

# **Windows Presentation Foundation Using C#**

*Student Guide*

**Revision 4.0**

# **Windows Presentation Foundation Using C#**

## **Rev. 4.0**

### **Student Guide**

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.



® is a registered trademark of Object Innovations.

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

**Special Thanks:** Dana Wyatt

Copyright ©2010 Object Innovations Enterprises, LLC All rights reserved.

Object Innovations  
877-558-7246  
[www.objectinnovations.com](http://www.objectinnovations.com)

Published in the United States of America.

## Table of Contents (Overview)

Chapter 1	Introduction to WPF
Chapter 2	XAML
Chapter 3	WPF Controls
Chapter 4	Layout
Chapter 5	Dialogs
Chapter 6	Menus and Commands
Chapter 7	Toolbars and Status Bars
Chapter 8	Dependency Properties and Routed Events
Chapter 9	Resources
Chapter 10	Data Binding
Chapter 11	Styles, Templates, Skins and Themes
Appendix A	Learning Resources

# Directory Structure

---

- **Install the course software by running the self-extractor *Install\_WpfCs\_40.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)

<b>Chapter 1: Introduction to WPF .....</b>	<b>1</b>
History of Microsoft GUI .....	3
Why WPF?.....	4
When Should I Use WPF? .....	5
WPF and .NET Framework 3.0 .....	6
.NET Framework 4.0 .....	7
Visual Studio 2010.....	8
WPF Core Types and Infrastructures.....	9
XAML.....	10
Controls.....	11
Data Binding.....	12
Appearance .....	13
Layout and Panels .....	14
Graphics .....	15
Media .....	16
Documents and Printing.....	17
Plan of Course.....	18
Application and Window .....	19
FirstWpf Example Program .....	20
Demo – Using Visual Studio 2010 .....	21
Creating a Button .....	22
Providing an Event Handler.....	23
Specifying Initial Input Focus.....	24
Complete First Program.....	25
Device-Independent Pixels .....	27
Class Hierarchy .....	28
Content Property .....	29
Simple Brushes .....	30
Panels .....	31
Children of Panels.....	32
Example – TwoControls .....	33
TwoControls – Code .....	34
Automatic Sizing .....	35
Lab 1 .....	37
Summary .....	38
<b>Chapter 2: XAML.....</b>	<b>43</b>
What Is XAML? .....	45
Default Namespace .....	46
XAML Language Namespace.....	47
.NET Class and Namespace.....	48
Elements and Attributes.....	49

XAML in Visual Studio 2010.....	50
Demo: One Button via XAML .....	51
Adding an Event Handler .....	54
Layout in WPF.....	56
Controlling Size .....	57
Margin and Padding.....	58
Thickness Structure.....	59
Children of Panels.....	60
Example – TwoControlsXaml .....	61
TwoControls – XAML.....	62
Automatic Sizing .....	63
TwoControls – Code .....	64
Orientation .....	65
Access Keys.....	66
Access Keys in XAML .....	67
Content Property .....	68
Checked and Unchecked Events .....	69
Lab 2 .....	70
Property Element Syntax .....	71
Type Converters.....	72
Summary .....	73
<b>Chapter 3: WPF Controls .....</b>	<b>79</b>
Buttons in WPF.....	81
ButtonDemo Example.....	82
Using the Button Class .....	83
Toggle Buttons.....	84
IsThreeState .....	85
CheckBox.....	86
CheckBox Code .....	87
ToolTip .....	88
RadioButton .....	89
GroupBox.....	90
Images .....	91
Lab 3A .....	92
TextBox.....	93
Initializing the TextBox .....	94
Clipboard Support.....	95
Items Controls .....	96
Selector Controls.....	97
Using a ListBox .....	98
ShowListSingle Example.....	99
Multiple-Selection ListBox.....	100
Selected Items .....	101
Using the ComboBox.....	102
ComboBox Example .....	103

Storing Objects in List Controls .....	105
Collection Items in XAML .....	106
Lab 3B .....	107
Summary .....	108
<b>Chapter 4: Layout.....</b>	<b>117</b>
Layout in WPF .....	119
Controlling Size: Review .....	120
Margin and Padding: Review .....	121
Thickness Structure: Review .....	122
SizeDemo Program .....	123
Top Panel .....	124
Content Property .....	125
XAML vs. Code.....	126
Type Converter .....	128
Alignment .....	129
Default Alignment Example .....	130
Alignment inside a Stack Panel .....	131
Vertical Alignment .....	132
Horizontal Alignment .....	133
Vertical Alignment in a Window .....	134
Content Alignment.....	135
Content Alignment Example.....	136
FlowDirection .....	137
Transforms .....	138
RotateTransform Example .....	139
Panels .....	140
Shapes .....	141
Size and Position.....	142
Simple Shapes Example.....	143
Attached Properties .....	144
StackPanel.....	145
Children of StackPanel .....	146
WrapPanel.....	147
DockPanel .....	148
Dock Example XAML and Code.....	149
Lab 4A .....	150
Grid .....	151
Grid Example .....	152
Grid Demo .....	153
Using the Collections Editor.....	154
Star Sizing.....	158
Grid.ColumnSpan .....	159
Scrolling.....	160
Scaling .....	161
ScrollViewer and Viewbox Compared .....	162

Lab 4B.....	163
Summary .....	164
<b>Chapter 5: Dialogs .....</b>	<b>181</b>
Dialog Boxes in WPF .....	183
MessageBox.....	184
MessageBox Show Method .....	185
Closing a Form: Review .....	188
Common Dialog Boxes.....	189
FileOpen Example .....	190
FileOpen Example Code.....	191
Custom Dialogs.....	192
Modal Dialogs.....	193
Modal Dialog Example.....	194
New Product Dialog.....	195
XAML for New Product Dialog .....	196
Code for New Product Dialog.....	197
Bringing up the Dialog .....	198
Dialog Box Owner .....	199
Modeless Dialog Box Example .....	200
Displaying the Dialog .....	201
Communicating with Parent .....	202
XAML for Modeless Dialog.....	203
Handler for the Apply Button .....	204
Handler for the Close Button .....	205
Instances of a Modeless Dialog .....	206
Checking for an Instance .....	207
Lab 5 .....	208
Summary .....	209
<b>Chapter 6: Menus and Commands .....</b>	<b>217</b>
Menus in WPF .....	219
Menu Controls .....	220
MenuCalculator Example .....	221
A Simple Menu .....	222
The Menu Using XAML.....	223
Handling the Click Event.....	224
The Menu Using Procedural Code.....	225
Icons in Menus.....	226
Context Menu .....	227
XAML for Context Menu .....	228
Separator .....	229
Lab 6A .....	230
Keyboard Shortcuts.....	231
Commands .....	232
Simple Command Demo.....	233



WPF Command Architecture.....	236
Command Bindings .....	237
Command Binding Demo .....	238
Custom Commands .....	241
Custom Command Example .....	242
MenuCalculator Command Bindings .....	244
Input Bindings.....	245
Menu Items .....	246
Running MenuCalculator.....	247
Checking Menu Items .....	248
Common Event Handlers.....	249
Menu Checking Logic .....	250
Calculation Logic.....	251
Automatic Checking .....	252
Automatic Checking Example .....	253
Lab 6B.....	254
Summary .....	255
<b>Chapter 7: Toolbars and Status Bars .....</b>	<b>263</b>
Toolbars in WPF.....	265
XAML for Toolbars.....	266
Commands and Events.....	267
Images on Buttons.....	268
Tool Tips.....	269
Other Elements on Toolbars .....	270
Status Bars .....	271
Lab 7 .....	272
Summary .....	273
<b>Chapter 8: Dependency Properties and Routed Events.....</b>	<b>281</b>
Dependency Properties .....	283
Change Notification.....	284
Property Trigger Example .....	285
Property Value Inheritance .....	286
Property Value Inheritance Example.....	287
Support for Multiple Providers .....	288
Logical Trees .....	289
Visual Tree.....	290
Visual Tree Example.....	291
Routed Events .....	292
Event Handlers.....	293
Routing Strategies.....	294
Ready-made Routed Events in WPF.....	295
Routed Event Example .....	296
Lab 8 .....	299
Summary .....	300

<b>Chapter 9: Resources.....</b>	<b>305</b>
Resources in .NET .....	307
Resources in WPF .....	308
Binary Resources .....	309
Loose Files as Resources .....	310
Binary Resources Example .....	311
Logical Resources .....	312
Logical Resources Demo .....	314
Logical Resources in Code .....	317
Static Resources .....	319
Dynamic Resources .....	320
DynamicResource Example.....	321
Lab 9 .....	322
Summary .....	323
<b>Chapter 10: Data Binding .....</b>	<b>329</b>
What is Data Binding? .....	331
Binding in Procedural Code.....	332
Procedural Code Example.....	333
Binding in XAML.....	335
Binding to Plain .NET Properties .....	336
Binding to .NET Properties Example .....	337
Binding to a Collection .....	339
Binding to a Collection Example.....	340
Lab 10A .....	341
Controlling the Selected Item .....	342
ComboBox Synchronization Example.....	343
Data Context .....	344
Data Context Demo.....	345
Data Templates .....	348
Data Template Example.....	349
Specifying a Data Template.....	350
Value Converters .....	351
Value Converter Example.....	352
Using a Value Converter in XAML.....	353
Collection Views.....	355
Sorting.....	356
Grouping .....	357
Grouping Example .....	358
Filtering.....	360
Filtering Example .....	361
Collection Views in XAML.....	362
Collection Views in XAML Example.....	363
Data Providers.....	364
ObjectDataProvider .....	365
ObjectDataProvider Example .....	366

XmlDataProvider .....	368
XmlDataProvider Example .....	369
Lab 10B .....	370
Data Access with Visual Studio 2010 .....	371
SmallPub Database .....	372
ADO.NET Entity Framework .....	374
Book Browser Demo .....	375
Book Browser Demo Completed .....	379
Navigation Code .....	380
DataGrid Control .....	381
Editing the Book Table .....	384
Class Library .....	385
Database Updates .....	386
Refreshing the DataGrid .....	387
Summary .....	388
<b>Chapter 11: Styles, Templates, Skins and Themes .....</b>	<b>401</b>
WPF and Interfaces .....	403
Styles .....	404
Style Example .....	405
Style Definition .....	406
Applying Styles .....	407
Style Inheritance .....	408
Style Overriding .....	409
Sharing Styles .....	410
Style Sharing Example .....	411
Demo: Restricting Styles .....	413
Typed Styles .....	416
Typed Style Example .....	417
Triggers .....	419
Property Trigger Example .....	420
Data Trigger Example .....	422
Multiple Conditions .....	424
Validation .....	425
Validation Example .....	426
Templates .....	427
A Simple Template Example .....	428
Improving the Template .....	429
Templated Parent's Properties .....	430
Respecting Properties Example .....	431
Respecting Visual States .....	434
Respecting Visual States Example .....	435
Using Templates with Styles .....	436
Templates with Styles Example .....	437
Skins .....	438
Changing Skins .....	439

Skins Example .....	440
Themes .....	442
Themes Example.....	443
Lab 11 .....	445
Summary .....	446
<b>Appendix A: Learning Resources.....</b>	<b>459</b>

# **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 4.0.**
- **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 2010.**
- **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?

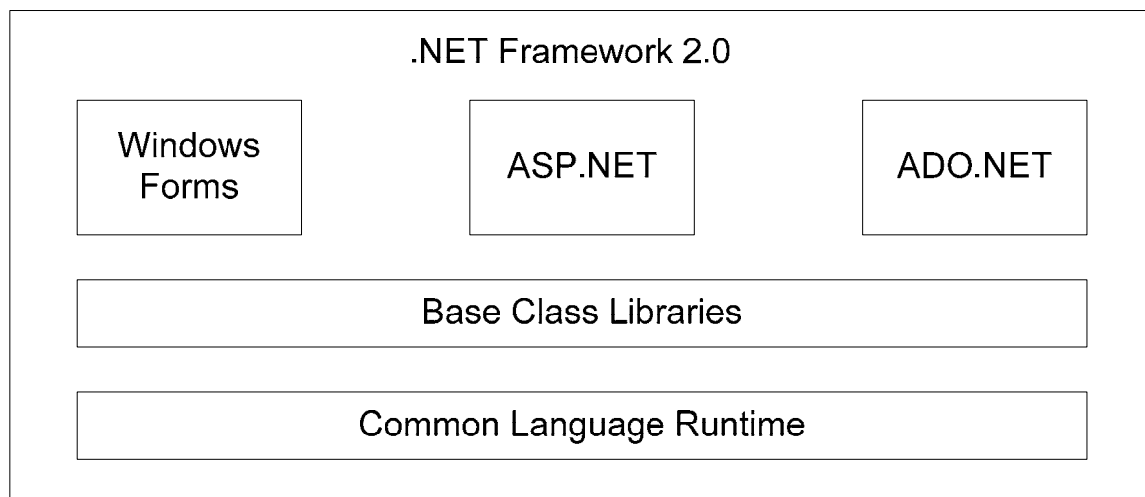
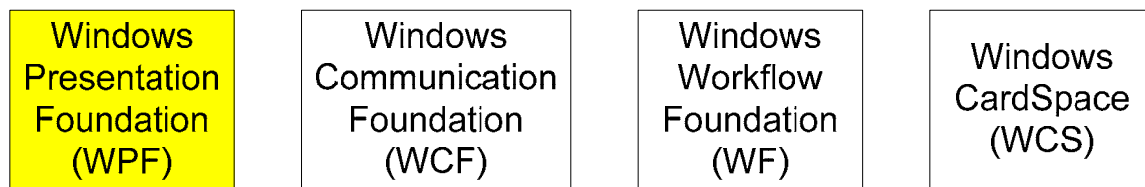
---

- **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 2010 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, a default for Windows Vista and Windows 7 but not for other Windows operating systems.
  - 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**

---

- **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 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 2 for Web development.
  - A new programming model for parallel programming.
  - And much more!

# Visual Studio 2010

---

- **Visual Studio 2010 provides effective tooling support for .NET Framework 4.0.**
  - Early support for WinFX involved add-ons to Visual Studio, but now there is a fully integrated environment.
- **Visual Studio 2010 has a new IDE with an attractive new graphical appearance.**
  - VS 2010 is implemented using WPF!
- **Features in Visual Studio 2010 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 or .NET 4.0.
- **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.**

# 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.
- **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 2010 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.

# 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;

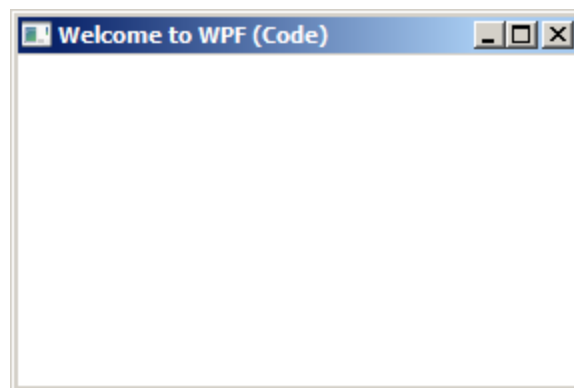
namespace FirstWpf
{
    public class MainWindow : Window
    {
        [STAThread]
        static void Main(string[] args)
        {
            Application app = new Application();
            app.Run(new MainWindow());
        }
        public MainWindow()
        {
            Title = "Welcome to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

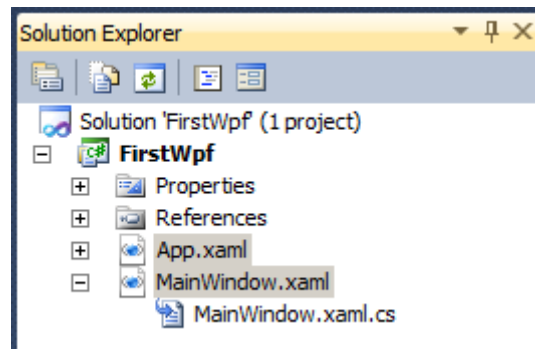




## Demo – Using Visual Studio 2010

---

- Although you can compile WPF programs at the command-line, for simplicity we will use Visual Studio 2010 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 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 2010!

