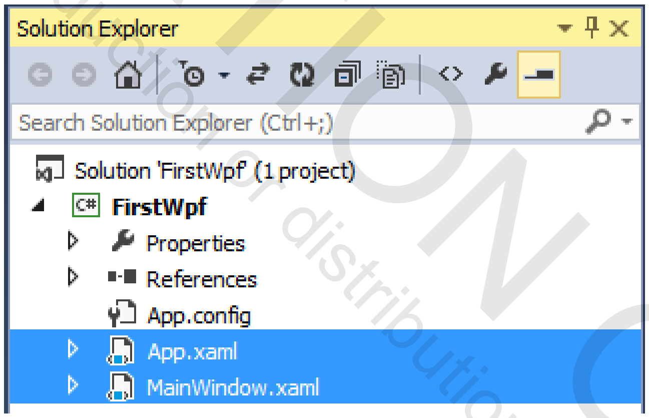
- **Our example program has the following features:**
  - Import the **System.Windows** namespace. This namespace includes the fundamental WPF classes, interfaces, delegates, and so on, including the classes Application and Window.
  - Make your class derive from the **Window** class.
  - Provide the 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 2013

- **Although you can compile WPF programs at the command-line, for simplicity we will use Visual Studio 2013 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 2013!

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

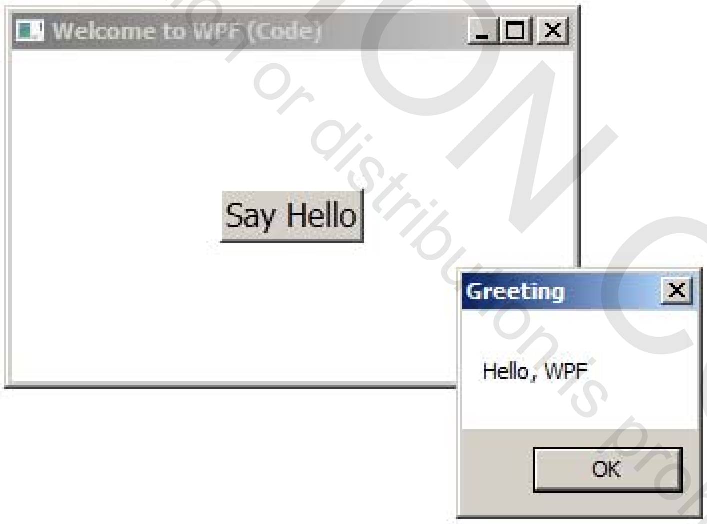
11. Continuing the demo, add the following code to specify an event handler for clicking the button.

```
   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



Hello, WPF

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*.**

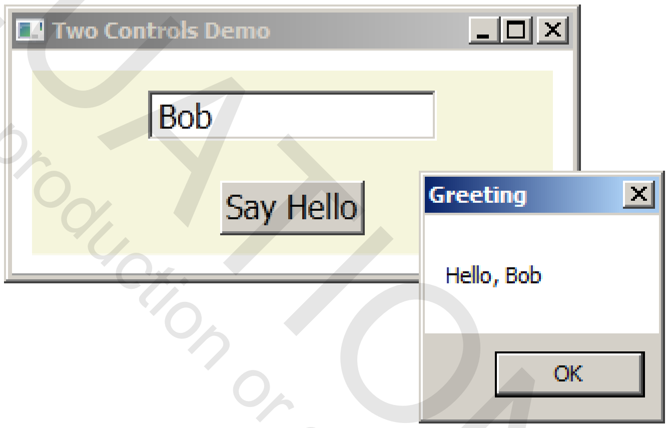
```
StackPanel panel = new StackPanel();
...

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 4.0/4.5.1 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 11

Styles, Templates, Skins and Themes

Styles, Templates, Skins and Themes

Objectives

After completing this unit you will be able to:

- **Understand how to group layout properties together using styles.**
- **Share and restrict styles within the application.**
- **Use typed styles to obtain the effect of a default style for some controls.**
- **Use triggers to improve styles.**
- **Improve user input validation using styles.**
- **Understand how templates are used in WPF controls and how to create your own templates.**
- **Respect the control's properties in the template definition.**
- **Specify a template within a style definition.**
- **Implement skins using style concepts.**
- **Respect themes from the operating system.**

WPF and Interfaces

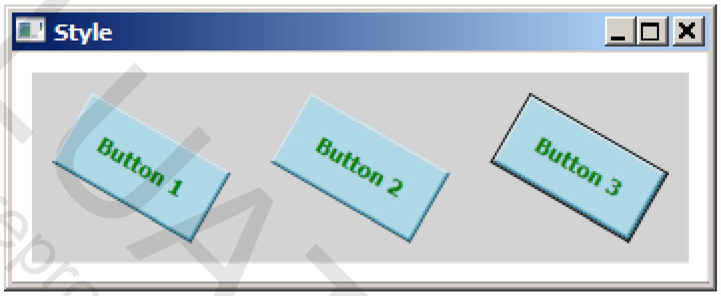
- **WPF is known for its powerful resources for restyling interfaces.**
- **It is possible to change completely the appearance of any control.**
  - This includes controls that are difficult to restyle in HTML such as combo boxes, check boxes and so on.
- **There are four main concepts regarding WPF's restyling support:**
  - Styles
  - Templates
  - Skins
  - Themes
- **We'll deal with each one of these concepts in this chapter.**

Styles

- **WPF comes with a simple mechanism for grouping property values together in a single object.**
  - These property values could be set individually.
  - The main advantage of grouping these values is to reuse them in multiple objects.
- **The *System.Windows.Style* class is responsible for this mechanism.**
- **The main purpose of a style is to separate property values that are related to the control's appearance from the element itself.**
  - This feature is similar to the way that HTML and Cascading Style Sheets (CSS) work together.
- **Other restyling concepts such as templates, skins and themes are based on styles.**

Style Example

- **Let's explain with an example when we can use styles and how to declare them in XAML.**
  - See **StyleDemo\Step1** in the chapter directory.
  - Build and run the application.



- **There are three buttons declared with similar property values.**
  - Most of these property values are responsible for the buttons' appearance.
  - This suggests they can be combined into a style definition.

```
...
<Button Margin="10"
        Background="LightBlue"
        Foreground="Green"
        FontWeight="Bold"
        Height="40"
        Width="80"
        Click="Greet">
   <Button.LayoutTransform>
      <RotateTransform Angle="30" />
   </Button.LayoutTransform>
   ...
```

Style Definition

- **A style can be defined using the properties mentioned in the previous page.**
  - Styles are defined in the Resources collection of some element in XAML code.
  - You can define them in the Window's resources collection, so that they will be visible to all elements in the window.
  - The style properties are defined using setters, which will set the target dependency properties in the elements that have the style applied.

```
<Window.Resources>
   <Style x:Key="buttonStyle">
      <Setter Property="Button.Margin" Value="10"/>
      <Setter Property="Button.Background"
              Value="LightBlue"/>
      <Setter Property="Button.Foreground"
              Value="Green"/>
      <Setter Property="Button.FontWeight"
              Value="Bold"/>
      <Setter Property="Button.Height" Value="40"/>
      <Setter Property="Button.Width" Value="80"/>
      <Setter Property="Button.LayoutTransform">
         <Setter.Value>
            <RotateTransform Angle="30"/>
         </Setter.Value>
      </Setter>
   </Style>
</Window.Resources>
```

Applying Styles

- **To apply the style to an element, just set its *Style* property referencing the key you assigned in the style definition.**

```
<Button Style="{StaticResource buttonStyle}"
        Click="Greet">
   Button 1
</Button>
```

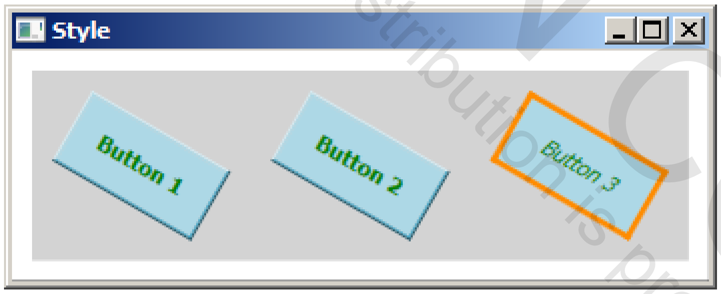
- **See the *StyleDemo\Step2* folder in the chapter directory.**
  - If you build and run the application, you’ll notice that it has the same look as before.
  - However, this version uses a style instead of repeating many property values.
  - The style is reused in all three buttons.
  - The XAML code is cleaner and the logic is better separated from layout.