## 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 see the button fills the whole client area of the main window.
9. Add the following code to specify the horizontal and vertical alignment of the button.

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

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

## Providing an Event Handler

---

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

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

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



# Specifying Initial Input Focus

---

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

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

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

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

# Complete First Program

---

- See *FirstWpf\Step2*.

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

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

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

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

            Content = btn;
        }
    }
}
```

## Complete First Program (Cont'd)

---

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

# Device-Independent Pixels

---

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

# Class Hierarchy

---

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

Object

DispatcherObject (abstract)

Application

DependencyObject

Visual (abstract)

UIElement

FrameworkElement

Control

ContentControl

Window

ButtonBase

Button



# Content Property

---

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

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

```
Content = btn;
```

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

# Simple Brushes

---

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

```
public SimpleBrush()  
{  
    Title = "Simple Brushes";  
    Width = 288;  
    Height = 192;  
    Background = new SolidColorBrush(Colors.Beige);  
  
    Button btn = new Button();  
    ...  
    btn.Background = new SolidColorBrush(  
        Color.FromRgb(0, 255, 0));  
    btn.Foreground = new SolidColorBrush(  
        Color.FromRgb(0, 0, 255));  
    Content = btn;  
}
```

# Panels

---

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

UIElement

    FrameworkElement

        Panel (abstract)

            Canvas

            DockPanel

            Grid

            StackPanel

            UniformGrid

            WrapPanel

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

# Children of Panels

---

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

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

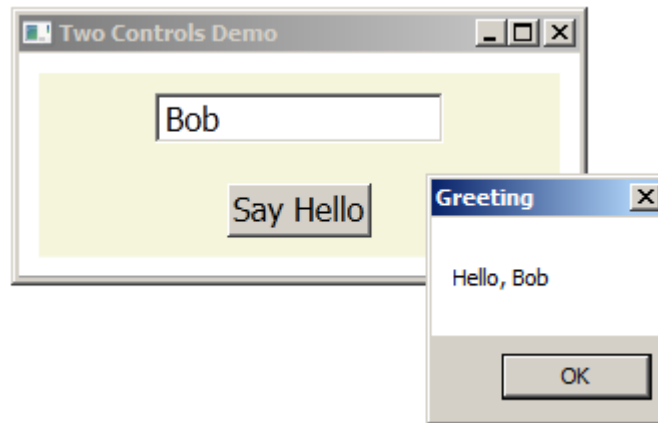
```
Button btnGreet = new Button();  
...
```

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

## Example – TwoControls

---

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



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

## TwoControls – Code

---

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

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

    private TextBox txtName;

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

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

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

# Automatic Sizing

---

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

```
public TwoControls()  
{  
    Title = "Two Controls Demo";  
    Width = 288;  
    const int MARGINSIZE = 10;  
  
    StackPanel panel = new StackPanel();  
    Content = panel;  
  
    SizeToContent = SizeToContent.Height;  
  
    panel.Background = Brushes.Beige;  
    panel.Margin = new Thickness(MARGINSIZE);
```

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

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

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

## TwoControls – Code (Cont'd)

---

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

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

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

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

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

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

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



# Lab 1

---

## A Windows Application with Two Controls

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

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

Suggested time: 30 minutes

# Summary

---

- **The goal of Windows Presentation Framework is to provide a unified framework for creating modern user experiences.**
- **WPF is a major component of the .NET Framework.**
  - In .NET 3.0/3.5, it is layered on top of .NET Framework 2.0.
  - In .NET 4.0 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 2010, 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

<b>Directories:</b>	<b>Labs\Lab1</b>	(do your work here)
	<b>Chap01\TwoControls\Step1</b>	(answer to Part 1)
	<b>Chap01\TwoControls\Step2</b>	(answer to Part 2)

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

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

1. Use Visual Studio to create a new WPF application **TwoControls** in the Lab1 folder.
2. In Solution Explorer, delete the files **App.xaml** and **MainWindow.xaml**.
3. Add a new code file **Program.cs** to your project.
4. In **Program.cs** enter the following code, which does the minimum of creating Application and Window objects.

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

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

```

        public TwoControls()
        {
        }
    }
}

```

5. Build and run. You should get a clean compile. You should see a main window, which has no title and an empty client area.
6. Add the following code to the **TwoControls** constructor.

```

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

```

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

```

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

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

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

```

9. Build. You'll get a compiler error because you need a new namespace for the **Brushes** class.
10. Bring in the **System.Windows.Media** namespace. Now you should get a clean build. Run your application. You should see the StackPanel displayed as solid beige, with a small margin.
11. Next we will add a TextBox as a child of the panel. Since we will be referencing the TextBox in an event-handler method as well as the constructor, define a private data member **txtName** of type **TextBox**.

```

private TextBox txtName;

```

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

```

txtName = new TextBox();

```

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

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

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

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

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

## Part 2. Event Handling and Layout

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

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

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

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

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

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

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

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

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

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

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



# **Chapter 2**

## **XAML**

# XAML

## Objectives

---

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

- **Describe Extensible Application Markup Language (XAML) and its role in WPF and .NET Framework 4.0.**
- **Explain the structure of XAML documents.**
- **Describe the XML namespaces you must use in your XAML documents.**
- **Use Visual Studio 2010 to create and edit XAML documents.**
- **Provide access key support in XAML.**
- **Explain the use of the content property in XAML.**
- **Use property elements to set values for complex properties in XAML documents.**
- **Explain the use of type converters in XAML.**



# What Is XAML?

---

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

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

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

## Default Namespace

---

- To ensure that the `<Button>` element gets mapped to the .NET `Button` class, we must ensure that the XML is in a suitable *namespace*.
    - The namespace should be applied to the root element of the XML document.
- ```
<Window x:Class="OneButton.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  ...

  <Button HorizontalAlignment="Center"
    VerticalAlignment="Center"
    FontSize="16"
    Click="Button_Click">
    Say Hello
  </Button>
```
- The attribute **xmlns** declares a default XML namespace, which applies to the element in which the namespace declaration appears as well as all child elements.
  - The URL "**http://schemas.microsoft.com/winfx/2006/xaml/presentation**" does not correspond to anything on the Web. Rather, WPF itself maps this XML namespace to common .NET namespaces, including **System.Windows**, **System.Windows.Controls**, and so on.
  - WPF will then recognize **Button** as a class and **FontSize**, **HorizontalAlignment** and so forth as properties of this class.

# XAML Language Namespace

---

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

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

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

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

## .NET Class and Namespace

---

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

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

# Elements and Attributes

---

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

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

# XAML in Visual Studio 2010

---

- **Visual Studio 2010 will create projects for you that use XAML for creating WPF objects used in your program.**

- The **Application** object is created in **App.xaml**.

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

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

- The **Window** object is created in **MainWindow.xaml**.

```
<Window x:Class="WpfApplication1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/
    2006/xaml"
  Title="MainWindow" Height="350" Width="525">
  <Grid>

    </Grid>
</Window>
```

## Demo: One Button via XAML

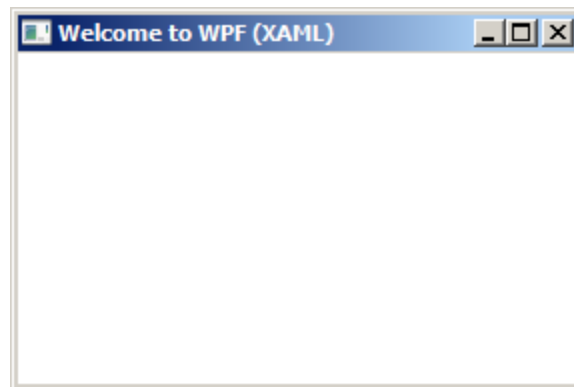
---

1. Use Visual Studio 2010 to create a new WPF Application called **OneButton** in the **Demos** folder.
2. Edit the file **MainWindow.xaml** to specify a title and the same height and width as in our procedural code version.

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

    </Grid>
</Window>
```

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



## One Button Demo (Cont'd)

---

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

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

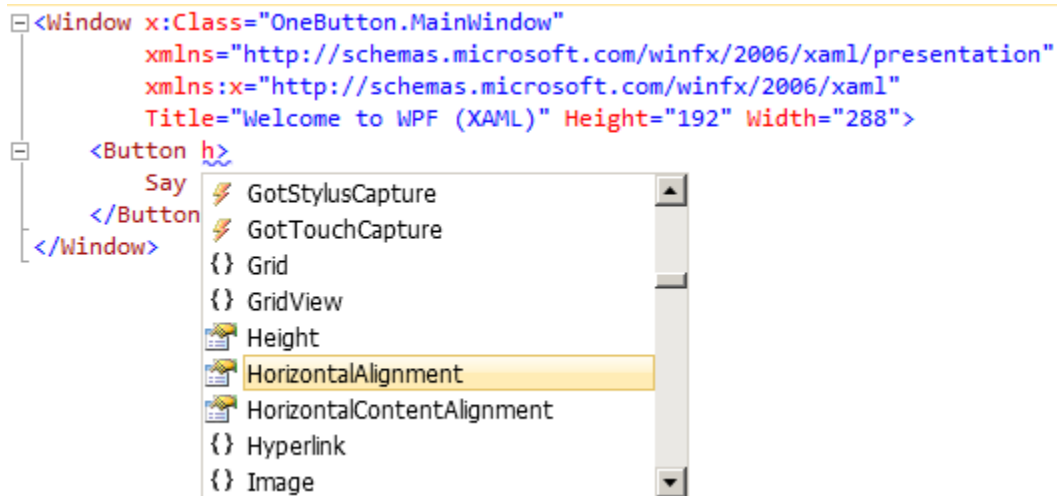
5. Build and run. The button fills the whole window!





## One Button Demo (Cont'd)

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



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

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


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



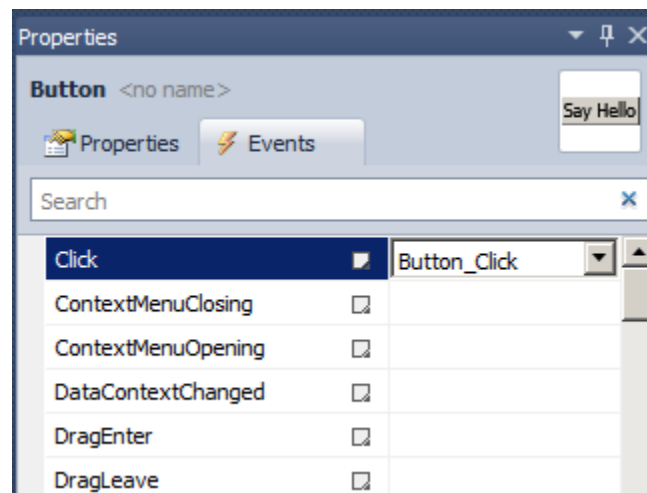
# Adding an Event Handler

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

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

A yellow tooltip box with a small icon on the left and the text "<New Event Handler>" on the right, appearing over the empty Click attribute in the XAML code.

- Use the Events tab of the Properties window. Double-click the event you wish handled.



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

9. Here is the final XAML for the button:

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

## One Button Demo (Cont'd)

---

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

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

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



# Layout in WPF

---

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

# Controlling Size

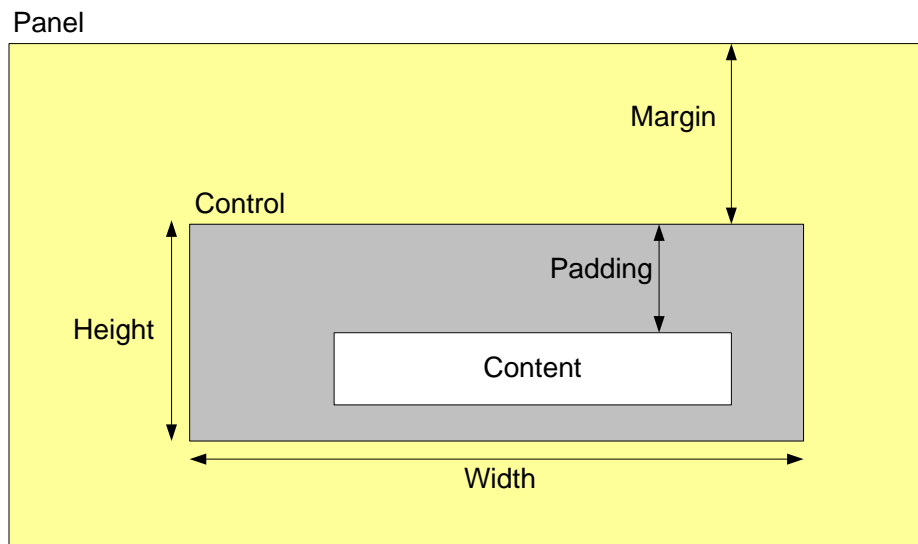
---

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

# Margin and Padding

---

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



# Thickness Structure

---

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

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

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

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

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

# Children of Panels

---

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

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

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

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

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

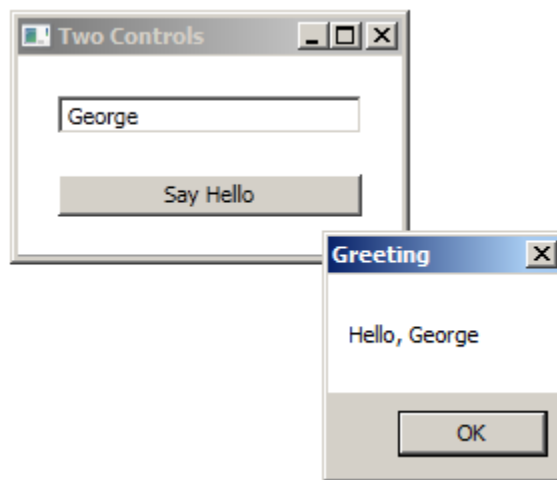
```
<StackPanel>  
    <TextBox Margin="20"  
        Name="txtName">  
    </TextBox>  
    <Button Margin="20, 0, 20, 20"  
        Name="btnSayHello"  
        Click="btnSayHello_Click">  
        Say Hello  
    </Button>  
</StackPanel>
```



## Example – TwoControlsXaml

---

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



- The program also illustrates various automatic sizing features of WPF and use of one of the list controls.
- This version of the program uses the *StackPanel*'s default vertical orientation of child controls.

## TwoControls – XAML

---

- **Here is the complete XAML:**

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

# Automatic Sizing

---

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

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

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

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

## TwoControls – Code

---

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

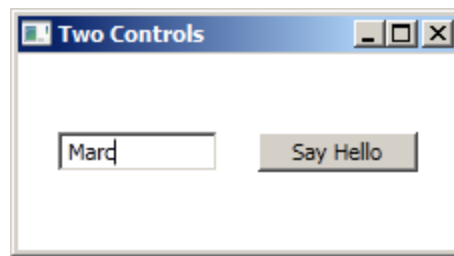
```
namespace TwoControlsXaml
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow: Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

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

# Orientation

---

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



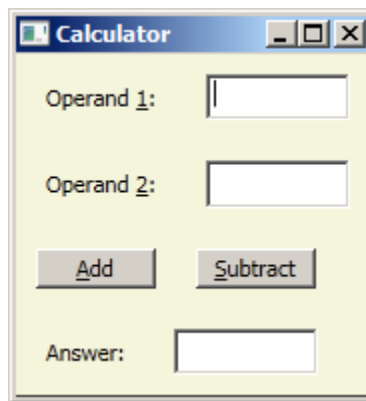
- In the **XAML** we also change a few other properties to illustrate the layout of controls.

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

# Access Keys

---

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



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

# Access Keys in XAML

---

- **No code is required to make access keys work—you can do everything in XAML.**

- Place an underscore in front of the character you want to serve as the access key. For a button, that is all you have to do.

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

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

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

# Content Property

---

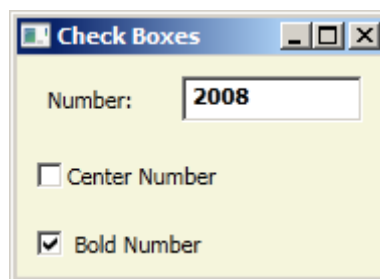
- **The content of a control in XAML may be placed between the start and end tags for the control.**

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

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

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

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



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



# Checked and Unchecked Events

---

- **Checking and unchecking a check box fires the events *Checked* and *Unchecked*.**

- Our example provides handlers for these events.

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

- Both events have a common handler.

- **We distinguish the two states by the *IsChecked* property, which is of type *Nullable<Boolean>*.**

- Cast it to **bool** to use it as a condition in an **if** statement.

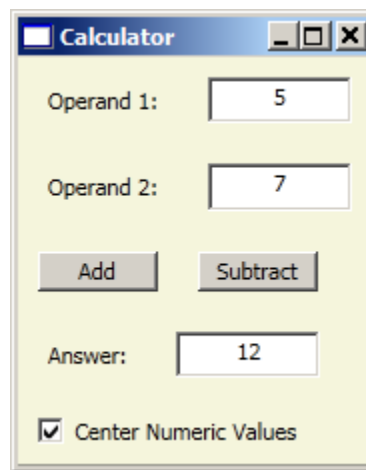
```
private void chkCenter_Checked(object sender,
RoutedEventArgs e)
{
    if ((bool)chkCenter.IsChecked)
    {
        txtNumber.TextAlignment =
            TextAlignment.Center;
    }
    else
    {
        txtNumber.TextAlignment = TextAlignment.Left;
    }
}
```

# Lab 2

---

## Calculator Program via XAML

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



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

Suggested time: 45 minutes

# Property Element Syntax

---

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

- **You could then use this notation:**

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

- The syntax is <TypeName.Property>.

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

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

# Type Converters

---

- **There is a subtlety in the way we commonly write attributes in XAML.**
  - Consider the **Background** attribute of **Button** and the manner we used it on the previous page.
- **XAML provides a number of *type converters* that will perform useful type conversions automatically.**
- **Without this feature, you would have to specify the *Background* property using more cumbersome property element syntax.**

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

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

# Summary

---

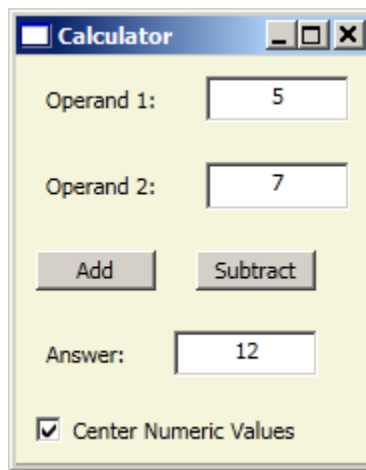
- **Extensible Application Markup Language (XAML) provides a declarative XML-based syntax for defining user interfaces in WPF.**
- **There are two XML namespaces that you must use in your XAML documents.**
- **Visual Studio 2010 makes it easy to create and edit XAML documents.**
- **Access key support can be provided in XAML without use of procedural code.**
- **The content property can be used as shorthand in lieu of the Content attribute.**
- **You can use property elements to set values for complex properties in XAML documents.**
- **Type converters can convert strings to other WPF types.**

## Lab 2

### Calculator Program via XAML

#### Introduction

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



**Suggested Time:** 45 minutes

**Root Directory:** OIC\WpfCs

<b>Directories:</b>	<b>Labs\Lab2</b>	(do your work here)
	<b>Chap02\Calculator\Step1</b>	(answer to Part 1)
	<b>Chap02\Calculator\Step2</b>	(answer to Part 2)
	<b>Chap02\Calculator\Step3</b>	(answer to Part 3)

#### Part 1. Basic User Interface

1. Use Visual Studio to create a new WPF application **Calculator** in the **Lab2** folder.
2. Edit the starter XAML to make the title “Calculator”, and replace the grid by a stack panel with beige background.

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

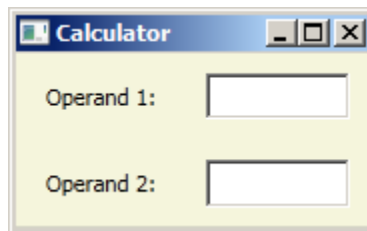
3. Provide XAML for the top pair of label and text box, for the first operand. They should be inside a nested stack panel with horizontal orientation.

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

4. Provide the XAML for the second pair of label and text box. Build and run. You should see your controls, but also a lot of extra space in the window.
5. Replace the hardcoded width and height by specifying **SizeToContent** as "WidthAndHeight"..

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

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

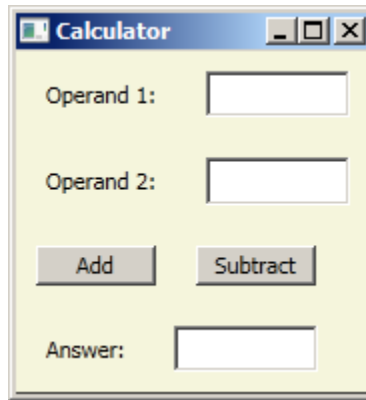


7. Add another horizontal stack panel with a pair of buttons for Add and Subtract.

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

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

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



## Part 2. Basic Functionality

- Add a handler for the Click event of the Add button. The simplest way to do this is to double-click on the Add button in Design view.
- In the event handler provide code to convert the strings entered for the operands to numbers, add these numbers, and store the answer.

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

- In a similar manner provide a handler for the Subtract button.
- Build and run. Do the Alt + 1 and Alt + 2 access keys work to position focus at the first and second text boxes, respectively? You are at Step 2.

## Part 3. Enhancements

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

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

- Provide a similar target for the second label.



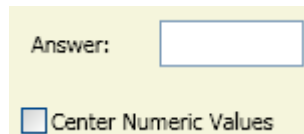
3. Add access key support for the buttons simply by providing an underscore in front of the desired letter.

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

4. Build and run. The access keys should now work.
5. Add XAML for a check box indicating whether or not to center the numeric values.

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

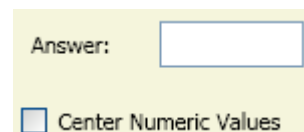
6. Build and run. The caption for the check box is jammed up against the box.



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

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

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



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

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

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

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

# **Chapter 7**

## **Toolbars and Status Bars**

# Toolbars and Status Bars

## Objectives

---

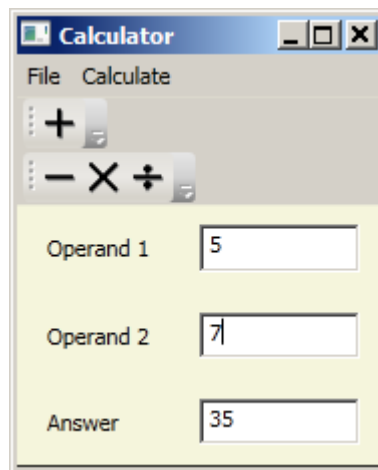
*After completing this unit you will be able to:*

- **Implement toolbars in your programs.**
- **Use tool tips in your programs.**
- **Provide status bars in your programs.**

# Toolbars in WPF

---

- **WPF makes it easy to create toolbars as a quick way for the user to access program functionality.**
- **You can create multiple toolbars that can be dragged around by the user.**
- **As an example, consider a new version of our calculator program.**
  - See **ToolbarCalculator\Step1** in the chapter directory.



- There are two toolbars, one for just Add and the other for the remaining arithmetic operations.
- The screenshot shows the second toolbar dragged below the first.

# XAML for Toolbars

---

```
<ToolBarTray Name="tbTray"
    DockPanel.Dock="Top">
    <ToolBar Band="0" BandIndex="0">
        <Button Command=
            "{x:Static custom:MainWindow.Add}"
            ToolTip="Add">
            <Image Source="c:\OIC\Data\Graphics\MISC18.ICO"
        />
    </Button>
    </ToolBar>
    <ToolBar Band="0" BandIndex="1">
        <Button Command=
            "{x:Static custom:MainWindow.Subtract}"
            ToolTip="Subtract">
            <Image Source="c:\OIC\Data\Graphics\MISC19.ICO"
        />
    </Button>
        <Button Command=
            "{x:Static custom:MainWindow.Multiply}"
            ToolTip="Multiply">
            <Image Source="c:\OIC\Data\Graphics\MISC20.ICO"
        />
    </Button>
        <Button Click="DivideClick"
            ToolTip="Divide">
            <Image Source="c:\OIC\Data\Graphics\MISC21.ICO"
        />
    </Button>
    </ToolBar>
</ToolBarTray>
```

- There are two toolbars in the toolbar tray.

# Commands and Events

---

- **You can specify events and handlers for buttons on toolbars in the same manner as you specify handlers for menu items.**
  - In fact, you can use the exact same commands used on menu items.
  - This provides an alternative way to invoke the command, without having to write any additional procedural code.
- **Using commands also allows you to tap into the WPF facility for disabling UI items.**
  - In our example, if either of the operand fields is blank, the Add, Subtract and Multiply buttons are disabled.
  - Try clicking on them—nothing happens.
  - But the buttons are not shown grayed out.
- **If you use a Click handler, you don't get automatic disabling.**
  - Try the Divide button. If either operand is blank, you will hit an exception.

## Images on Buttons

---

- **The XAML syntax for buttons on toolbars is the same as for any other kind of button.**
- **Thus we can specify images the same way we did before.**

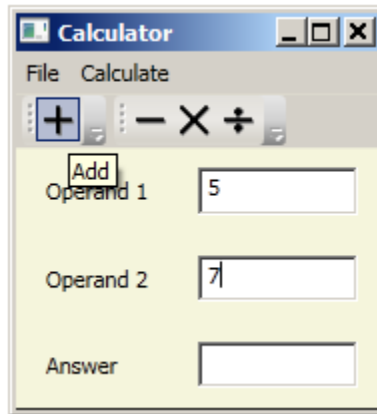
```
<Button Command=
    "{x:Static custom:MainWindow.Add}"
    ToolTip="Add">
    <Image Source="c:\OIC\Data\Graphics\MISC18.ICO" />
</Button>
```



# Tool Tips

---

- A tool tip is a little yellow box that supplies information when the mouse hovers over a UI element.



- In WPF you can specify a tool tip by using the *ToolTip* class.

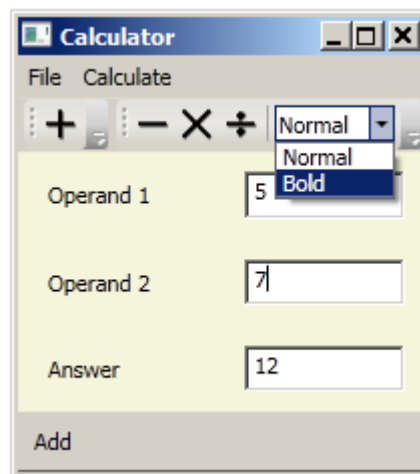
```
<Button Command=
    "{x:Static custom:MainWindow.Add}"
    ToolTip="Add">
    <Image Source="c:\OIC\Data\Graphics\MISC18.ICO" />
</Button>
```

- The tool tip is shown if the button is enabled.

## Other Elements on Toolbars

---

- **A toolbar may contain any control, not just a button.**
  - Some common elements for toolbars besides buttons are text boxes and combo boxes.
- **As an example, Step 2 of *ToolbarCalculator* provides a combo box for specifying the font weight for the answer text box.**
  - The allowed options are Normal and Bold.



- Here is the XAML:

```
<ComboBox Name="cmbBold"
           Width="60"
           SelectionChanged=
               "cmbBold_SelectionChanged">
</ComboBox>
```

- **This example also illustrates a *StatusBar*.**

# Status Bars

---

- **Status bars provide a consistent user interface for providing information at the bottom of a window.**
- **WPF provides the *StatusBar* control for this purpose.**
- **You will normally dock the *StatusBar* to the bottom of the window.**
- **The *StatusBar* can contain any kind of control, but most commonly labels are used.**

```
<StatusBar DockPanel.Dock="Bottom">
    <Label Name="lblOp" Width="60">
        Ready
    </Label>
</StatusBar>
```

- **You can write information to a child of the *StatusBar* using the name of the child.**

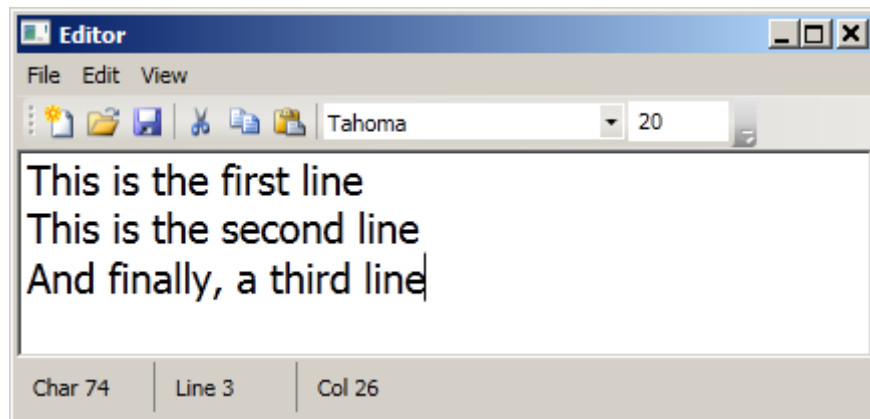
```
private void AddExecuted(object sender,
ExecutedRoutedEventArgs args)
{
    int num1 = Convert.ToInt32(txtOp1.Text);
    int num2 = Convert.ToInt32(txtOp2.Text);
    int answer = num1 + num2;
    txtAns.Text = answer.ToString();
    lblOp.Content = "Add";
}
```

# Lab 7

---

## Toolbar and Status Bar for the Simple Editor

In this lab you will enhance your simple editor by providing a toolbar and a status bar. You will provide an icon on a number of menu items, corresponding to the toolbar button images. You will also implement a checkable menu item to show or hide the status bar.



Detailed instructions are contained in the Lab 7 write-up at the end of the chapter.

Suggested time: 45 minutes

# Summary

---

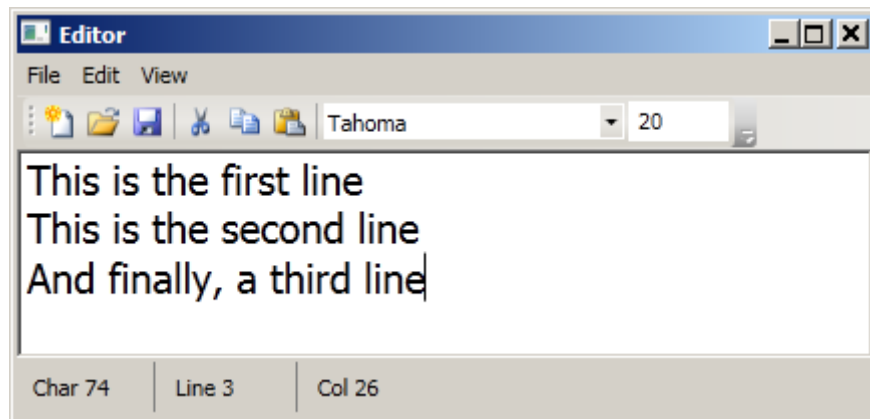
- **WPF makes it easy to create toolbars as a quick way for the user to access program functionality.**
- **A tool tip is a little yellow box that supplies information when the mouse hovers over a UI element.**
- **Status bars provide a consistent user interface for providing information at the bottom of a window.**

## Lab 7

### Toolbar and Status Bar for the Simple Editor

#### Introduction

In this lab you will enhance your simple editor by providing a toolbar and a status bar. You will provide an icon on a number of menu items, corresponding to the toolbar button images. You will also implement a checkable menu item to show or hide the status bar.



**Suggested Time:** 45 minutes

**Root Directory:** OIC\WpfCs

<b>Directories:</b>	<b>Labs\Lab7\Editor</b>	(do your work here)
	<b>Chap06\Editor\Step5</b>	(backup of starter code)
	<b>Chap07\Editor\Step6</b>	(answer)

#### Part 1. Implement a Toolbar

1. If you would like to continue with your own version of the Editor, delete the supplied starter files and replace them with your files from the Lab6B folder. Build and run your starter code.
2. Add the following to the XAML file, after the definition of the menu. Note that the various image files are in the directory **C:\OIC\Data\Graphics**.