Style Inheritance

- **You can create styles that inherit from an existing one.**
  - Just use the **BasedOn** property when defining the style and you can add or overwrite style setters.

```
<Style x:Key="specialButtonStyle"
       BasedOn="{StaticResource buttonStyle}">
   <Setter Property="Button.FontWeight"
           Value="Normal"/>
   <Setter Property="Button.FontStyle"
           Value="Italic"/>
   <Setter Property="Button.BorderBrush"
           Value="DarkOrange"/>
</Style>
```

- **See the *StyleInheritance* folder in the chapter directory for an example of the *BasedOn* property usage.**



  - Note that the third button uses the **specialButtonStyle** style, which inherits the **buttonStyle** style.
  - This new style inherits all property values defined in the base style, but it overwrites the **Button.FontWeight** and defines two additional properties, **Button.FontStyle** and **Button.BorderBrush**.

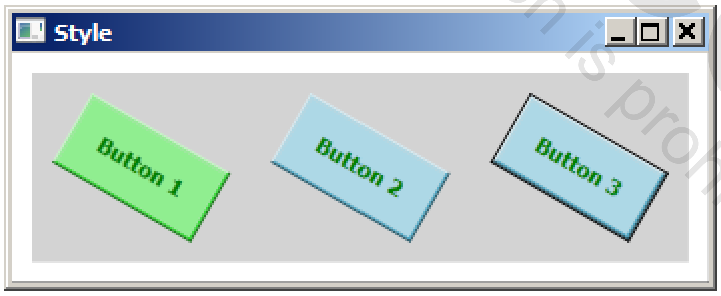
Style Overriding

- **When you apply a style to an element, you can still override a property from the style definition by setting it directly in the element's definition.**

```
<Button Style="{StaticResource buttonStyle}"
        Click="Greet"
        Background="LightGreen">
   Button 1
</Button>
```

- **Because of the order of precedence for dependency properties, the local value set directly in the element overrides the value set in the style definition.**

- **See the *StyleOverride* folder in the chapter directory for an example.**
  - Build and run the application. You'll see that the style is applied to all the three buttons.
  - However, the first button overrides the **Button.Background** property by setting a local value.



Sharing Styles

- **You may have noticed that the property setters in the styles defined so far are fully qualified.**
  - For example, when referencing the Button's **Background** property inside the style definition, we use **Button.Background** as the property name, instead of just **Background**.
  - WPF styles need this full reference to know which dependency property to look for in the element when they are being applied.
- **From this concept we can infer that we could change the Button prefix in the property names by a Control prefix.**
  - Instead of **Button.Background**, we can use **Control.Background**.
  - This way we can use the style in any class that derives from Control.
  - This is what we call *style sharing* between different element types.

Style Sharing Example

- **See the *StyleSharing* folder in the chapter directory for an example of style sharing between heterogeneous elements.**

```
<Style x:Key="shareableStyle">
   <Setter Property="Control.Margin" Value="10"/>
   <Setter Property="Control.Background"
Value="LightBlue"/>
   <Setter Property="Control.Foreground"
Value="Green"/>
   <Setter Property="Control.FontWeight"
Value="Bold"/>
   <Setter Property="Control.Height" Value="40"/>
   <Setter Property="Control.Width" Value="100"/>
   <Setter Property="TextBox.TextAlignment"
Value="Right"/>
   <Setter Property="Control.LayoutTransform">
      <Setter.Value>
          <RotateTransform Angle="30"/>
      </Setter.Value>
   </Setter>
</Style>
```

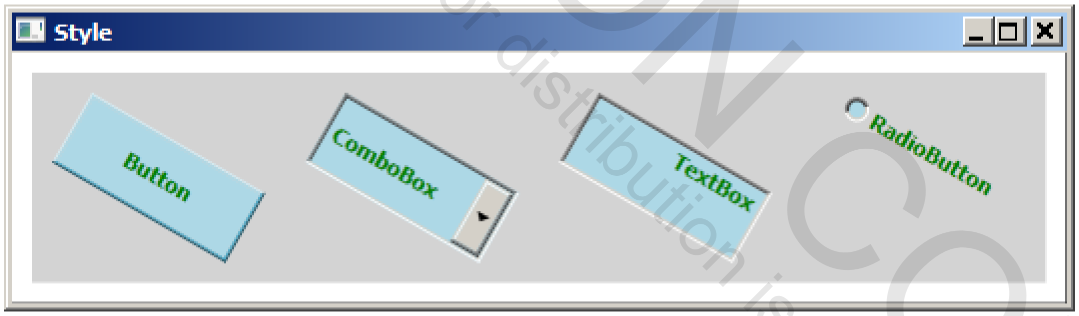
- **Note that the *TextAlignment* property is not defined in the Control class, so it must be referenced as a member of another class, such as TextBox.**

Style Sharing Example (Cont'd)

- **The style is applied to different element types.**

```
<Button Style="{StaticResource shareableStyle}"
        Click="Greet">
   Button
</Button>
<ComboBox Style="{StaticResource shareableStyle}">
   ...
</ComboBox>
<TextBox Style="{StaticResource shareableStyle}">
   TextBox
</TextBox>
<RadioButton Style="{StaticResource
shareableStyle}">
   RadioButton
</RadioButton>
```

- **Build and run the application to see how it looks like.**

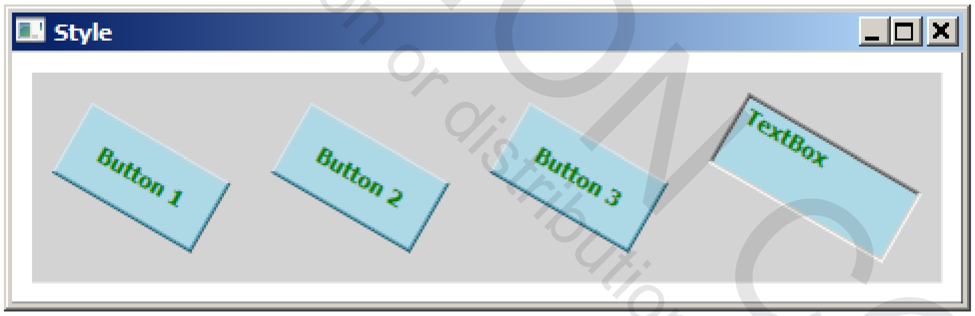


- **Invalid properties are ignored.**
  - As the **TextBox.TextAlignment** property is implemented only in the TextBox, the style setter ignores this property in the other elements.
  - This same behavior occurs whenever a style has a property that does not exist in the element being applied.

Demo: Restricting Styles

- **When defining a style, it is possible to restrict the element types that can use it.**
  - You can set the **TargetType** property to say which element type that can use the style.
  - By using the **TargetType** property, you don't need to prefix the property names anymore.
- **To better understand how we can restrict the usage of a style, we'll modify the solution in the *Demos\StyleRestricted* folder, backed up in the *StyleRestricted\Step1* folder in the chapter directory.**

1. Build and run the solution. You'll see this window:



2. Open the **MainWindow.xaml** file. You'll notice a style definition that is shared among the four controls in the window. Create a new style based on the existing one, to modify the text alignment in the TextBox.

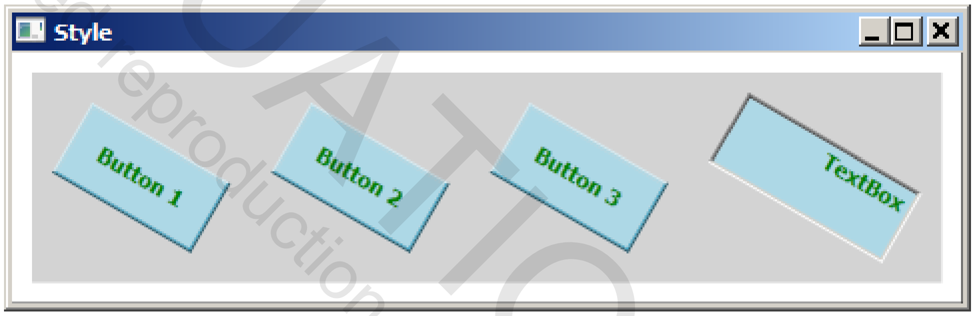
```
<Style x:Key="textboxStyle"
       BasedOn="{StaticResource controlStyle}">
   <Setter Property="TextBox.TextAlignment"
           Value="Right"/>
</Style>
```

Demo: Restricting Styles (Cont'd)

3. Apply the new style to the TextBox.

```
<TextBox Width="100"
         VerticalAlignment="Center"
         Style="{StaticResource textboxStyle}">
   TextBox
</TextBox>
```

4. Build and run. Notice that the text is aligned to the right in the TextBox.



5. Apply this new **textboxStyle** style to the first button too. You won't notice any change in the appearance, since the **TextAlignment** property doesn't exist in the button. Hence, the style setter is ignored.

6. It is possible to avoid the misuse of our newly created style. We can assure that the style will be used only in text boxes by setting its **TargetType** property. By doing that, we can remove the TextBox prefix from the setter in the style definition.

```
<Style x:Key="textboxStyle"
       BasedOn="{StaticResource controlStyle}"
       TargetType="{x:Type TextBox}">
   <Setter Property="TextAlignment"
           Value="Right"/>
</Style>
```

Demo: Restricting Styles (Cont'd)

7. Build the application. Notice that it doesn't build successfully anymore. That's because the **textboxStyle** is being used in the first button, which is not allowed. Change the style of that button to **controlStyle** again.