```
<ToolBarTray Name="tbTray"
    DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="New">
            <Image Source="c:\OIC\Data\Graphics\new.png" />
        </Button>
```

```

<Button Command="Open">
    <Image Source="c:\OIC\Data\Graphics\open.png" />
</Button>
<Button Command="Save">
    <Image Source="c:\OIC\Data\Graphics\save.png" />
</Button>
<Separator/>
<Button Command="Cut">
    <Image Source="c:\OIC\Data\Graphics\cut.png" />
</Button>
<Button Command="Copy">
    <Image Source="c:\OIC\Data\Graphics\copy.png" />
</Button>
<Button Command="Paste">
    <Image Source="c:\OIC\Data\Graphics\paste.png" />
</Button>
</ToolBar>
</ToolBarTray>

```

3. Build and run. You'll hit an exception. If you run under the debugger, you'll see that the exception is thrown on the first statement of **EditorCanExecute()**. At this point, **txtData** has not yet been initialized. Why did a similar problem not occur with menus?
4. The toolbar is always visible, but the menus have to be pulled down by the user. Thus **EditorCanExecute()** won't be called before there is some interaction from the user.
5. Now let's fix the problem. Include in **EditorCanExecute()** a test for **txtData** being **null**.

```

private void EditorCanExecute(object sender, CanExecuteRoutedEventArgs
e)
{
    if (txtData == null)
        e.CanExecute = false;
    else if (txtData.Text != "")
        e.CanExecute = true;
    else
        e.CanExecute = false;
}

```

6. Build and run. You should now have a functional toolbar without having written any additional procedural code!
7. Try to see any difference in appearance of a disabled toolbar button from one that is not disabled.
8. The disabled buttons are not grayed out, but a subtle difference is that the enabled buttons are highlighted while the mouse hovers over them.
9. Add tool tips. For example, this XAML will add a tool tip for the New button.

```

<Button Command="New"

```

```

        ToolTip="New">
        <Image Source="c:\OIC\Data\Graphics\new.png" />
    </Button>

```

10. Build and run. Now you can see an additional behavior difference with disabled buttons: the tool tip is not shown.

11. Enhance the menu items by providing an icon with image from the corresponding toolbar button. For example, this XAML will provide an icon for the New menu item.

```

<MenuItem Header="_New"
    Command="New">
    <MenuItem.Icon>
        <Image Source="c:\OIC\Data\Graphics\new.png"/>
    </MenuItem.Icon>
</MenuItem>

```

12. Add a combo box to the toolbar for specifying the font family. Also add a text box for specifying the font size. There should be a separator between the previous buttons and these new items on the toolbar.

```

<Separator/>
<ComboBox Name="cmbFontFamily"
    Width="150"
    SelectionChanged="cmbFont_SelectionChanged">
</ComboBox>
<TextBox Name="txtFontSize"
    Width="50"
    SelectionChanged="cmbFont_SelectionChanged">
</TextBox>

```

13. Provide code to initialize the combo box with the system font families and the text box with the current font size. You can use the same code that you had in connection with the font dialog box.

```

public Editor()
{
    InitializeComponent();
    foreach (FontFamily fam in Fonts.SystemFontFamilies)
        cmbFontFamily.Items.Add(fam.Source);
    txtFontSize.Text = txtData.FontSize.ToString();
    cmbFontFamily.Text = txtData.FontFamily.Source;
}

```

14. Implement the common handler for the **SelectionChanged** event of the combobox.

```

void cmbFont_SelectionChanged(object sender, RoutedEventArgs args)
{
    txtData.FontSize = Convert.ToDouble(txtFontSize.Text);
    txtData.FontFamily =
        new FontFamily(cmbFontFamily.SelectedItem.ToString());
}

```



15. Build and run. You get a crash, because the handler is called before **txtData** has been initialized.

16. Provide a **bool** data member **init**, set to false initially and to true after **InitializeComponent()** has been called.

```
private bool init = false;

public Editor()
{
    InitializeComponent();
    foreach (FontFamily fam in Fonts.SystemFontFamilies)
        cmbFontFamily.Items.Add(fam.Source);
    txtFontSize.Text = txtData.FontSize.ToString();
    cmbFontFamily.Text = txtData.FontFamily.Source;
    init = true;
}
```

17. In the **SelectionChanged** event handler, test that **init** is true.

```
if (init)
{
    txtData.FontSize = Convert.ToDouble(txtFontSize.Text);
    txtData.FontFamily =
        new FontFamily(cmbFontFamily.SelectedItem.ToString());
}
```

18. Build and run. You should now be able to change the font using the controls on the toolbar. Exercise all the functionality of your little editor.

## Part 2. Implement a Status Bar

1. In the XAML file add a StatusBar docked to the bottom.

```
<StatusBar DockPanel.Dock="Bottom"
           Name="statEditor">
</StatusBar>
```

2. Add three label controls to the status bar with content “Char”, “Line” and “Col”. There should be a separator between them.

```
<Label Name="lblChar" Width="60">
    Char
</Label>
<Separator/>
<Label Name="lblLine" Width="60">
    Line
</Label>
<Separator/>
<Label Name="lblCol" Width="60">
    Col
</Label>
```

3. Provide a handler for the **SelectionChanged** event of the **txtData** control.

```
<TextBox Name="txtData"
        TextWrapping="Wrap"
        AcceptsReturn="True"
        VerticalScrollBarVisibility="Auto"
        SelectionChanged="txtData_SelectionChanged">
</TextBox>
```

4. Implement the handler in the code-behind file. “Char” represents a zero-based index of the current character position of the insertion point, which can be found from the **SelectionStart** data member. “Line” represents the line number, starting from 1. This can be found via the **GetLineIndexFromCharacterIndex()** method. “Col” represents the column position starting from 1 in the current line. This can be found from the **GetCharacterIndexFromLineIndex()** method.

```
private void txtData_SelectionChanged(object sender, RoutedEventArgs
args)
{
    int iChar = txtData.SelectionStart;
    lblChar.Content = string.Format("Char {0}", iChar);
    int iLine = txtData.GetLineIndexFromCharacterIndex(iChar);
    lblLine.Content = string.Format("Line {0}", iLine + 1);
    int iCol = iChar - txtData.GetCharacterIndexFromLineIndex(iLine);
    lblCol.Content = string.Format("Col {0}", iCol + 1);
}
```

5. Build and run. Observe how the information in the status bar changes as you type or move about the insertion point.
6. As a final touch, add a new “StatusBar” menu item to the View menu. This is checkable and indicates whether the status bar is visible or not. It is initialized to be checked. There is a common handler for the **Checked** and **Unchecked** events.

```
<MenuItem Header="_StatusBar"
        IsCheckable="True"
        IsChecked="True"
        Checked="OnStatusChecked"
        Unchecked="OnStatusChecked">
</MenuItem>
```

7. Implement **OnStatusChecked()** in the code behind file. Be sure to test that **init** is true.

```
private void OnStatusChecked(object sender, RoutedEventArgs args)
{
    if (init)
    {
        MenuItem item = sender as MenuItem;
        if (item.IsChecked)
            statEditor.Visibility = Visibility.Visible;
        else
            statEditor.Visibility = Visibility.Collapsed;
    }
}
```

8. Build and run. Exercise the various features of your little editor, which should now be fully functional.



# **Chapter 10**

## **Data Binding**

# Data Binding

## Objectives

---

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

- **Understand what happens in a binding relationship between a source and a target object.**
- **Set up binding in procedural code and XAML.**
- **Understand how WPF monitors changes in the data source to keep the targets updated.**
- **Use a .NET collection as a source in data binding.**
- **Share a data source using data context.**
- **Improve a control's look by modifying data templates.**
- **Create a value converter for customizing the data source value.**
- **Take advantage of collection views.**
- **Use the data providers that ship with WPF.**
- **Use the visual data binding facilities of Visual Studio 2010.**
- **Bind to a database using the Entity Data Model.**

# What is Data Binding?

---

- **When talking about data binding, we can consider any arbitrary .NET object as *data* in WPF.**
- **The typical usage of data binding features is to provide a visual representation for some data.**
  - Data can be a collection object, an XML file or a database, for example.
  - An example of a visual representation could be a ComboBox with a list of countries that are stored in a collection.
  - A simple implementation approach would be to iterate through the collection and manually add each item to the ComboBox.
- **The WPF data binding implementation provides us some useful features just by setting up some information on the ComboBox in the situation above.**
  - We can “tell” the ComboBox to get the items list from the specified collection.
  - We can configure the binding so that the ComboBox will keep that list up-to-date, in case a new item is added to the collection.
  - Additionally it is possible to format the data, customizing the way the ComboBox shows it.

# Binding in Procedural Code

---

- **The binding is an open channel of communication between two properties.**
- **To set up binding in procedural code, we need:**
  - A binding object;
  - A data source: the .NET object from which the binding object will retrieve information;
  - A path: an optional information that helps the binding object in finding the information within the data source;
  - A target: we must tell the binding object which property in the target object will be connected to the data source in this binding relationship.

```
Binding binding = new Binding();  
binding.Source = cmbStates;  
binding.Path =  
    new PropertyPath("SelectedItem.Content");  
lblSelectedState.SetBinding(  
    Label.ContentProperty, binding);
```

- **In this example:**
  - We're using a ComboBox as source and a Label as target.
  - The path is SelectedItem.Content, which means that we want the binding object to use the text of the selected item in the ComboBox as the relevant source information.
  - Then the binding is set to a label, so that the selected item in the ComboBox will always be shown in the Label.



# Procedural Code Example

---

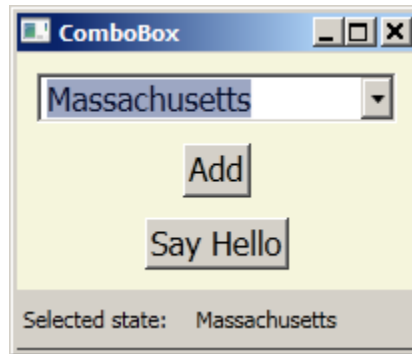
- Here is the XAML code for this example:
  - Open the solution in the **ComboBoxBinding\Step1** folder in the chapter directory.

```
<StackPanel
    DockPanel.Dock="Top"
    Background="Beige">
    <ComboBox
        Name="cmbStates"
        FontSize="16"
        HorizontalAlignment="Center"
        Margin="10"
        Width="180"
        IsEditable="True">
        <ComboBoxItem>California</ComboBoxItem>
        <ComboBoxItem>Massachusetts</ComboBoxItem>
        <ComboBoxItem>Illinois</ComboBoxItem>
    </ComboBox>
    <Button
        Name="btnAdd"
        FontSize="16"
        HorizontalAlignment="Center"
        Click="BtnAdd_Click">Add</Button>
    <Button
        Name="btnHello"
        FontSize="16"
        HorizontalAlignment="Center"
        Margin="10"
        Click="BtnHello_Click">Say Hello</Button>
</StackPanel>
<StatusBar DockPanel.Dock="Bottom">
    Selected state:
    <Label Name="lblSelectedState"></Label>
</StatusBar>
```

## Procedural Code Example (Cont'd)

---

- Build and run the application.



- Note that if you change the selected state in the ComboBox, the label *lblSelectedState* will be updated with the content of the selected item.
  - This is done automatically by WPF's binding mechanism.

# Binding in XAML

---

- **Setting up binding in XAML is easier than in procedural code, although the concepts are similar.**

- You just need to use a special binding markup in the target property.

```
<Label Name="lblSelectedState"
        Content="{Binding ElementName=cmbStates,
                          Path=SelectedItem.Content}">
</Label>
```

- **You can see a working example in the *ComboBoxBinding\Step2* folder in the chapter directory.**
  - Build and run the application.
  - Change the selected state to see the label content updated via binding.
- **In this example we used the Binding's *ElementName* argument, which is a reference to an existing element in XAML.**
  - This is the easiest way for binding in XAML.
- **We'll see alternative XAML markup extensions for Binding later, using:**
  - The **Source** argument, which is a reference to a resource.
  - The **RelativeSource** argument, which is a reference to an element by its relationship with the target. You can use, for example, the target object as the source.

# Binding to Plain .NET Properties

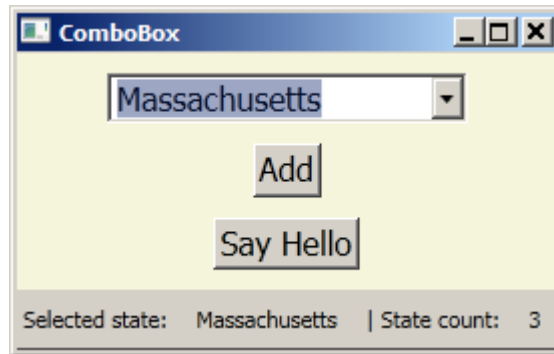
---

- **The examples so far worked so easily because the sources used were dependency properties.**
  - As we know, dependency properties have built-in change notification.
  - Hence, the target property in the binding relationship “knows” when the source’s value changes.
- **However, WPF supports any .NET property as a source in data-binding.**
  - But there is a caveat: as plain .NET properties don’t implement change notification, the target property won’t receive updates when the value of those properties changes.
- **For this reason, the .NET object used as source should implement the *INotifyPropertyChanged* interface.**
  - This interface simply contains an event called **PropertyChanged**.
  - The idea is to raise the **PropertyChanged** event whenever a property is modified, passing the property name as a parameter.

# Binding to .NET Properties Example

---

- See the example in the *ComboBoxBinding\Step3* folder in the chapter directory.
  - Build and run the solution.



- Notice the “State count” information in the bottom of the Window.
  - Try adding states to the ComboBox using the Add button to see the updated value.
- In the XAML code, the Label with the state count is bound to *cmbStates.Items.Count*.