8. Build and run the application. Now it works, and you have successfully created a separate style for text boxes in the application.

- **You could do the same thing creating a new separate style for the buttons.**

- **The working demo is saved in the *StyleRestricted\Step2* folder in the chapter directory.**

Typed Styles

- **There is an additional feature of the *TargetType* property in the style definition.**
  - If you do not set a key for the style, but you set the **TargetType** property, the elements of that type will have the style applied implicitly unless they have their **Style** property set locally.
  - This is valid only for the scope of the Resources collection in which the style is defined.
- **The styles that don't have a name set in the *x:Key* property but have the *TargetType* property set are called *typed styles*.**
  - The styles that have the **x:Key** property set are called *named styles*.

Typed Style Example

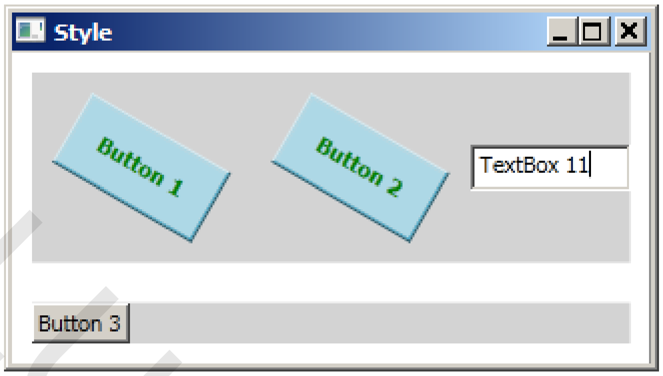
- **Open the solution in the *TypedStyle* folder in the chapter directory.**
- **Open the *MainWindow.xaml* file. Notice that the interface is built based on three StackPanels.**
  - The first one is just for grouping the other two panels.
  - The second one is the upper panel in the window, and it contains two buttons and a text box. Notice that this StackPanel has a Resources collection with a typed style defined.

```
<StackPanel.Resources>
   <Style TargetType="{x:Type Button}">
      <Setter Property="Button.Margin" Value="10"/>
      <Setter Property="Button.Background"
              Value="LightBlue"/>
      <Setter Property="Button.Foreground"
              Value="Green"/>
      <Setter Property="Button.FontWeight"
              Value="Bold"/>
      <Setter Property="Button.Height" Value="40"/>
      <Setter Property="Button.Width" Value="80"/>
      <Setter Property="Button.LayoutTransform">
         <Setter.Value>
            <RotateTransform Angle="30"/>
         </Setter.Value>
      </Setter>
   </Style>
</StackPanel.Resources>
```

  - The third StackPanel is the bottom panel in the window, and it has only a simple button.

Typed Style Example (Cont'd)

- **Build and run the application to see how it looks.**



- **Note that the typed style was applied only in the two buttons that are in the scope of the upper StackPanel.**
  - The text box, despite being in the scope, does not get the style because it isn't a Button.
  - The Button 3, despite being a Button, does not get the style because it isn't on the upper StackPanel's scope.
- **If we move the style definition to the Window's Resources collection, Button 3 will have the style implicitly applied to it.**

Triggers

- **Triggers define a collection of setters just like styles do, but their setters are applied based on some conditions.**
- **The conditions can be set using dependency properties or plain .NET properties, depending on the trigger type.**
  - Property triggers have their conditions set with dependency property values.
  - Data triggers have their conditions set with plain .NET property values.
- **The property trigger checks a dependency property to see if it has a specific value.**
  - When the value matches the condition, the trigger executes the setters.
  - When the value changes again, the trigger "undoes" the setters automatically.
- **Data triggers use binding to check the value of a plain .NET property.**
  - Then the process of matching the condition works the same as with property triggers.

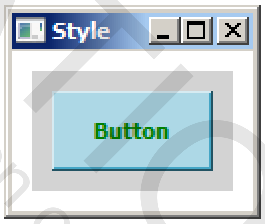
Property Trigger Example

- **Open the solution in the *StylePropertyTrigger* folder in the chapter directory.**
- **Open the file *MainWindow.xaml* and analyze the *Style* declaration.**

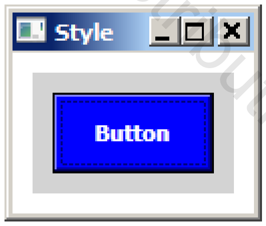
```
<Style x:Key="buttonStyle"
       TargetType="{x:Type Button}">
   <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
         <Setter Property="Background"
                 Value="Blue"/>
         <Setter Property="Foreground"
                 Value="White"/>
      </Trigger>
   </Style.Triggers>
   <Setter Property="Margin" Value="10"/>
   <Setter Property="Background"
           Value="LightBlue"/>
   <Setter Property="Foreground" Value="Green"/>
   <Setter Property="FontWeight" Value="Bold"/>
   <Setter Property="Height" Value="40"/>
   <Setter Property="Width" Value="80"/>
</Style>
```

Property Trigger Example (Cont'd)

- **The Style has a property trigger that depends on the *Button.IsMouseOver* property.**
  - When the property has the True value, the trigger will set the background to blue and the foreground to white.
  - When the property changes to False, the trigger will undo these setters.
- **Build and run the application to see the behavior.**
  - Before the mouse is over the button:



  - After the mouse is over the button:



Data Trigger Example

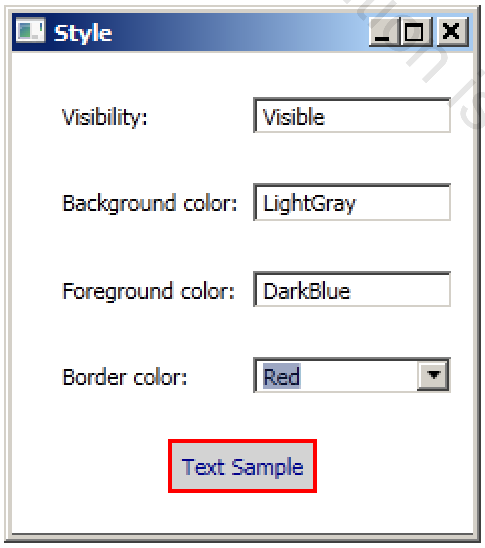
- **Open the solution in the *StyleDataTrigger* folder in the chapter directory.**
- **Open the file *MainWindow.xaml* and analyze the *Style* declaration.**

```
<Style x:Key="labelStyle"
       TargetType="{x:Type Label}">
   <Style.Triggers>
      <DataTrigger Binding="{Binding
            ElementName=txtVisibility, Path=Text}"
            Value="Hidden">
         <Setter Property="Visibility"
                 Value="Hidden"/>
      </DataTrigger>
   </Style.Triggers>
   <Setter Property="Background"
           Value="{Binding
              ElementName=txtBackgroundColor,
              Path=Text}"></Setter>
   <Setter Property="Foreground"
           Value="{Binding
              ElementName=txtForegroundColor,
              Path=Text}"></Setter>
   <Setter Property="BorderBrush"
           Value="{Binding
              ElementName=cmbBorderColor,
              Path=Text}"></Setter>
</Style>
```

- **This example shows how a style can dynamically modify the appearance of a control, which is a Label in this case.**

Data Trigger Example (Cont'd)

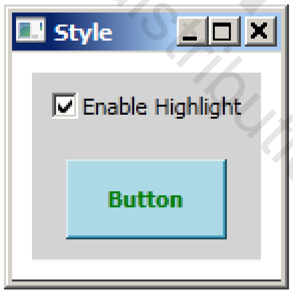
- **The style has a data trigger that depends on the value of the *Text* property of the *txtVisibility* text box.**
  - Actually, there's only one value that invokes this trigger: "Hidden". Try typing this value in the **Visibility** text box to see the setter in action.
- **The other setters within the style show an additional feature that you can use to have the setters working dynamically.**
  - There are setters changing the background, foreground and the border of the label.
  - These setters have their values bound to the appropriate input control in the interface, so that the user can choose the colors dynamically.
- **Build and run the application to see these features.**
  - We've typed in some different values.



Visibility: Visible
Background color: LightGray
Foreground color: DarkBlue
Border color: Red

Multiple Conditions

- **It is possible to obtain a logical OR condition if you define more than one trigger in the style.**
  - To achieve this, we just have to create two triggers with the same setters, but different conditions.
- **But how can we handle a logical AND condition?**
- **That's why WPF provide us with two additional trigger classes:**
  - **MultiTrigger**, for property triggers with multiple conditions.
  - **MultiDataTrigger**, for data triggers with multiple conditions
- **You can see an example of the *MultiDataTrigger* usage in the *StyleMultiDataTrigger* folder in the chapter directory.**



- **The trigger will be invoked only if both conditions are satisfied: the mouse is over the button, and the CheckBox is checked.**
  - See **MainWindow.xaml** for the **MultiDataTrigger** syntax.