```
<Label Name="lblStateCount"
        Content="{Binding ElementName=cmbStates,
                           Path=Items.Count}">
</Label>
```

## Binding to .NET Properties (Cont'd)

---

- **The property *Items.Count* is NOT a dependency property.**
  - So, why the label content gets updated when new items are added to the `ComboBox`?
- **Because *ComboBox.Items* is an *ItemsCollection*!**
  - The **`ItemsCollection`** class inherits from **`CollectionView`**, which implements the **`INotifyPropertyChanged`** interface.
  - For this reason, there is no need to implement anything else in this case.
- **WPF comes with some other useful classes that implement the *INotifyPropertyChanged* interface.**
  - For example, if you're using the **`Collection<T>`** class and want it to have change notification, just replace it by the **`ObservableCollection<T>`** class.

## Binding to a Collection

---

- **Our previous examples have a ComboBox with its items defined in XAML.**
  - However, frequently the list of ComboBox items comes from some data source such as a collection.
- **A collection object must be defined as a resource to be used as a data source.**
  - The following code binds the accounts collection to the ComboBox.
  - The ComboBox iterates automatically through the collection.
  - The **DisplayMemberPath** property tells the ComboBox which information from each account object to display.

```
<ComboBox Name="cmbAccounts" Margin="10" Width="96"
  SelectionChanged="CmbAccounts_SelectionChanged"
  ItemsSource="{Binding
    Source={StaticResource accounts}}"
  DisplayMemberPath="Name">
</ComboBox>
```

- **This is how the source collection is declared and added as a resource in procedural code.**

```
public ManageAccounts()
{
    accounts = new ObservableCollection<Account>();
    this.Resources.Add("accounts", accounts);

    InitializeComponent();
}
```

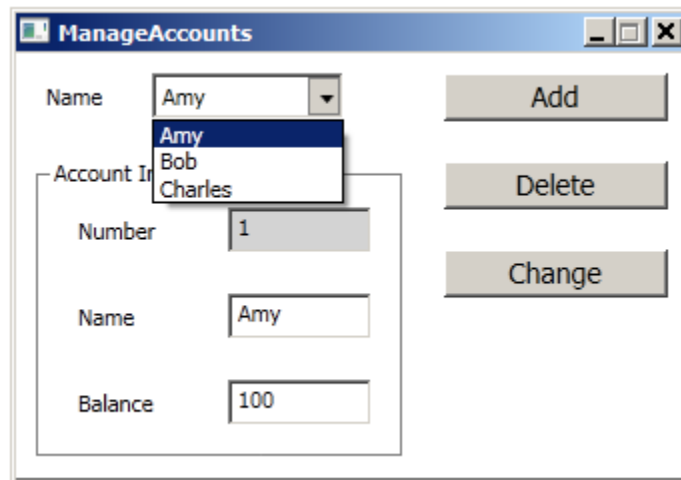
## Binding to a Collection Example

---

- See the example in the *CollectionBinding* folder in the chapter directory.
- When the window is loaded, some items are added to the collection in the procedural code.

```
NewAccount("Amy", 100m);  
NewAccount("Bob", 200m);  
NewAccount("Charles", 300m);
```

- Build and run the application to see the items in the ComboBox.



- Note that the ComboBox shows the name of the owner of each account.
  - That's because we've used the **DisplayMemberPath** property.
  - If this property is not set, the ComboBox will display the result of a **ToString()** call in the account object.



# Lab 10A

---

## Binding to a Collection

In this lab you will enhance our first binding example by using a collection to store the U.S. states shown in the ComboBox. You will declare the collection in the procedural code and use it in XAML, and will modify the Add button handler to update the collection instead of the **ComboBox.Items** collection.



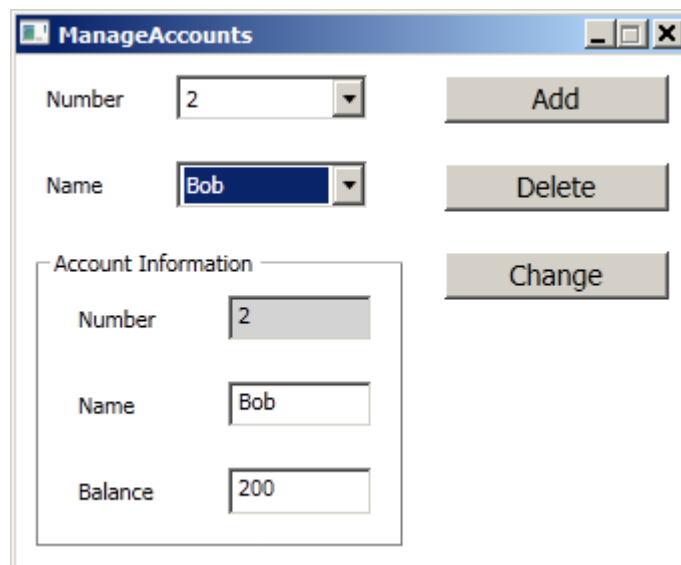
Detailed instructions are contained in the Lab 10A write-up at the end of the chapter.

Suggested time: 30 minutes

## Controlling the Selected Item

---

- **When binding a ComboBox to a collection, WPF keeps track of the selected item automatically.**
  - This is useful for implementing master/detail controls or for synchronizing different controls that are bound to the same collection.
  - WPF does this for any Selectors, such as ComboBox and ListBox, which are bound to a class that implements **IEnumerable**.
- **See the example in the *CollectionSync* folder in the chapter directory.**
  - This is our Account program, but now we can select the account using either the Number or the Name.
  - When the selected item in one of the combo boxes changes, the other is synchronized, having its selected item updated.



# ComboBox Synchronization Example

---

- **This synchronization feature is automatically provided by WPF!**
  - The synchronization of the combo boxes in this example is done by WPF when the **IsSynchronizedWithCurrentItem** property is set to True in both controls.
  - Note that both controls are bound to the same data source.

```
<StackPanel Orientation="Horizontal">
  <Label Margin="10" Width="50">Number</Label>
  <ComboBox
    Name="cmbNumbers" Margin="10" Width="96"
    SelectionChanged="CmbAccounts_SelectionChanged"
    ItemsSource="{Binding
                  Source={StaticResource accounts}}"
    DisplayMemberPath="Number"
    IsSynchronizedWithCurrentItem="True">
  </ComboBox>
</StackPanel>
<StackPanel Orientation="Horizontal">
  <Label Margin="10" Width="50">Name</Label>
  <ComboBox
    Name="cmbAccounts"
    Margin="10"
    Width="96"
    SelectionChanged="CmbAccounts_SelectionChanged"
    ItemsSource="{Binding
                  Source={StaticResource accounts}}"
    DisplayMemberPath="Name"
    IsSynchronizedWithCurrentItem="True">
  </ComboBox>
</StackPanel>
```

# Data Context

---

- **WPF supports specifying an implicit data source in a given scope for sharing purposes.**
- **This implicit data source is called a *data context*.**
  - When using a data context, there is no need to specify **ElementName**, **Source** or **RelativeSource** in the Binding markup.
- **The data context can be set on procedural code or in XAML.**
- **Any element inside the scope where the data context is defined can bind to it.**
  - The only thing that should be done is use the Binding object without specifying the source, and optionally set the Path.

# Data Context Demo

---

- **Let's show how to use data context with a demo.**
  - Open the solution in the **Demos\DataContext** folder.
- 1. Open the **MainWindow.xaml** file and examine the code. There is a ComboBox with two buttons for adding new items and greeting, and a status bar in the bottom. The ComboBox is bound to a collection defined in procedural code, in the file **MainWindow.xaml.cs**.
- 2. Let's bind the labels in the StatusBar to the ComboBox. Use the following code to accomplish this.

```
<StatusBar DockPanel.Dock="Bottom">
    Selected state:
    <Label Name="lblStatus"
           Content="{Binding ElementName=cmbStates,
                             Path=SelectedItem}">
    </Label>
    | State count:
    <Label Name="lblStateCount"
           Content="{Binding ElementName=cmbStates,
                             Path=Items.Count}">
    </Label>
</StatusBar>
```

## Data Context Demo (Cont'd)

---

3. Build and run the application. Notice that if you change the selected state, the **lblStatus** label will receive the update since **SelectedItem** is a dependency property. However, if you add more states, the ComboBox isn't updated because the source collection doesn't implement change notification. Let's fix this by changing the collection type.

```
...
public static ObservableCollection<String> states;

public ComboBox()
{
    states = new ObservableCollection<string>();
    ...
}
```

4. Build and run to see the result. Try adding a state to the ComboBox to see the updated list and the count in the **lblStateCount** Label.
5. Take a look again at the XAML code in the StatusBar, modified in step 2. The reference to the **cmbStates** element appears twice, and this situation suggests the use of data context. Add a **DataContext** property to the StatusBar referencing **cmbStates** and remove the **ElementName** argument from the bindings.

```
<StatusBar DockPanel.Dock="Bottom"
    DataContext="{Binding ElementName=cmbStates}">
    Selected state:
    <Label Name="lblStatus"
        Content="{Binding Path=SelectedItem}">
    </Label>
    | State count:
    <Label Name="lblStateCount"
        Content="{Binding Path=Items.Count}">
    </Label>
```

## Data Context Demo (Cont'd)

---

6. Build and run the application. Test it to see that the binding is working. Notice that the data context cannot be used outside the `StatusBar` scope.
  - The application at this point is saved in the **DataContext\Step2** folder in the chapter directory.
7. Let's modify our solution to have the same implementation, but using procedural code instead of XAML. To do this, remove the **DataContext** property from the `StatusBar` definition in the **ComboBox.xaml** file.
8. Now, add a name to the `StatusBar`, so that we'll be able to use it in the procedural code. The XAML code for the `StatusBar` will look like this:

```
<StatusBar DockPanel.Dock="Bottom"
           Name="statusBar">
```

9. Set the data context property of the `StatusBar` in the **MainWindow.xaml.cs** file, just after the **InitializeComponent()** method call.

```
public MainWindow()
{
    ...

    InitializeComponent();

    statusBar.DataContext = cmbStates;
}
```

10. Build and run the solution to see the result. There is a copy of this solution in the **DataContext\Step3** folder in the chapter directory.

# Data Templates

---

- **A *data template* is a powerful resource for customizing the view of a control.**
- **Binding can be used inside a data template definition.**
  - This feature provides freedom to customize the appearance of the data that is bound to the control.
- **Setting a data template for a control means changing its default visual tree.**
  - The visual elements used to build that control's default look will be replaced by a new visual structure.
- **For example, consider an ordinary ComboBox control from WPF.**
  - When you click it, a list of data bound items comes up in a plain text fashion.
  - Setting the **ComboBox.ItemsTemplate** property to a new visual structure will change that default look.
  - This new visual structure is simply a XAML code snippet!



# Data Template Example

---

- **Let's see how a control's look can be improved using data templates.**
  - As an example, we'll use a simple window with a ListBox for selecting a language.
  - This solution is saved on the **DataTemplate\Step1** folder in the chapter directory.



- **Here's how the ListBox looks after modifying its items' data template.**
  - Notice how the item's list default look has its visual structure changed.



## Specifying a Data Template

---

- The solution with the modified template is saved on the *DataTemplate\Step2* folder in the chapter directory.

- Open the solution and examine the ListBox code in the **MainWindow.xaml** file.

```
<ListBox Name="lstLanguages"
        ItemsSource="{Binding
            Source={StaticResource languages}}"
    SelectionChanged="lstLanguages_SelectionChanged">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Image Source="{Binding Path=Flag}"
                    Height="20">
            </Image>
            <Label Content="{Binding Path=Name}">
            </Label>
        </StackPanel>
    </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

- Notice the use of Binding inside the template definition.
- That original plain text template from the ListBox was replaced by this template with the flag image and the language name.

# Value Converters

---

- **While data templates modify the way data is rendered, *value converters* are used for modifying the source value.**
  - Sometimes the source and target values are from completely different data types.
  - For this reason, it may be useful to modify the source value to make it understandable at the target.
  - Even when the data types from source and target match, a value converter can be used to give a better format to the source value.
- **A value converter is a class that provides custom logic implementation between the source and target in the binding relationship.**
  - You can write custom code without giving up WPF's data binding features.
- **To create a value converter class you just need to implement a simple interface called *IValueConverter*.**
  - This interface has only two methods to be implemented: **Convert()** and **ConvertBack()**.

# Value Converter Example

---

- **Open the solution in the *ValueConverter* folder in the chapter directory.**
  - The example implements a value converter that takes a count as the source and returns a description string, like “3 users in the list”.
- **Here is how the value converter class looks like.**

```
public class SimpleCountToDescriptionConverter :
    IValueConverter
{
    public object Convert(object value, Type
targetType, object parameter, CultureInfo culture)
    {
        int count = int.Parse(value.ToString());
        if (count == 0)
            return "No user in the list";
        else if (count == 1)
            return count + " user in the list";
        else
            return count + " users in the list";
    }

    public object ConvertBack(object value, Type
targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

- **Note that we don’t implement the *ConvertBack()* method in this example, since we don’t need it.**

## Using a Value Converter in XAML

---

- **The converter must be defined in XAML as a resource.**

```
<Window x:Class="ValueConverter.MainWindow"
...
xmlns:local="clr-namespace:ValueConverter"
...>
<Window.Resources>
    <local:SimpleCountToDescriptionConverter
        x:Key="myConverter" />
</Window.Resources>
...
```

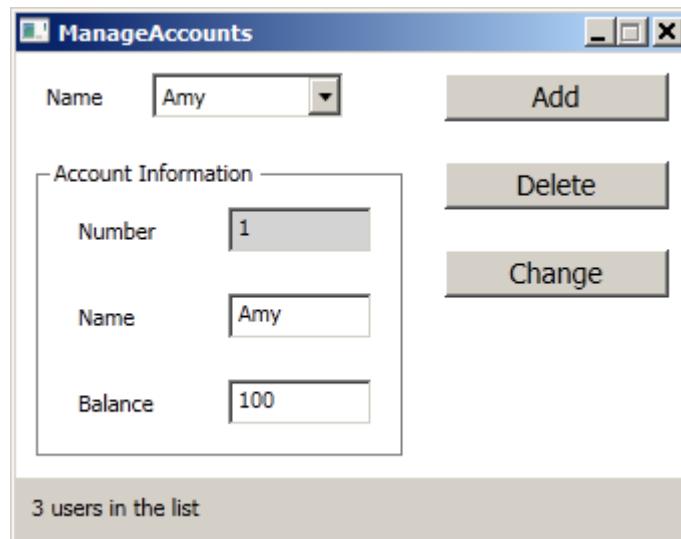
- **Then, the converter must be passed as a parameter to the Binding object.**
  - Every time the source value is updated, the Binding object passes it to the converter before it reaches the target.