Validation

- **We can combine a property trigger with a binding validation rule to have a more complicated style example.**
- **See the *Validate* folder in the chapter directory.**
  - The style is defined to be applied to all text boxes in the window every time the **Validation.HasError** property is true.
  - An additional tweak is done by setting the **ToolTip** property of the text box with the first error message returned by the validator.

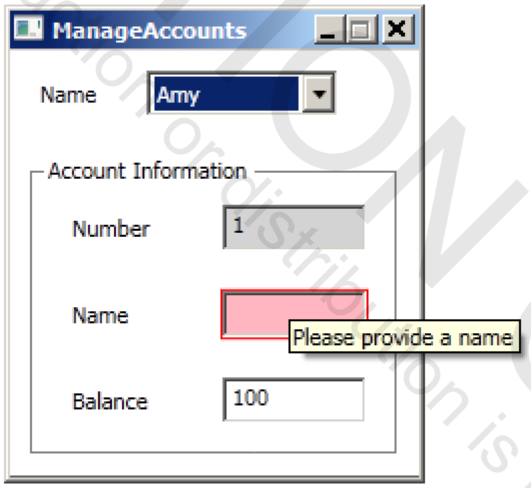
```
<Style TargetType="{x:Type TextBox}">
   <Style.Triggers>
      <Trigger Property="Validation.HasError"
               Value="true">
         <Setter Property="ToolTip"
                 Value="{Binding
RelativeSource={x:Static RelativeSource.Self},
Path=(Validation.Errors)[0].ErrorContent}"/>
         <Setter Property="Background"
                 Value="LightPink"/>
      </Trigger>
   </Style.Triggers>
</Style>
```

Validation Example

- **A validation rule is added to each editable text box.**

```
<TextBox Name="txtName" Margin="10" Width="72">
   <Binding ElementName="cmbAccounts"
            Path="SelectedItem.Name">
      <Binding.ValidationRules>
         <local:NameValidationRule/>
      </Binding.ValidationRules>
   </Binding>
</TextBox>
```

- **Then, if we try to enter invalid data in some of the text boxes, the validation rule will fail and the style trigger will be executed.**



Please provide a name

- **The following rules are being validated:**
  - Name must be provided and the maximum length is 10.
  - Balance must be an integer number.
- **Validation occurs at time of dropping down the list box.**

Templates

- **WPF allows us to completely replace a control’s visual tree by using a *template*.**
  - Actually, this concept is so important that every control in WPF comes with its default visual tree defined in a template.
  - These templates are also called *control templates*.
- **Here is how a template definition looks like.**

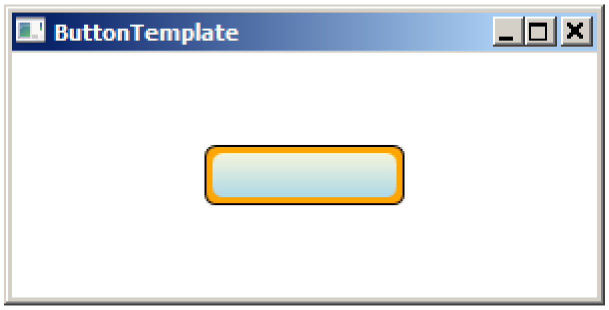
```
<ControlTemplate x:Key="buttonTemplate">
   <Grid>
      <Rectangle Height="30"
                 Stroke="Black"
                 Width="100"
                 RadiusX="5"
                 RadiusY="5"
                 Fill="Orange">
      </Rectangle>
      <Rectangle Height="22"
                 Width="92"
                 RadiusX="5"
                 RadiusY="5">
         <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0,0"
                                 EndPoint="0,1">
               <GradientStop Offset="0"
                             Color="Beige"/>
               <GradientStop Offset="1"
                             Color="LightBlue"/>
            </LinearGradientBrush>
         </Rectangle.Fill>
      </Rectangle>
   </Grid>
</ControlTemplate>
```

A Simple Template Example

- **Open the solution in the *SimpleControlTemplate* folder in the chapter directory.**
  - Note that the template is defined as a resource in the **MainWindow.xaml** file.
- **Applying a template is similar to applying a style.**

```
<Button Margin="10"
        Click="Button_Click"
        Template="{StaticResource buttonTemplate}">
   My Button
</Button>
```

- **The template replaces completely the default visual tree of the Button.**
  - The entire Button's visual implementation is replaced by our template.
  - The shades, content text and pressing effect don't exist anymore for this button, since we didn't implement these features in this new simple template.
  - However, note that the click event handler is still working, since this is not part of the visual tree.



Improving the Template

- **As with styles, it's possible to use all sorts of triggers in the template definition.**
  - We can use a trigger to implement the hover and pressing effects in the previous example.
- **Open the example in the *TemplateTrigger* folder in the chapter directory.**

```
<ControlTemplate x:Key="buttonTemplate"
                 TargetType="{x:Type Button}">
   ...
   <ControlTemplate.Triggers>
      <Trigger Property="IsMouseOver"
               Value="True">
         <Setter TargetName="outerBorders"
                 Property="StrokeThickness"
                 Value="2"/>
      </Trigger>
      <Trigger Property="IsPressed"
               Value="True">
         <Setter Property="RenderTransform">
            <Setter.Value>
               <ScaleTransform ScaleX=".9"
                               ScaleY=".9"/>
            </Setter.Value>
         </Setter>
         <Setter Property="RenderTransformOrigin"
                 Value=".5,.5"/>
      </Trigger>
   </ControlTemplate.Triggers>
</ControlTemplate>
```

- **Run the application, and then try passing the mouse over and clicking the button to see the result.**

Templated Parent's Properties

- **The problem with the previous template examples is that they don't respect some control properties.**
  - For example, they don't show the Content that was set for the Button.
- **We must provide implementation to respect such properties.**
  - We can do this through data binding, optionally using special lightweight class named **TemplateBindingExtension**.
  - Here is an example of how we can include the Button's Content property inside the template definition.

```
<TextBlock Margin="10"
   Text="{TemplateBinding Button.Content}"/>
```

- **But what if the Content is not a text value?**
  - In this situation, a more generic solution should be used.
  - WPF provides a special class for this purpose, named **ContentPresenter**.

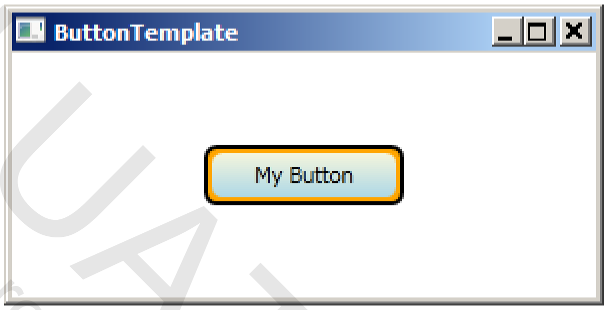
```
<ContentPresenter Margin="10"
   Text="{TemplateBinding Button.Content}"/>
```

- **By design, *ContentPresenter* has a built-in shortcut that allows omitting the binding in case you just want to show the Content property.**

```
<ContentPresenter Margin="10" />
```

Respecting Properties Example

- **An example of the *ContentPresenter* usage is saved on the *RespectingProperties\Step1* folder in the chapter directory.**
  - Build and run the application to see the Button's Content.



- **Other properties such as *Height*, *Width*, *Background* and *Padding* should be respected by the template too.**
  - In our example, the border is built by placing one rectangle on top of another bigger rectangle, resulting in a double border effect.
  - To respect **Height** and **Width**, we'll bind these properties from the Button to the outer rectangle, and implement a value converter to have a lower size in the inner rectangle.
  - The difference between the size of the outer and inner rectangles is responsible for the border thickness.
  - A similar example is implemented in detail in this chapter's lab.

Respecting Properties (Cont'd)

- **Here is a new version of the rectangles respecting *Height* and *Width* from the Button.**
  - Open the solution in the **RespectingProperties\Step2** folder in the chapter directory.

```
<Rectangle Name="outerBorders"
   Height="{Binding RelativeSource={RelativeSource
      TemplatedParent}, Path=Height}"
   Stroke="Black"
   Width="{Binding RelativeSource={RelativeSource
      TemplatedParent}, Path=Width}"
   RadiusX="5"
   RadiusY="5"
   Fill="Orange">
</Rectangle>
<Rectangle Name="innerBorders"
   Width="{Binding RelativeSource={RelativeSource
      TemplatedParent}, Path=Width,
      Converter={StaticResource sizeConverter},
      ConverterParameter=4}"
   Height="{Binding RelativeSource={RelativeSource
      TemplatedParent}, Path=Height,
      Converter={StaticResource sizeConverter},
      ConverterParameter=4}"
   RadiusX="5"
   RadiusY="5">
   ...
```

- **Notice the usage of the value converter.**
  - It is responsible for decreasing the **Height** and **Width** to give the thickness idea of the double border.
  - Examine the converter code in the **SizeConverter.cs** file.