```
<Label Content="{Binding
    ElementName=cmbAccounts,
    Path=Items.Count,
    Converter={StaticResource myConverter}}">
</Label>
```

## Value Converter Example (Cont'd)

---

- **Build and run the application to see the result.**



- **Note that the Label in the StatusBar, despite being directly bound to the Count property of the ComboBox, shows a description of the value.**
  - When the count is “3”, it shows “3 users in the list”.
  - When the count is “0”, it shows “No user in the list”.
  - Try removing items from the ComboBox using the delete button to see different values in the Label.
- **An interesting usage for value converters would be the implementation of a feature similar to the conditional formatting from MS Excel.**
  - In the previous example, you could bind the StatusBar’s **Background** property to the **Count** with a value converter that returns a highlighting Brush in case there is no user in the list.

# Collection Views

---

- In “Controlling the Selected Item” earlier in the chapter, we’ve seen that WPF has a mechanism to keep track of the selected item automatically when binding to a **Collection**.
- This is possible because WPF maintains a *view* between the source and target objects, whenever the source is a **Collection**.
  - The view is a class that implements the **ICollectionView** interface.
  - It stores the information about the current item and adds support for sorting, grouping and filtering.
- A source collection can have more than one view.
- A view can be shared among many targets, and changes to the view will automatically be seen by all the targets.

# Sorting

---

- ***ICollectionView* has a *SortDescriptions* property that allows sorting the items in the view.**
  - Only the view is sorted, as the collection remains untouched.

```
ICollectionView view =  
    CollectionViewSource.DefaultView(languages);  
view.SortDescriptions.Clear();  
view.SortDescriptions.Add(  
    new SortDescription("Name",  
        ListSortDirection.Ascending));
```

- **In the example above, the view is sorted in the ascending direction based on the *Name* property.**
- **Open the solution in the *CollectionView\Sorting* folder in the chapter directory.**
  - Notice in the file **MainWindow.xaml.cs** that the languages collection is initialized with non-sorted items, which are shown sorted when the application runs.





# Grouping

---

- **The collection views' grouping support is similar to sorting.**

- **ICollectionView** has a **GroupDescriptions** property that allows grouping the items in the view.
- Here is an example of grouping based on the **Region** property.

```
ICollectionView view =  
    CollectionViewSource.GetDefaultView(  
        this.FindResource("languages"));  
view.SortDescriptions.Clear();  
view.SortDescriptions.Add(  
    new SortDescription("Region",  
        ListSortDirection.Ascending));  
view.SortDescriptions.Add(  
    new SortDescription("Name",  
        ListSortDirection.Ascending));  
  
view.GroupDescriptions.Clear();  
view.GroupDescriptions.Add(  
    new PropertyGroupDescription("Region"));
```

- **For the grouping to be intuitive, it's important to sort the collection view first.**
  - The first **SortDescription** added must be the based on the same property used in the grouping.
  - Additional **SortDescriptions** will apply to the items within each group.

# Grouping Example

---

- **The effect of the grouping applied to the view is not automatically displayed.**
  - You must set the **GroupStyle** property of the **ItemsControl** class, which is inherited in list controls like **ListBox** and **ComboBox**.

```
<ListBox Name="lstLanguages"
...
>
<ListBox.GroupStyle>
  <GroupStyle>
    <GroupStyle.HeaderTemplate>
      <DataTemplate>
        <TextBlock
          Text="{Binding Path=Name}"
          FontWeight="Bold"
          Margin="0,10,0,0"
          HorizontalAlignment="Center"/>
        </DataTemplate>
      </GroupStyle.HeaderTemplate>
    </GroupStyle>
  </ListBox.GroupStyle>
...

```

- **In this example, we're setting a template for the header in the *GroupStyle*.**
  - Notice that **Binding** is being used to show the **Name** property of the group, that is **Region** in this case.
  - This code means that the **ItemsControl** will render a centered **TextBlock** with the region name when a language of a different region is about to be shown in the list.

## Grouping Example (Cont'd)

---

- The example is saved in the *CollectionView\Grouping* folder in the chapter directory.
  - Build and run the application to see the result of the grouping.



- Note that the list is ordered first by the region name, that is the grouping property, and then by the language name within each group.

# Filtering

---

- **It is possible to selectively remove items from a collection view based on some condition.**
  - This is done by using the **Filter** property of **ICollectionView**.
- ***ICollectionView.Filter* is a *Predicate<Object>* type.**
  - This means that the **Filter** is a delegate that returns a Boolean value depending on the Object passed as a parameter.
- **The delegate set in the *Filter* property will be called for each item in the source collection.**
  - It will determine whether the item should be shown or hidden by returning true or false.
  - Let's see an example using the anonymous delegate feature of C#.

```
view.Filter = delegate(object o)
{
    return ((o as Language).Region ==
           (WorldRegion)cmbFilter.SelectedItem);
};
```

- **This delegate is testing the filter condition:**
  - If the object passed as a parameter, which is a **Language**, is from the same **WorldRegion** as the current selected region in **cmbFilter**, it returns true.
  - In case the **Language** being tested is from a different region, the delegate returns false, which means that it'll be hidden.

# Filtering Example

---

- **Open the solution in the *CollectionView\Filtering* folder in the chapter directory.**
  - Notice the **cmbFilter** ComboBox, which handles the **SelectionChanged** event for filtering the languages available in the ListBox.
  - The **cmbFilter\_SelectionChanged()** method checks if a filter needs to be applied or set to null, based on **cmbFilter**'s selected item.
- **Build and run the application to see the working example.**
  - Try changing the Region Filter ComboBox and see how the filter is instantly applied to the collection view.



## Collection Views in XAML

---

- **The collection views we've seen so far were handled using procedural code.**
  - We used the default view created by WPF when a collection is a data binding source.
  - The **CollectionViewSource.GetDefaultView()** method was used in procedural code to get the default view.
- **However, we can use XAML to create a collection view to be used as a data source.**
  - The element used to create a collection view is **CollectionViewSource**.
  - We can add SortDescriptions, GroupDescriptions and Filters to its XAML syntax.

```
<Window.Resources>
  <CollectionViewSource x:Key="viewSource"
    Source="{StaticResource languages}">
    <CollectionViewSource.SortDescriptions>
      <componentModel:SortDescription
        PropertyName="Region"
        Direction="Ascending" />
    </CollectionViewSource.SortDescriptions>
    <CollectionViewSource.GroupDescriptions>
      <PropertyGroupDescription
        PropertyName="Region" />
    </CollectionViewSource.GroupDescriptions>
  </CollectionViewSource>
</Window.Resources>
```

## Collection Views in XAML Example

---

- Instead of binding the target to the collection directly, now we bind it to the collection view.

```
<ListBox Name="lstLanguages"
          ItemsSource="{Binding
                        Source={StaticResource viewSource}}}"
...

```

- To see a working example of using a collection view in XAML, open the example in the *XamlCollectionView* folder in the chapter directory.
  - Build and run the solution.



# Data Providers

---

- **As WPF supports binding to any arbitrary .NET object, it is possible to implement any data binding scenario using the appropriate objects.**
  - You could use an ADO.NET DataSet object to bind to a database.
  - Similarly you could bind to an Excel spreadsheet, the Windows Registry, and so on.
- **Additionally, WPF has two classes for providing a generic way of exposing common sources.**
  - **ObjectDataProvider**, which adds some features to the arbitrary .NET object binding in WPF.
  - **XmlDataProvider**, which provides an easy way to use a piece of XML data as a source.



# ObjectDataProvider

---

- The *ObjectDataProvider* class exposes a .NET object as a data binding source.
- Although it is possible to use a .NET object as a source without using *ObjectDataProvider*, this class adds some capabilities to the binding relationship:
  - Instantiate the source object with parameterized constructors.
  - Bind to a specific method in the source object.
  - Greater support for asynchronous binding.
- Let's see an example of an *ObjectDataProvider* definition in which we pass parameters to the object's constructor.
  - Open the solution in the **ObjectDataProvider** folder.
  - Examine the **ObjectDataProvider** definition, which passes four parameters to the constructor of the **Account** class, which can be found in the **Account.cs** file.

```
<Window.Resources>
  <ObjectDataProvider x:Key="accountDataProvider"
    ObjectType="{x:Type local:Account}">
    <ObjectDataProvider.ConstructorParameters>
      <sys:Int32>1</sys:Int32>
      <sys:String>Amy</sys:String>
      <sys:Decimal>980</sys:Decimal>
      <sys:Decimal>375</sys:Decimal>
    </ObjectDataProvider.ConstructorParameters>
  </ObjectDataProvider>
  ...
```

## ObjectDataProvider Example

---

- **Binding to an *ObjectDataProvider* is quite similar to binding to a .NET object.**

```
<Label Margin="10" Width="120" >
    Number:
</Label>
<TextBlock Margin="10" Width="50"
    VerticalAlignment="Center"
    Text="{Binding Source={StaticResource
        accountDataProvider}, Path=Number}">
</TextBlock>
```

- **We can use *ObjectDataProvider* to bind to a method from the *Account* class.**
  - In this example, we create an additional data provider that references to the instance of the previous **ObjectDataProvider**, and adds the **MethodName** property.
  - As you can see in the **Account.cs** file, the **GetConsolidatedBalance()** method returns the difference between the **Balance** and the **CreditCardBalance** properties.

```
<ObjectDataProvider
    x:Key="consolidationDataProvider"
    ObjectInstance="{StaticResource
        accountDataProvider}"
    MethodName="GetConsolidatedBalance" />
```

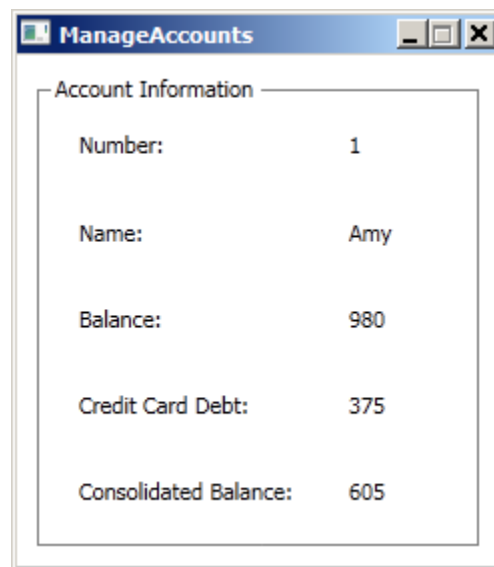
## ObjectDataProvider Example (Cont'd)

---

- When binding to the previous data provider, we don't use the *Path* argument.

```
<Label Margin="10" Width="120">
    Consolidated Balance:
</Label>
<TextBlock Margin="10" Width="50"
    VerticalAlignment="Center"
    Text="{Binding Source={StaticResource
        consolidationDataProvider}}">
</TextBlock>
```

- Build and run the application to see the result of these bindings.



# XmlDataProvider

---

- **The *XmlDataProvider* class provides an easy way to use a piece of XML data as a source.**
  - It can be an in-memory fragment or an external file, for example.
- **Let's see an example of using an external XML file as a data source.**
  - Open the solution in the **AccountManager\Step3** folder in the chapter directory.

```
<Window.Resources>
    <XmlDataProvider x:Key="dataProvider"
        XPath="Accounts" Source="AccountsData.xml"/>
</Window.Resources>
```

- **The *AccountsData.xml* file contains a list of accounts to be used in our example.**

```
<?xml version="1.0" encoding="utf-8" ?>
<Accounts xmlns="">
    <Account>
        <Number>1</Number>
        <Name>Amy</Name>
        <Balance>100</Balance>
    </Account>
    <Account>
        <Number>2</Number>
        <Name>Bob</Name>
        <Balance>200</Balance>
    </Account>
    ...
</Accounts>
```

## XmlDataProvider Example

---

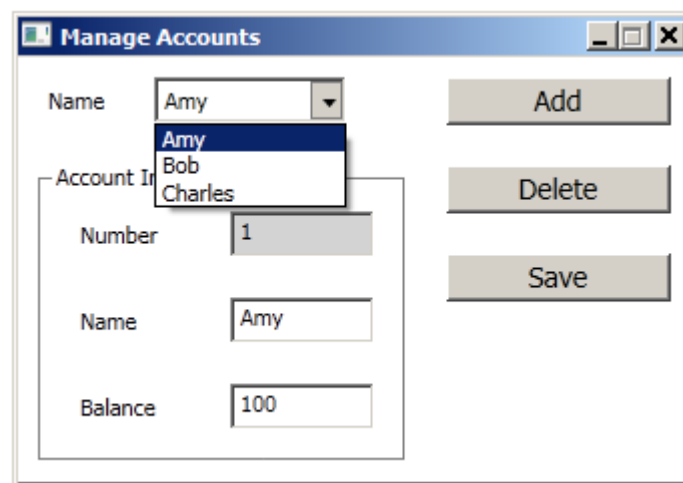
- **The application contains a ComboBox that is bound to the data provider.**

- Note the usage of **XPath** argument in the binding, so that the appropriate list can be found in the binding.