Respecting Properties (Cont'd)

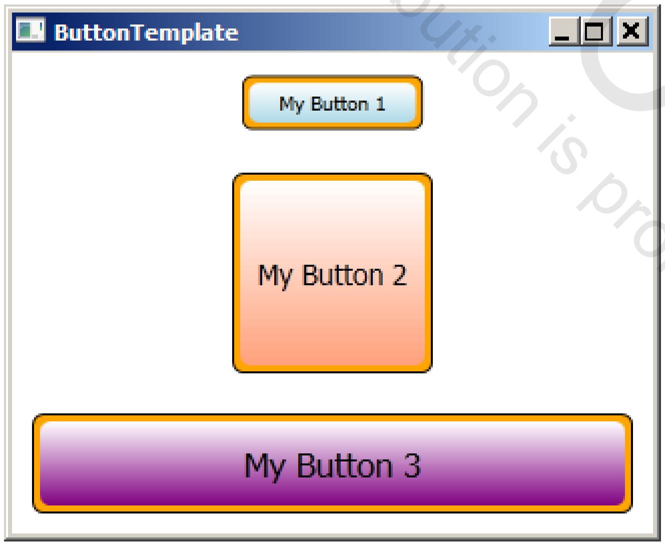
- **The *Background* property is respected too.**
  - It is implemented through the background gradient in the inner rectangle.

```
<GradientStop Offset="1" Color="{Binding
   RelativeSource={RelativeSource TemplatedParent},
   Path=Background.Color}"/>
```

- **The template implements the Button's *Padding* property by setting the ContentPresenter's *Margin*.**
  - In fact, this is exactly the definition of padding: the margin of an inner element.

```
<ContentPresenter
   Margin="{Binding RelativeSource={RelativeSource
      TemplatedParent}, Path=Padding}"
   ...
```

- **Look how our Button looks now with different sizes.**



Respecting Visual States

- **In addition to respecting the control properties, it's important to consider visual states too.**
  - Properties such as **IsEnabled** and **IsDefaulted** can modify the visual state, which implies layout changes.
  - A progress bar, for example, must implement the visual representation of the **Value** property to give the idea of progress.
- **Let's add to our example the support for the *IsEnabled* property.**
  - Open the solution in the **RespectingProperties\Step3** folder in the chapter directory.
  - Notice that we respect the Disabled visual state just by using triggers!

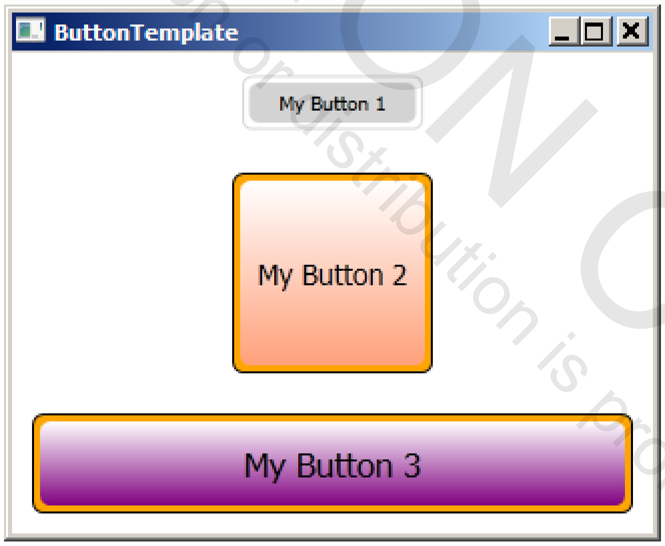
```
<Trigger Property="IsEnabled"
         Value="False">
   <Setter TargetName="outerBorders"
           Property="Stroke"
           Value="Orange">
   </Setter>
   <Setter TargetName="outerBorders"
           Property="Rectangle.Fill"
           Value="White">
   </Setter>
   <Setter TargetName="innerBorders"
           Property="Rectangle.Fill"
           Value="LightGray">
   </Setter>
</Trigger>
```

Respecting Visual States Example

- **One of the buttons has its *IsEnabled* property set to false.**

```
<Button Margin="10"
        Click="Button_Click"
        Width="100"
        Height="30"
        Background="LightBlue"
        Padding="10"
        IsEnabled="False"
        Template="{StaticResource buttonTemplate}">
   My Button 1
</Button>
```

- **And this is how our disabled visual state looks like for Button 1.**



Using Templates with Styles

- **It's more likely that you'll define templates inside a style definition.**
  - The template examples seen so far were set directly in the controls for the sake of simplicity.
  - To use a template inside a style, just add the template definition to the style and then apply the style to the control.
- **There are some advantages in defining a template within a style.**
  - We can set additional properties in the style to make the template's appearance more attractive.
  - When used along with typed styles we can obtain the effect of a default template, since it will be applied to all controls of that type by default.
  - We can provide default values for properties that modify the look of the template, yet allowing these values to be overridden in the control definition.

Templates with Styles Example

- **Open the solution in the *StyleAndTemplate* folder in the chapter directory.**
  - Note that in the file **MainWindow.xaml** we use a setter to modify the template property.

```
<Style TargetType="{x:Type Button}">
   <Setter Property="Background"
           Value="LightBlue"/>
   <Setter Property="Template">
      <Setter.Value>
         <ControlTemplate
            TargetType="{x:Type Button}">
            ...
            <ControlTemplate.Triggers>
               ...
            </ControlTemplate.Triggers>
         </ControlTemplate>
      </Setter.Value>
   </Setter>
</Style>
```

- **There are two buttons defined in the window.**
  - Both buttons get the template because of the typed style.
  - Note that one of them doesn't set the **Background** property, but gets the LightBlue background because of the typed style.
  - The other button overrides the style background by setting a different brush to the **Background** property locally.
  - Build and run the application to see the result.

Skins

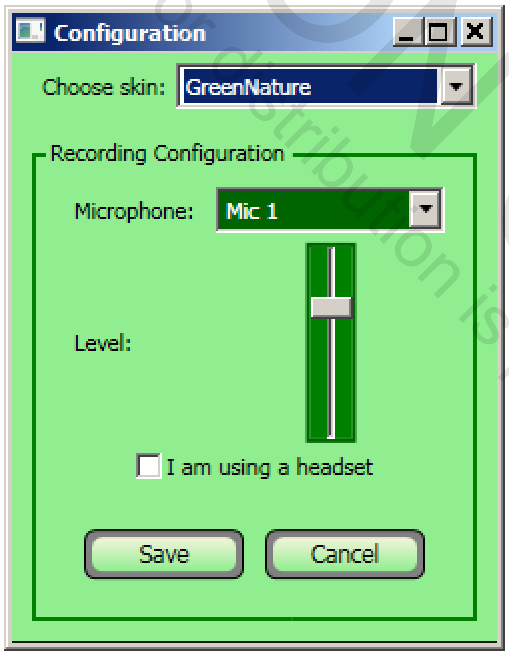
- ***Skins* are a set of predefined appearance changes that we can apply to the application on the fly.**
- **There is no formal definition for skins in WPF.**
  - However, WPF doesn't need such definition as it provides features to obtain skin functionality.
  - Basically, skins are implemented by using dynamic resources and styles.
- **First of all, the styles used in the window must be set using the DynamicResource markup.**
  - It's good practice to use styles in most of the items, since it gives the skin authors more control over the visual experience.

```
<StackPanel
   Style="{DynamicResource ButtonsPanelStyle}">
```

- **One good idea of skins implementation using WPF relies on creating separate XAML files with styles and/or templates definition for each skin.**
  - Each XAML file is a resource repository.
  - We can achieve this by creating the files with a **ResourceDictionary** as the root element.
  - The default application skin should be defined in the application resources dictionary, in **App.xaml**.

Changing Skins

- **To change a skin, we do the following:**
  - Read an XAML file with the resource dictionary that contains the styles and templates.
  - Replace the **Application.Resources** collection with this new resource dictionary.
- **As an example, open the solution in the *SkinSelector* folder in the chapter directory.**
  - This is a hypothetical window for audio configuration.
  - Build and run the application, and notice the ComboBox for changing the skin on the top.
  - Try changing the skin to see it working.



Skins Example

- **Let's examine the code to understand how it works.**
  - Notice in the **Configuration.xaml** file that many items have their styles set using the DynamicResource markup.
  - Examine the **App.xaml** file and check the default styles definition. These styles are used when the application starts.
- **Skin files are provided.**
  - Notice the files **GreenNature.xaml** and **SpecialGray.xaml**, which contains a **ResourceDictionary** with the same styles of the **App.xaml** file, but with different property values.
- **The GreenNature skin shows an example of changing many visual properties such as fonts and colors.**
- **The SpecialGray skin shows an example of changing the structures more deeply.**
  - Notice that the template for the GroupBox provided in GroupBoxStyle has a completely different visual tree from the original control.
  - A StackPanel is responsible for the new appearance of the GroupBox, with a Label being used to show the Header text.

Skins Example (Cont'd)

- **Changing the skin is done by the method *cmbSkin_SelectionChanged()*.**

```
ResourceDictionary newResources = null;
ComboBoxItem selectedSkin =
   cmbSkin.SelectedItem as ComboBoxItem;

if (selectedSkin.Content != null)
{
   if (selectedSkin.Content.ToString() !=
"Default")
   {
      FileStream newResourceFile = new FileStream(
         selectedSkin.Content.ToString() + ".xaml",
         FileMode.Open,
         FileAccess.Read);
      newResources =
         XamlReader.Load(newResourceFile) as
         ResourceDictionary;
      newResourceFile.Close();
      newResourceFile = null;
   }
   else
   {
      newResources = defaultResources;
   }
   Application.Current.Resources = newResources;
}
```

- **This approach is interesting because it allows the skin authors to edit the skin files independently.**
  - There is no need to rebuild the application.

Themes

- **A theme has to do with respecting visual styles from the operating system.**
  - However, when a designer develops custom templates or skins, usually she is trying to avoid the OS themes.
  - Despite that, it is a nice touch to have the default control templates matching the OS appearance.
- **We can use the technique discussed in the skins section to implement themes.**
  - Despite being flexible, this is a more complex approach because we'd need to implement a separate resource file for each theme.
- **A simpler approach would be using .NET classes to obtain some OS theme properties and apply them to our styles.**
  - The classes **SystemColors**, **SystemFonts** and **SystemParameters** allow us to get colors, fonts and parameters being used in the OS theme.
  - These classes get updated when the OS theme changes.

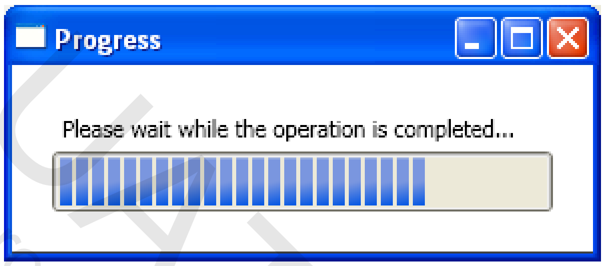
Themes Example

- **Open the solution in the *ProgressTheme* folder in the chapter directory.**
  - The application is a simple window containing a ProgressBar simulating some processing.
  - There is a typed style for the ProgressBar in the **MainWindow.xaml** file.

```
<Style TargetType="{x:Type ProgressBar}">
   <Style.Resources>
      <LinearGradientBrush x:Key="foregroundBrush"
         StartPoint="0,0" EndPoint="1,1">
         <GradientStop Offset="0"
            Color="{DynamicResource {x:Static
SystemColors.InactiveCaptionColorKey}}"/>
         <GradientStop Offset="0.5"
            Color="{DynamicResource {x:Static
SystemColors.InactiveCaptionColorKey}}"/>
         <GradientStop Offset="1"
            Color="{DynamicResource {x:Static
SystemColors.ActiveCaptionColorKey}}"/>
      </LinearGradientBrush>
   </Style.Resources>
   <Setter Property="Foreground"
      Value="{StaticResource foregroundBrush}"/>
   <Setter Property="Background"
      Value="{DynamicResource {x:Static
SystemColors.ControlBrushKey}}"/>
</Style>
```

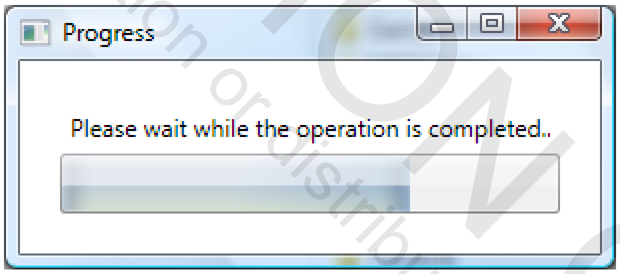
Themes Example (Cont'd)

- **If we run the application in different OS themes, we'll get different results based on the *SystemColors* class properties.**
- **Here is the result of running in Windows XP with the default theme.**



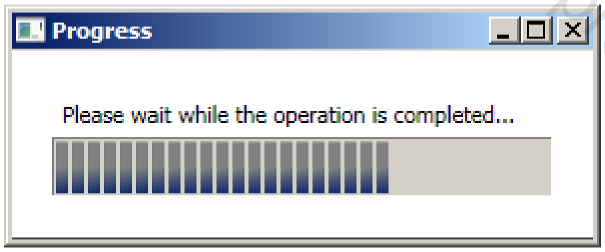
Please wait while the operation is completed...

- **And now in Windows Vista with the Aero Theme.**



Please wait while the operation is completed..

- **Finally, running in Windows 7 with the Windows Classic Theme.**

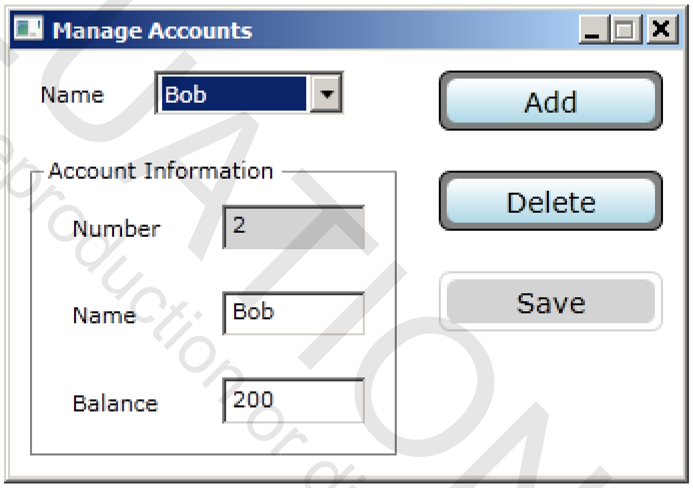


Please wait while the operation is completed...

Lab 11

Improving the Account Manager with Styles

In this lab you will improve the Account Manager developed in the Data Binding chapter by using styles. From the starter code, you'll need to create typed styles for the WPF controls used in the interface, create a template for the buttons and enrich the user experience by adding validation support and feedback.



Detailed instructions are contained in the Lab 11 write-up at the end of the chapter.

Suggested time: 90 minutes

Summary

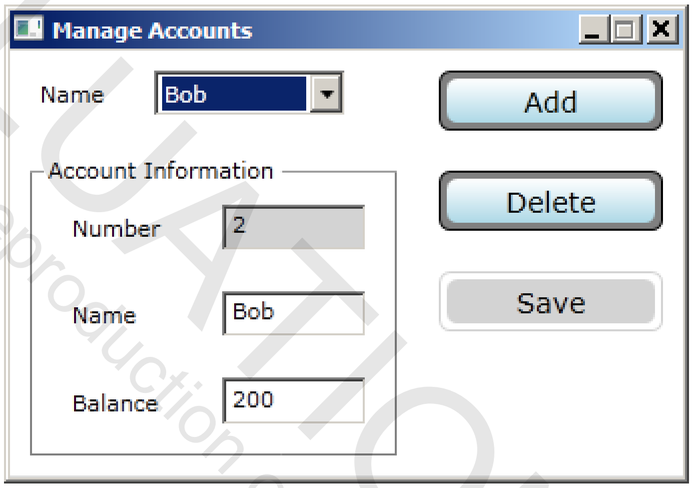
- **Styles can be used to group layout properties together.**
- **Typed styles are used to obtain the effect of a default style for some controls.**
- **Styles can be improved using triggers.**
- **Styles can be used to improve user input validation.**
- **Templates are used in WPF controls to store their default representation and to customize them.**
- **There are means of respecting a control's properties when defining a custom template.**
- **Template can be defined within a style definition.**
- **Skins are predefined changes in the application appearance applied on the fly.**
- **Themes have to do with respecting visual styles from the operating system.**

Lab 11

Improving the Account Manager with Styles

Introduction

In this lab you will improve the Account Manager developed in the Data Binding chapter by using styles. From the starter code, you'll need to create typed styles for the WPF controls used in the interface, create a template for the buttons and enrich the user experience by adding validation support and feedback.



Suggested Time:	90 minutes	
Root Directory:	**OIC\WpfCs**	
Directories:	**Labs\Lab11\AccountManager**	(do your work here)
	Chap10\AccountManager\Step3	(backup of starter code)
	Chap11\AccountManager\Step4	(answer to part 1)
	Chap11\AccountManager\Step5	(answer to part 2)
	Chap11\AccountManager\Step6	(answer to part 3)

Part 1. Grouping Properties in Style Definitions

1. Build and run the starter code. There is a ComboBox for account selection, three text boxes for showing account information, and three action buttons for account management. You can add, remove and modify account data, and save the data back to the XML source using the Save button.