```
<ComboBox Name="cmbAccounts" Margin="10" Width="96"
  ItemsSource="{Binding Source={StaticResource
    dataProvider}, XPath=Account}"
  DisplayMemberPath="Name"
  SelectionChanged="cmbAccounts_SelectionChanged">
</ComboBox>
```

- **Build and run the application.**

- Note the account list bound to the ComboBox, showing only the name of each account.
- There is additional implementation for binding the text boxes to the XML data when an account is selected, which we'll see in detail in the Lab 10B.

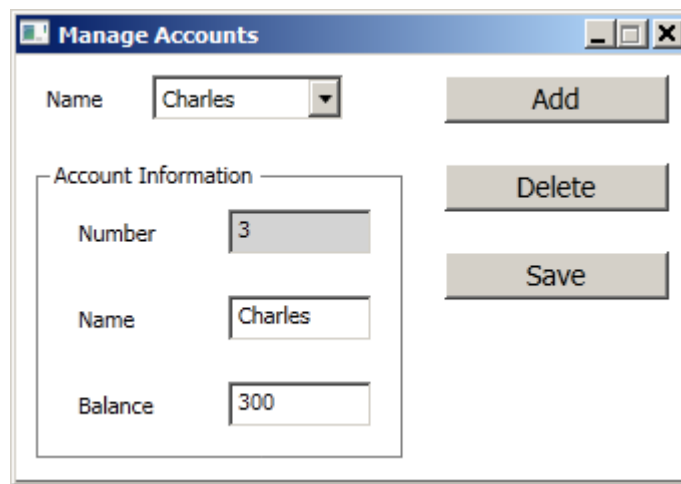


# Lab 10B

---

## An Account Manager Storing Data in XML

In this lab you will implement an account manager which relies on data from an external XML file. The starter code provides the interface, and you will provide code for XML binding, adding/removing accounts and saving data back to the XML file.



Detailed instructions are contained in the Lab 10B write-up at the end of the chapter.

Suggested time: 60 minutes

# Data Access with Visual Studio 2010

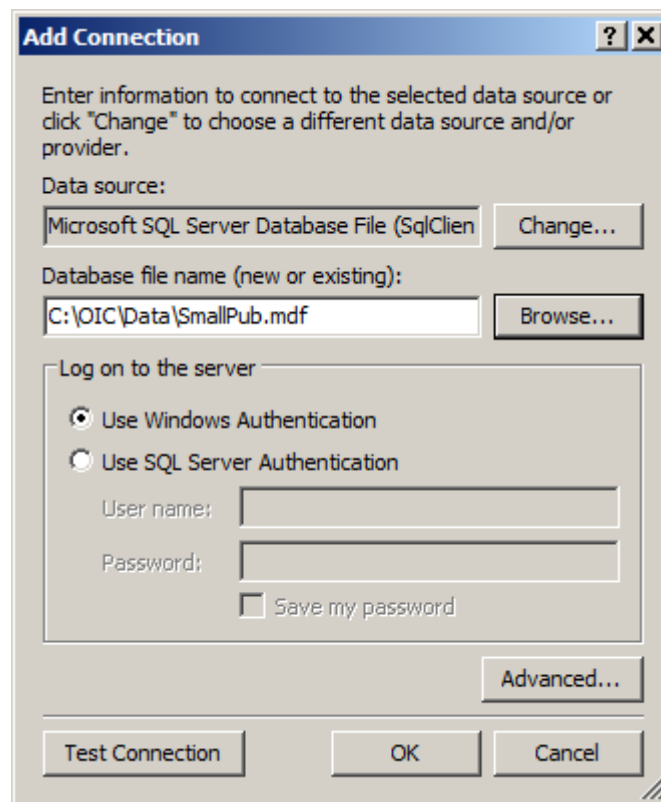
---

- **Visual Studio 2010 simplifies data access in your WPF applications with the Data Sources Window.**
  - Establish a data connection, such as to a SQL Server database.
  - Drag a data table onto the design surface.
  - Data bindings are set up automatically, using both XAML and code.
  - You can bind to an object, a dataset, an Entity Data Model, or to a WCF service.
- **You can use a variety of controls.**
  - TextBox, Label, ComboBox and TextBlock for individual items.
  - Lists using controls such as ListBox and DataGrid.
- **DataGrid is a new control in .NET 4.0 and functions almost exactly like the DataGrid in Silverlight.**
- **We'll illustrate this technology with example programs accessing the Book table in the SQL Server SmallPub database.**

# SmallPub Database

---

- **The SmallPub database contains information about books classified according to categories.**
  - It is stored in the file **SmallPub.mdf** in the **OIC\Data** folder.
  - Set up a connection to it in Server Explorer in Visual Studio.



- Click Test Connection. It should work!<sup>1</sup>

---

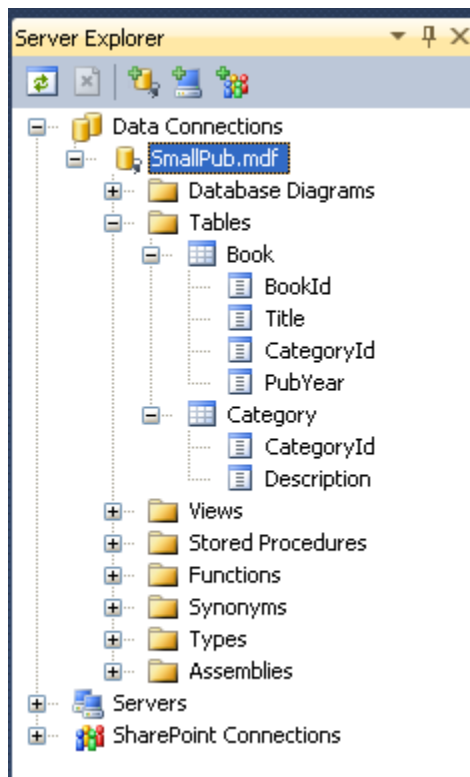
<sup>1</sup> If it fails, you may be able to work around the problem by modifying the connection. Click the Advanced button and try setting the User Instance to False.



## SmallPub Database (Cont'd)

---

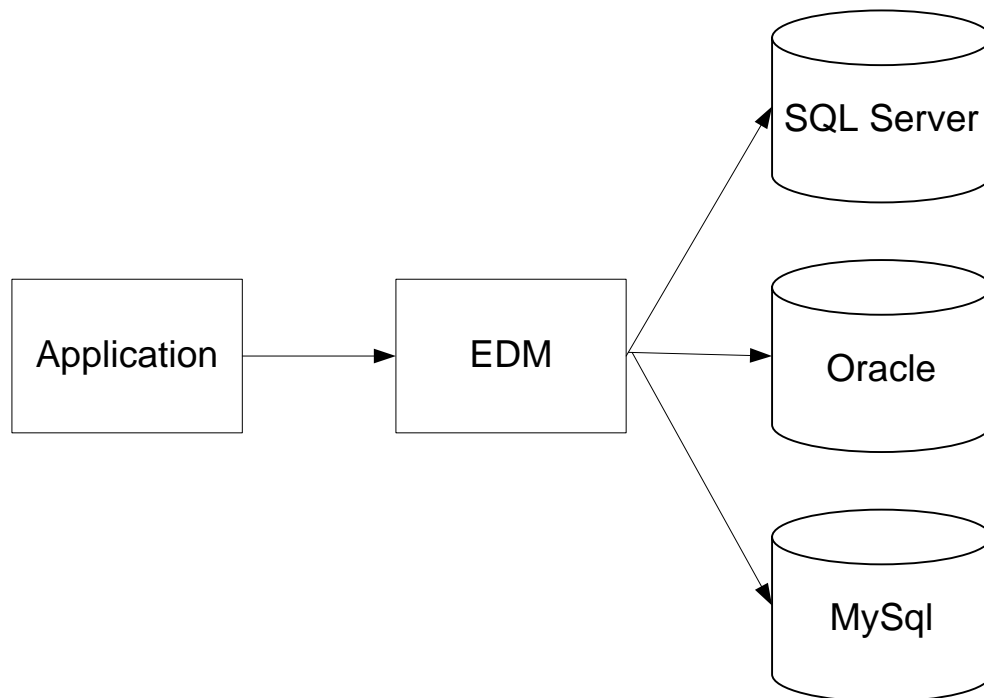
- When the connection is set up, you can examine the database in Server Explorer.



# ADO.NET Entity Framework

---

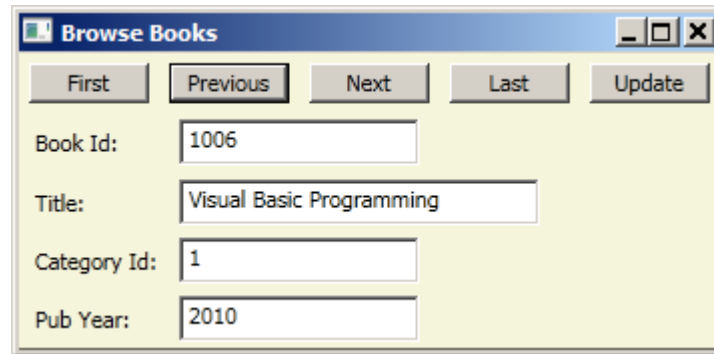
- **With the ADO.NET Entity Framework you program against a conceptual application model instead of directly against a database.**
  - The Entity Data Model (EDM) conceptually describes the structure of the data independently of how the data is actually stored.
  - The Entity Framework implements the EDM, translating between the model and the data store.
  - The Entity Framework uses ADO.NET data providers to talk to databases such as SQL Server, Oracle, MySql, and so on.



## Book Browser Demo

---

- The first example displays books in text boxes, allowing you to navigate through the **Book** table.
  - See **BrowseBooks\Step2** in the chapter directory.



- Open up the starter application in *BrowseBooks* in the **Demos** directory.
    - Starter code is backed up in **BrowseBooks\Step0**.
1. Open **MainWindow.xaml** in Design view. A user interface is provided for the buttons.

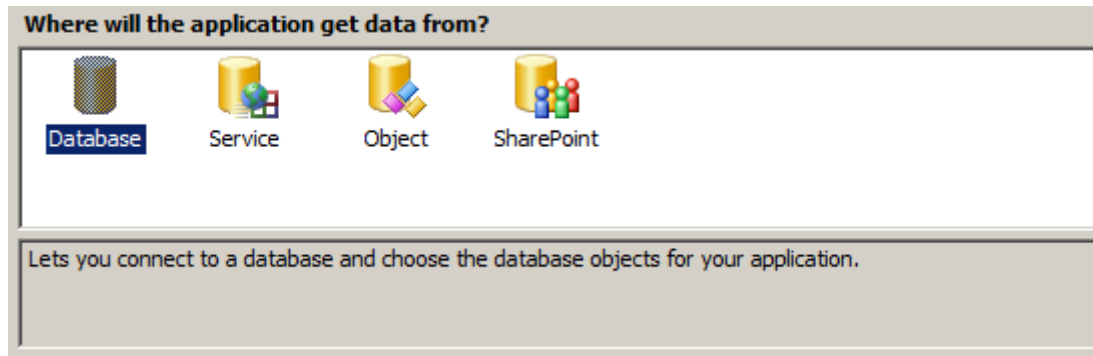


2. If the Data Sources windows is not showing, display it by the menu **Data | Show Data Sources**.

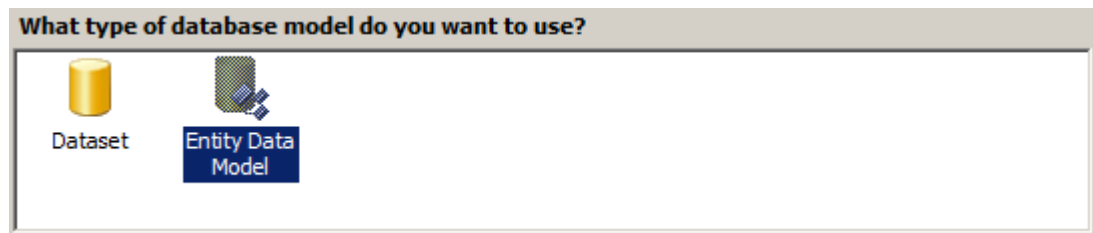
## Book Browser Demo (Cont'd)

---

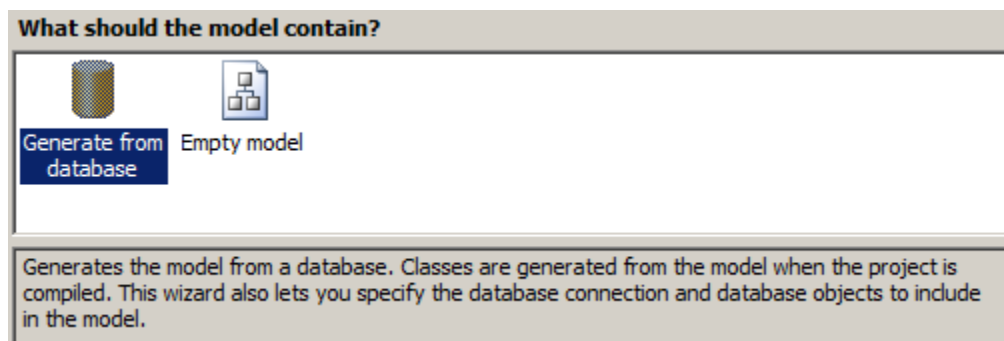
3. Click the link Add New Data Source, or use the menu Data | Add New Data Source.
4. Select Database for the source where the application will get its data. Click Next.



5. Select Entity Data Model as the database model. Click Next.



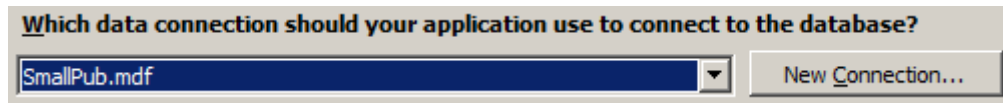
6. Select Generate from database. Click Next.



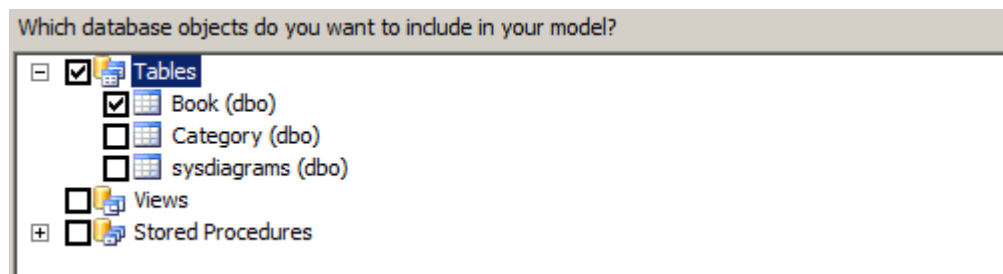
## Book Browser Demo (Cont'd)

---

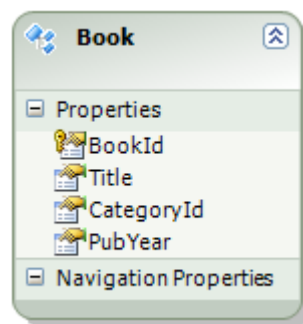
7. Select **SmallPub.mdf** as the connection. (If you had not previously set up this connection, click the New Connection button.) Click Next.



8. Say No to copying the file to your project.
9. Check only the Book table for the database objects you wish to include in your model. Click Finish.



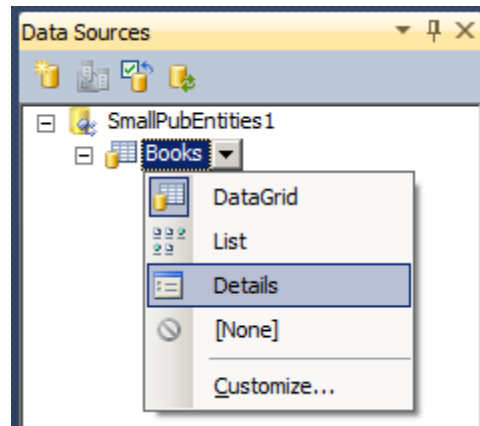
10. Observe that the model file **Model1.edmx** has been created, and you are shown a graphical representation of the Book entity.



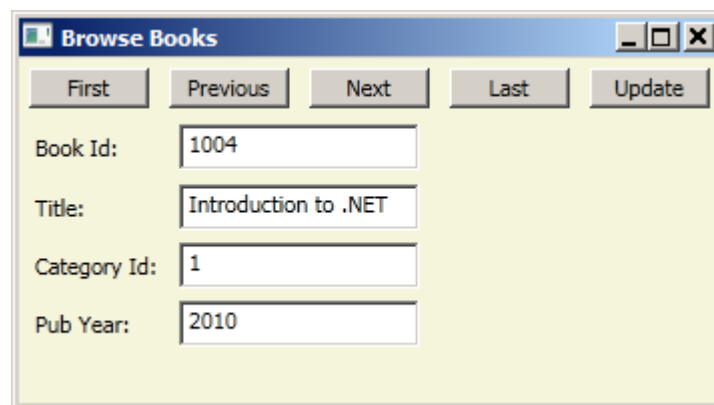
## Book Browser Demo (Cont'd)

---

11. Examine the Data Sources window. Choose Details from the dropdown next to Books.



12. Open **MainWindow.xaml** in Design view and drag the Books entity set to the area underneath the buttons. Label and TextBox controls have been generated.
13. Build and run the application. The first record is displayed, and you did not have to write any code yet. You are at Step 1.



14. Examine the XAML. You will see that data bindings have been created for you.
15. Examine the C# code. Query code has been created to access the data model.

## Book Browser Demo Completed

---

- **Step 2 provides complete code. We've renamed some variables and relied on *using* statements to remove explicit namespaces in class names.**
  - The key interface is **ICollectionView**, which is initialized in the handler of the windows's **Loaded** event.