2. The first thing you should do to start using styles in this program is to create typed styles for each WPF control in the Window, making it easier for the style designer to control the application layout. Let's add to the **Window.Resources** collection one style for each existing WPF control in the Window: ComboBox, Label, TextBox, GroupBox and Button. If we wanted to customize the layout of the StackPanels in this example, we could add styles for them too, but this is not the case.

```
<Window.Resources>
   <XmlDataProvider x:Key="dataProvider"
                    XPath="Accounts"
                    Source="AccountsData.xml"/>
   <Style TargetType="{x:Type Label}">
   </Style>
   <Style TargetType="{x:Type ComboBox}">
   </Style>
   <Style TargetType="{x:Type TextBox}">
   </Style>
   <Style TargetType="{x:Type GroupBox}">
   </Style>
   <Style TargetType="{x:Type Button}">
   </Style>
</Window.Resources>
```

3. For each of these controls, let's analyze the common properties that affect the appearance of the control used in the XAML code and group them in the style definition. Let's start with the Label: if you take a look at its four occurrences in the code, you'll notice the **Margin** property being repeatedly used, so this is a nice candidate to be in the style definition. Add a setter in the Label's typed style for the **Margin** property setting its value to 10.

```
<Style TargetType="{x:Type Label}">
   <Setter Property="Margin" Value="10"/>
</Style>
```

4. Now, remove the **Margin** property from the four occurrences of Label in the code, because you don't need them anymore.

5. Build and run the application. Notice that your changes didn't modify the application's appearance, but now the code is more organized with the shared layout properties defined inside styles.

6. For each one of the controls listed in step 2, repeat what you did on steps 3 to 5 to group the layout properties. Don't forget to remove the properties set locally on each control, as they are now defined in the style. After your changes, the typed styles for these controls should look like this:

```
<Style TargetType="{x:Type Label}">
   <Setter Property="Margin" Value="10"/>
</Style>
<Style TargetType="{x:Type ComboBox}">
   <Setter Property="Margin" Value="10"/>
</Style>
```

```
<Style TargetType="{x:Type TextBox}">
   <Setter Property="Margin" Value="10"/>
</Style>
<Style TargetType="{x:Type GroupBox}">
   <Setter Property="Margin" Value="10"/>
</Style>
<Style TargetType="{x:Type Button}">
   <Setter Property="FontSize" Value="14"/>
   <Setter Property="Margin" Value="10"/>
</Style>
```

7. Now that we have separated the layout properties into these new styles, it's easy to modify all controls in the window by changing these styles. For example, let's use Verdana as the default font for all controls used in our window. Add a setter for the **FontFamily** property with the value Verdana for all typed styles.

```
<Style TargetType="{x:Type Label}">
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
</Style>
<Style TargetType="{x:Type ComboBox}">
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
</Style>
<Style TargetType="{x:Type TextBox}">
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
</Style>
<Style TargetType="{x:Type GroupBox}">
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
</Style>
<Style TargetType="{x:Type Button}">
   <Setter Property="FontSize" Value="14"/>
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
</Style>
```

8. Build and run the application to see the result. In addition, you can think of other layout properties to add to some or all of these typed styles, in order to get a nice customized window. Don't change the typed style for the Button now, since we'll change it on the next steps.

Part 2. Creating a Template for the Buttons

1. Let's add a template to the Button's typed style definition. Our goal is to obtain a button with a double border effect, and with a gradient background. To obtain the double border effect, create a rectangle overlapping another slightly bigger rectangle, and the smaller one will be filled by a gradient. For now, set the sizes of the rectangles manually, and set the **Name** property of the rectangles so that they can be used later as a reference in this program.

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

```
   <Setter Property="FontSize" Value="14"/>
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
   <Setter Property="Template">
      <Setter.Value>
         <ControlTemplate TargetType="{x:Type Button}">
            <Viewbox>
               <Grid>
                  <Rectangle Name="outerBorders"
                             Height="30"
                             Width="112"
                             Stroke="Black"
                             RadiusX="5"
                             RadiusY="5"
                             Fill="Gray">
                  </Rectangle>
                  <Rectangle Name="innerBorders"
                             Width="104"
                             Height="22"
                             RadiusX="5"
                             RadiusY="5">
                     <Rectangle.Fill>
                        <LinearGradientBrush StartPoint="0,0"
                                             EndPoint="0,1">
                           <GradientStop Offset="0"
                                         Color="White"/>
                           <GradientStop Offset="1"
                                         Color="LightBlue"/>
                        </LinearGradientBrush>
                     </Rectangle.Fill>
                  </Rectangle>
               </Grid>
            </Viewbox>
         </ControlTemplate>
      </Setter.Value>
   </Setter>
</Style>
```

2. Build and run the application. You’ll notice that the layout of the buttons was replaced, but many properties were lost, including the text. Let’s fix the text first by using the **ContentPresenter** control.

```
<ControlTemplate TargetType="{x:Type Button}">
   <Viewbox>
      <Grid>
         ...
         </Rectangle>
         <ContentPresenter HorizontalAlignment="Center"
                           VerticalAlignment="Center"/>
      </Grid>
   </Viewbox>
</ControlTemplate>
```

3. In this new template, the button's **Width** and **Height** can be represented by the outer rectangle's **Width** and **Height**. Modify the values of these properties to bind them to the button's properties.

```
<Rectangle Name="outerBorders"
           Height="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Height}"
           Width="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Width}"
           Stroke="Black"
           RadiusX="5"
           RadiusY="5"
           Fill="Gray">
</Rectangle>
```

4. To create the double border effect, we'll bind the inner rectangles **Width** and **Height** properties to the button's properties, but we'll need a value converter to make them smaller. Right-click the project in Solution Explorer, select Add and then New Item… to add a class file named **SizeConverter.cs**. Provide the code below in this file, and import the namespaces **System.Windows.Data** and **System.Globalization**.

```
public class SizeConverter : IValueConverter
{
   public object Convert(object value, Type targetType, object
parameter, CultureInfo culture)
   {
      double num = Double.Parse(value.ToString());
      double thickness = Double.Parse(parameter.ToString());

      return (num - (2 * thickness));
   }
   public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
   {
      throw new NotSupportedException();
   }
}
```

5. Add this value converter as a resource in **MainWindow.xaml**.

```
<Window x:Class="AccountManager.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:local="clr-namespace:AccountManager"
   Title="ManageAccounts" SizeToContent="WidthAndHeight"
   ResizeMode="CanMinimize" Loaded="Window_Loaded" >
   <Window.Resources>
      <local:SizeConverter x:Key="sizeConverter"/>
      ...
```

6. Bind the **Width** and **Height** of the inner rectangle in the Button's template passing the value converter as a parameter. In the code below, note that we're passing the value 4 as the converter parameter, which represents the thickness we want as a result of the conversion.

```
<Rectangle Name="innerBorders"
           Height="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Height, Converter={StaticResource
sizeConverter}, ConverterParameter=4}"
           Width="{Binding RelativeSource={RelativeSource
TemplatedParent}, Path=Width, Converter={StaticResource sizeConverter},
ConverterParameter=4}"
           RadiusX="5"
           RadiusY="5">
   ...
```

7. Because of these changes to the template, now we need the developer to always provide values to **Width** and **Height** when creating buttons, as they are essential for the double border effect. To avoid the need to always remember to provide these values, we'll define default values for them in the Button's style definition.

```
<Style TargetType="{x:Type Button}">
   <Setter Property="FontSize" Value="14"/>
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
   <Setter Property="Width" Value="112"/>
   <Setter Property="Height" Value="30"/>
   <Setter Property="Template">
   ...
```

8. Build and run the application to see our updated buttons. Let's modify the template now to respect the Button's **Background**.

```
<Rectangle Name="innerBorders"
   ...
   <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0,0"
                           EndPoint="0,1">
         <GradientStop Offset="0"
                       Color="White"/>
            <GradientStop Offset="1"
                          Color="{Binding
RelativeSource={RelativeSource TemplatedParent},
Path=Background.Color}"/>
      </LinearGradientBrush>
   </Rectangle.Fill>
</Rectangle>
```

9. The last change depends on the developer providing a value for the Button's **Background** property. So let's add a default value for this property to the template.

```
<Style TargetType="{x:Type Button}">
   <Setter Property="FontSize" Value="14"/>
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
   <Setter Property="Width" Value="112"/>
   <Setter Property="Height" Value="30"/>
   <Setter Property="Background" Value="LightBlue"/>
   ...
```

10. Now, let’s add some triggers to the Button’s template to obtain a better response from the user interactions. Add one trigger for increasing the thickness of the outer rectangle border in case the mouse is over the button, and another for shrinking the button when it is pressed.

```
<ControlTemplate TargetType="{x:Type Button}">
   <Viewbox>
      ...
   </Viewbox>
   <ControlTemplate.Triggers>
      <Trigger Property="IsMouseOver"
               Value="True">
         <Setter TargetName="outerBorders"
                 Property="StrokeThickness"
                 Value="2"/>
      </Trigger>
      <Trigger Property="IsPressed"
               Value="True">
         <Setter Property="RenderTransform">
            <Setter.Value>
               <ScaleTransform ScaleX=".9"
                               ScaleY=".9"/>
            </Setter.Value>
         </Setter>
         <Setter Property="RenderTransformOrigin"
                 Value=".5,.5"/>
      </Trigger>
   </ControlTemplate.Triggers>
</ControlTemplate>
```

Part 3. Adding Validation Rules

1. Add a trigger to the TextBox typed style to mark it in LightPink in case there are any validation errors in the control. Additionally, we want the error message to be shown in the TextBox’s **ToolTip** property.

```
<Style TargetType="{x:Type TextBox}">
   <Setter Property="Margin" Value="10"/>
   <Setter Property="FontFamily" Value="Verdana"/>
   <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="true">
         <Setter Property="Background" Value="LightPink"/>
         <Setter Property="ToolTip"
                 Value="{Binding RelativeSource={x:Static
RelativeSource.Self}, Path=(Validation.Errors)[0].ErrorContent}"/>
      </Trigger>
   </Style.Triggers>
</Style>
```

2. To add a validation rule to a binding relationship, we need first the validation class, which inherits **ValidationRule**. Add a new class file to the project named **NameValidationRule.cs**, and provide the code below. You will need to import the namespace **System.Windows.Controls**. This class will return the validation as failed in case the value is blank or larger than 10 characters.

```
public class NameValidationRule : ValidationRule
{
   public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
   {
      string name = value.ToString();

      if (name.Length > 10)
         return new ValidationResult(false, "Name length must be 10
characters or less");
      else if (name.Length == 0)
         return new ValidationResult(false, "Please provide a name");
      else
         return new ValidationResult(true, null);
   }
}
```

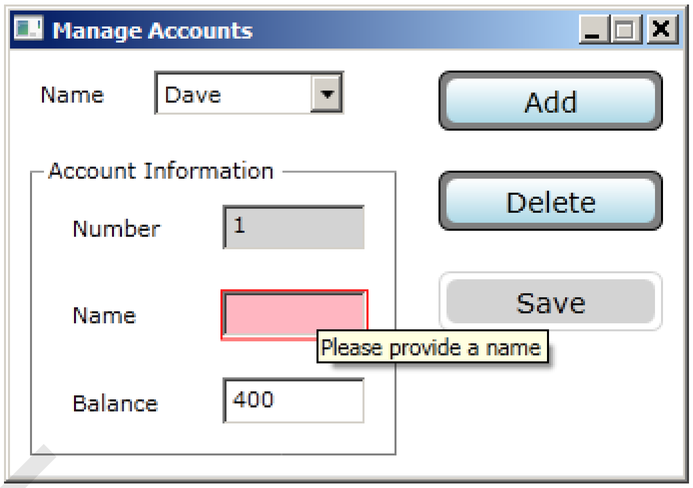
3. Go to the file **MainWindow.xaml.cs** and find the data binding code for the **txtName** TextBox in the **BindAccountInformation()** method. Add the NameValidationRule to the Binding's **ValidationRules** collection.

```
Binding nameBinding = new Binding();
nameBinding.Source = account;
nameBinding.XPath = "Name";
nameBinding.ValidationRules.Add(new NameValidationRule());
txtName.SetBinding(TextBox.TextProperty, nameBinding);
```

4. Build and run the application. Select one of the accounts from the ComboBox and try leaving the **Name** text box empty or with more than 10 characters. You'll notice the trigger working when the TextBox loses focus, which is when it tries to update the data source. We can modify this behavior to let the TextBox update the data source whenever its value changes, and so we'll see the validation working before the TextBox loses focus. To do this, set the binding's **UpdateSourceTrigger** property to PropertyChanged.

```
Binding nameBinding = new Binding();
nameBinding.Source = account;
nameBinding.XPath = "Name";
nameBinding.ValidationRules.Add(new NameValidationRule());
nameBinding.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged;
txtName.SetBinding(TextBox.TextProperty, nameBinding);
```

5. Build and run the application. Notice the validation happening while you type in the **Name** TextBox. When a validation error occurs, you can see the error message in the TextBox's tooltip by placing the mouse over the TextBox for a few seconds.



6. You can add now a validation rule for the **txtBalance** TextBox. Add a new class file named **BalanceValidationRule.cs** to the project, with the following code:

```
public class BalanceValidationRule : ValidationRule
{
   public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
   {
      int balance;

      if (!Int32.TryParse(value.ToString(), out balance))
         return new ValidationResult(false, "Balance must be an integer
value");
      else
         return new ValidationResult(true, null);
   }
}
```

7. Add the validation rule repeating the same procedure done in steps 3 and 4. Build and run the application to check the result.

8. Despite the validation done at the text boxes, the user can still click the Save button when invalid data is entered in one of the text boxes. One approach to fix this issue is disabling the Save button in XAML code, and enabling it in procedural code whenever data is changed. Set the **IsEnabled** property of the **btnSave** button to False.

```
<Button
   Name="btnSave"
   Width="112"
   IsEnabled="False"
   Click="btnSave_Click">
   Save
</Button>
```

9. By using this property, you can notice that the user cannot view that the button is disabled. This can be easily fixed by adding one more trigger to the Button's typed style handling the **IsEnabled** property.

```
<ControlTemplate.Triggers>
   ...
   <Trigger Property="IsEnabled"
            Value="False">
      <Setter TargetName="outerBorders"
              Property="Stroke"
              Value="LightGray">
      </Setter>
      <Setter TargetName="outerBorders"
              Property="Rectangle.Fill"
              Value="White">
      </Setter>
      <Setter TargetName="innerBorders"
              Property="Rectangle.Fill"
              Value="LightGray">
      </Setter>
   </Trigger>
</ControlTemplate.Triggers>
```

10. Add an event handler to the **txtName** TextBox for the **SourceUpdated** event. This event will be raised when the TextBox update the data source through the binding relationship. Use this same handler in the **txtBalance** TextBox.

```
<StackPanel Orientation="Horizontal">
   <Label Width="60">Name</Label>
   <TextBox
      Name="txtName"
      Width="72"
      SourceUpdated="textBox_SourceUpdated"
      >
   </TextBox>
</StackPanel>
<StackPanel Orientation="Horizontal">
   <Label Width="60">Balance</Label>
   <TextBox
      Name="txtBalance"
      Width="72"
      SourceUpdated="textBox_SourceUpdated"
      >
   </TextBox>
</StackPanel>
```

11. The handler code will enable the **btnSave** Button to allow the changes to be saved to the XML file.

```
private void textBox_SourceUpdated(object sender, DataTransferEventArgs
e)
{
   btnSave.IsEnabled = true;
}
```

12. This handler will be called only if we set the binding's **NotifyOnSourceUpdated** property is set to true. Do this in the binding code for the **txtName** and **txtBinding** text boxes.

```
Binding nameBinding = new Binding();
nameBinding.Source = account;
nameBinding.XPath = "Name";
nameBinding.ValidationRules.Add(new NameValidationRule());
nameBinding.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged;
nameBinding.NotifyOnSourceUpdated = true;
txtName.SetBinding(TextBox.TextProperty, nameBinding);

Binding balanceBinding = new Binding();
balanceBinding.Source = account;
balanceBinding.XPath = "Balance";
balanceBinding.ValidationRules.Add(new BalanceValidationRule());
balanceBinding.UpdateSourceTrigger =
   UpdateSourceTrigger.PropertyChanged;
balanceBinding.NotifyOnSourceUpdated = true;
txtBalance.SetBinding(TextBox.TextProperty, balanceBinding);
```

13. Build and run the application. Select an account and try editing name or balance. Notice that when you type in some information in either one of the text boxes the Save button is enabled.

14. Some additional places require enabling the **btnSave** Button too: the methods which add and delete accounts.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
   try
   {
      ...
      btnSave.IsEnabled = true;
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message, "Accounts");
   }
}

private void btnDelete_Click(object sender, RoutedEventArgs e)
{
   try
   {
      int index = cmbAccounts.SelectedIndex;
      if (index != -1)
      {
         ...
         btnSave.IsEnabled = true;
      }
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message, "Accounts");
   }
}
```

15. Finally, modify the code in the **btnSave_Click()** method to check if there are no validation errors before saving, and to disable the Save button back again when the saving is complete.

```
try
{
   if (!txtName.GetBindingExpression(TextBox.TextProperty).HasError &&
      !txtBalance.GetBindingExpression(TextBox.TextProperty).HasError)
   {
      XmlDataProvider dp =
         this.FindResource("dataProvider") as XmlDataProvider;
      dp.Document.Save("AccountsData.xml");
      MessageBox.Show("The data was successfully saved");
      btnSave.IsEnabled = false;
   }
   else
   {
      MessageBox.Show(
         "Please solve all the validation problems before saving");
   }
}
catch (Exception ex)
{
   MessageBox.Show(ex.Message, "Accounts");
}
```

16. Build and run the application. Now your program should be fully functional, and you can test the Add, Delete and Save buttons thoroughly.