```
private SmallPubEntities context;
private CollectionViewSource bvSource;
private ICollectionView bookView;

private ObjectQuery<Book>
GetBooksQuery(SmallPubEntities context)
{
    // Auto generated code

    ObjectQuery<Book> booksQuery = context.Books;
    // Returns an ObjectQuery.
    return booksQuery;
}

private void Window_Loaded(object sender,
RoutedEventArgs e)
{
    context = new SmallPubEntities();
    // Load data into Books. You can modify this
    // code as needed.
    bvSource = (CollectionViewSource)
        this.FindResource("booksViewSource");
    ObjectQuery<BrowseBooks.Book> booksQuery =
        this.GetBooksQuery(context);
    bvSource.Source =
        booksQuery.Execute(MergeOption.AppendOnly);
bookView = bvSource.View;
}
```

# Navigation Code

---

```
private void btnFirst_Click(object sender,
RoutedEventArgs e)
{
    bookView.MoveCurrentToFirst();
}

private void btnPrevious_Click(object sender,
RoutedEventArgs e)
{
    bookView.MoveCurrentToPrevious();
    if (bookView.IsCurrentBeforeFirst)
        bookView.MoveCurrentToFirst();
}

private void btnNext_Click(object sender,
RoutedEventArgs e)
{
    bookView.MoveCurrentToNext();
    if (bookView.IsCurrentAfterLast)
        bookView.MoveCurrentToLast();
}

private void btnLast_Click(object sender,
RoutedEventArgs e)
{
    bookView.MoveCurrentToLast();
}
```

- **Besides navigation we also provide for saving changes to the database.**

```
private void btnUpdate_Click(object sender,
RoutedEventArgs e)
{
    context.SaveChanges();
}
```



# DataGrid Control

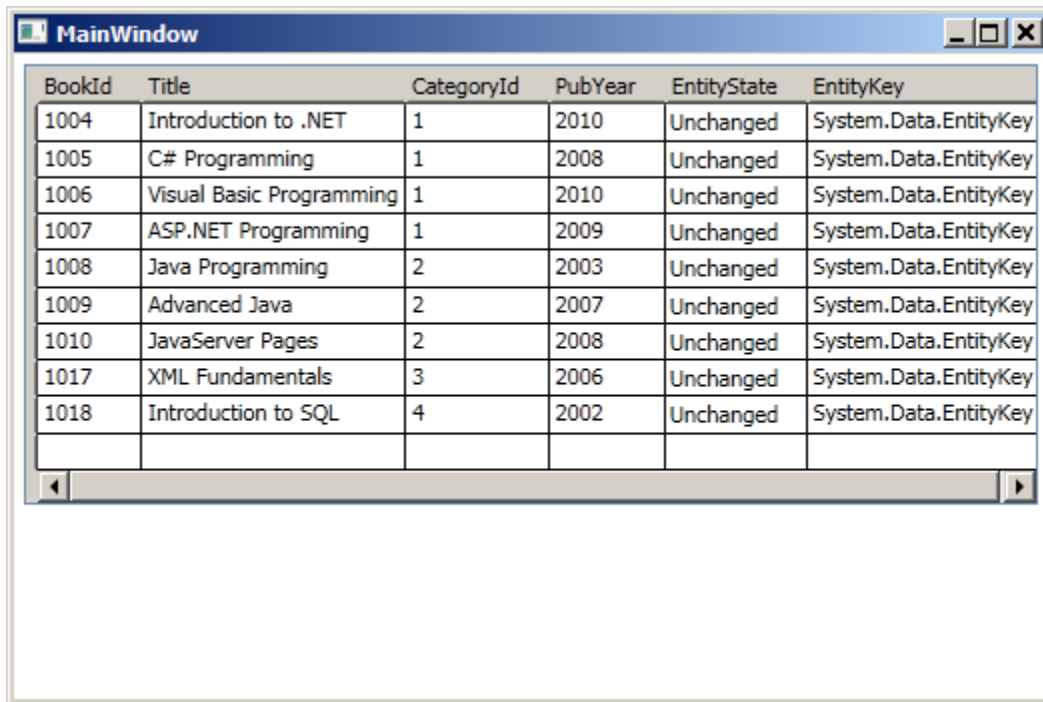
---

- .NET 4.0 provides a very useful *DataGrid* control, which has almost identical functionality to the corresponding Silverlight control.
- As a demo let's create from scratch a WPF application to display the Book table.
  1. Create a new WPF application **SimpleDataGrid**.
  2. Add a new data source like we did before to represent the Book table using the Entity Data Model
  3. Drag a DataGrid control from the Toolbox onto the design surface of your new application.
  4. Drag the Books entity set from the Data Sources window onto the DataGrid.
  5. Edit the XAML to set `AutoGenerateColumns` to `True`. Also set the `Name` property as shown, delete the explicit `Height` and `Width`, and set the `Margin` and `Name` as shown.

```
<Grid DataContext=
    "{StaticResource booksViewSource}">
  <DataGrid AutoGenerateColumns="True"
    HorizontalAlignment="Left" Margin="5"
    Name="dgBook" VerticalAlignment="Top"
    ItemsSource="{Binding}" />
</Grid>
```

6. Build and run. You will see the book data displayed in the DataGrid! See screen capture on the following page. We'd like to remove the columns `EntitySet` and `EntityKey`. Also, the main window is too big.

## DataGrid Control (Cont'd)



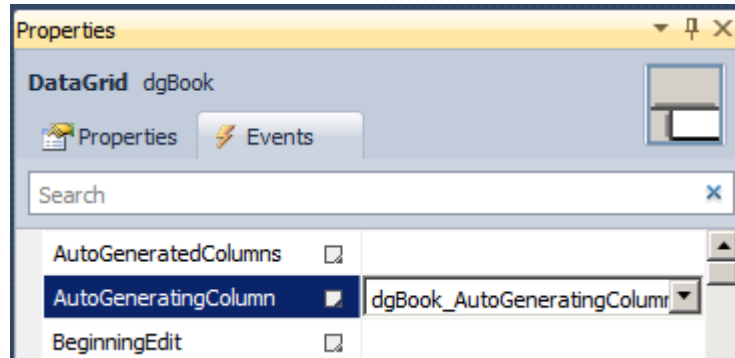
7. Edit the XAML for the main window to remove the explicit Height and Width and instead use the SizeToContent property. Also specify a suitable title.

```
<Window x:Class="SimpleDataGrid.MainWindow"
    ...
    Title="Simple DataGrid"
    SizeToContent="WidthAndHeight"
    ...
    Loaded="Window_Loaded">
```

8. Build and run. The DataGrid should now fit neatly in the main window without extra space except for the designated margin.

## DataGrid Control (Cont'd)

9. Now let's tackle the problem of the two columns we don't want. Add a handler for the DataGrid's AutoGeneratingColumn event.



10. In this handler cancel generating the column if the property is either EntityState or EntityKey.

```
private void dgBook_AutoGeneratingColumn(object
sender, DataGridAutoGeneratingColumnEventArgs e)
{
    if (e.PropertyName == "EntityKey" ||
        e.PropertyName == "EntityState"
    )
        e.Cancel = true;
}
```

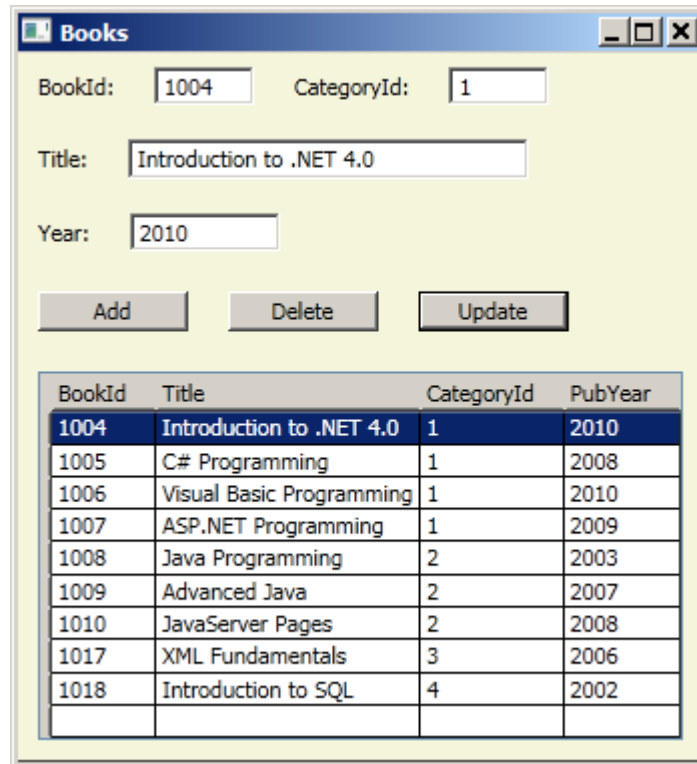
11. Build and run. We have the desired result!

BookId	Title	CategoryId	PubYear
1004	Introduction to .NET	1	2010
1005	C# Programming	1	2008
1006	Visual Basic Programming	1	2010
1007	ASP.NET Programming	1	2009
1008	Java Programming	2	2003
1009	Advanced Java	2	2007
1010	JavaServer Pages	2	2008
1017	XML Fundamentals	3	2006
1018	Introduction to SQL	4	2002

## Editing the Book Table

---

- Our final example provides the capability to insert, update and delete records from the Book table.
  - See **BookDemo** in the chapter directory.



BookId	Title	CategoryId	PubYear
1004	Introduction to .NET 4.0	1	2010
1005	C# Programming	1	2008
1006	Visual Basic Programming	1	2010
1007	ASP.NET Programming	1	2009
1008	Java Programming	2	2003
1009	Advanced Java	2	2007
1010	JavaServer Pages	2	2008
1017	XML Fundamentals	3	2006
1018	Introduction to SQL	4	2002

- In this example we did not use Visual Studio to create the data binding but implemented it in simple code.

```
private void ShowBooks()  
{  
    dgBook.ItemsSource = DB.GetBooks();  
    dgBook.SelectedIndex = 0;  
}
```

# Class Library

---

- **A class library implements the data access code.**

```
public class DB
{
    static SmallPubEntities context;

    static DB()
    {
        context = new SmallPubEntities();
        foreach (Book b in context.Books)
        {
            Debug.WriteLine(
                "{0} {1, -20} {2} {3}",
                b.BookId, b.Title, b.CategoryId,
                b.PubYear);
        }
    }

    public static List<Book> GetBooks()
    {
        List<Book> books = new List<Book>();
        var query = from Book b in context.Books
                     select b;
        foreach (var b in query)
            books.Add(b);
        return books;
    }
}
```

- **The query code uses LINQ to Entities.**
  - LINQ stands for Language Integrated Query, an important new data access technology from Microsoft.
- **The *App.config* file in the WPF project is copied from the class library project.**

# Database Updates

---

- **The class library provides methods for insert, update and delete.**

```
public static void update(int id, string title,
int catId, int year)
{
    // query for row(s) to be updated
    var query = from b in context.Books
                where b.BookId == id
                select b;

    // update matching rows
    foreach (var b in query)
    {
        b.Title = title;
        b.CategoryId = catId;
        b.PubYear = year;
    }
    // Save model to the database
    context.SaveChanges();
}
```

- **The WPF application calls these methods.**

```
private void btnUpdate_Click(object sender,
RoutedEventArgs e)
{
    int id = Int32.Parse(txtBookId.Text);
    int catId = Int32.Parse(txtCategoryId.Text);
    string title = txtTitle.Text;
    int year = Int32.Parse(txtYear.Text);
    DB.update(id, title, catId, year);
}
```

## Refreshing the DataGrid

---

- **In the case of an update, the data binding automatically takes care of refreshing the DataGrid.**
- **In the case of an insert or delete you must do this explicitly yourself.**

```
private void btnDelete_Click(object sender,
RoutedEventArgs e)
{
    int id = Int32.Parse(txtBookId.Text);
    DB.delete(id);
    Refresh();
}
```

- **In order to force the DataGrid to show new or changed data you need to set the *ItemsSource* property to null.**

```
private void Refresh()
{
    dgBook.ItemsSource = null;
    dgBook.ItemsSource = DB.GetBooks();
}
```

# Summary

---

- **Binding takes place as a relationship between a source and a target object.**
- **Binding can be set up in procedural code and XAML.**
- **WPF monitors changes in the data source to keep the targets updated.**
- **A .NET collection can be used as a source in data binding.**
- **A data source can be shared using data context.**
- **It's possible to improve a control's look by modifying data templates.**
- **Value converters are used for customizing the data source value.**
- **WPF comes with useful data providers such as *ObjectDataProvider* and *XmlDataProvider*.**
- **Visual Studio 2010 provides visual data binding facilities.**
- **You can bind to a database using the Entity Data Model.**



## Lab 10A

### Binding to a Collection

#### Introduction

In this lab you will enhance our first binding example by using a collection to store the U.S. states shown in the ComboBox. You will declare the collection in the procedural code and use it in XAML, and will modify the Add button handler to update the collection instead of the **ComboBox.Items** collection.



**Suggested Time:** 30 minutes

**Root Directory:** OIC\WpfCs

**Directories:**      **Labs\Lab10A\ComboBoxBinding** (do your work here)  
                          **Chap10\ComboBoxBinding\Step3** (backup of starter code)  
                          **Chap10\ComboBoxBinding\Step4** (answer)

1. Build and run your starter code. Notice that you can add new states by typing in the name of the state in the ComboBox and clicking the Add button. There is a label with the selected state in the bottom, which is updated when the selected index of the ComboBox is modified. There is another label with the state count, which is updated when new states are added to the list.
2. Let's create our collection that will be used as a source by the **cmbStates** ComboBox. Declare a new **states** collection of type **Collection<String>** in the **MainWindow.xaml.cs** file. You'll need to import the namespace **System.Collections.ObjectModel**.

```
...
using System.Collections.ObjectModel;
```

```
namespace ComboBoxBinding
```

```

{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Collection<String> states;

        public ComboBox()
        {
            ...

```

- Now, let's populate the newly created collection with the values that are currently being defined in the ComboBox XAML code. Besides adding these values to the collection, we need to add it to the Window Resources collection to allow it to be visible by the XAML elements.

```

public MainWindow()
{
    states = new Collection<string>();
    states.Add("California");
    states.Add("Massachusetts");
    states.Add("Illinois");
    this.Resources.Add("states", states);

    InitializeComponent();
}

```

- In the **MainWindow.xaml** file, remove the list of ComboBox item elements defined in XAML for the **cmbStates** ComboBox. Include the **ItemsSource** property defining a Binding to the states collection, which can be referenced as a resource here.

```

<ComboBox
    Name="cmbStates"
    FontSize="16"
    HorizontalAlignment="Center"
    Margin="10"
    Width="180"
    IsEditable="True"
    ItemsSource="{Binding Source={StaticResource states}}"/>
</ComboBox>

```

- Build and run the application. Notice that the ComboBox has its **Items** collection attached to the states collection defined in the procedural code. The state count label appears to be showing the correct information, but the label for the selected state stopped working. Additionally, if you try to add a new state using the Add button, you'll hit a runtime exception. Let's first fix the add item feature by modifying the **Click** handler of the button to add the new item to the states collection, instead of the **ComboBox.Items** collection.

```

private void BtnAdd_Click(object sender, RoutedEventArgs e)
{

```

```

        states.Add(cmbStates.Text);
        cmbStates.SelectedIndex = cmbStates.Items.Count - 1;
    }

```

6. Build and run. Try adding a new item to the ComboBox. Despite not hitting any exception, there is no visible update to the **ComboBox.Items** or the state count label. Actually, the states collection was updated in the procedural code, but this update is not “seen” by the controls in XAML because the **Collection<T>** class doesn’t implement the **INotifyPropertyChanged** interface. This can be easily fixed by changing the collection type to **ObservableCollection<T>**, which implements the **INotifyPropertyChanged** interface.

```

public partial class MainWindow: Window
{
    private ObservableCollection<String> states;

    public ComboBox()
    {
        states = new ObservableCollection<string>();
        ...
    }
}

```

7. Build and run the application, and try adding a new state again. You’ll notice the new state added to the **ComboBox.Items** list and the new state count in the bottom label. However, the label for the selected state still doesn’t get updated. If you take a look at the **Path** property of the binding in this label, you’ll see that it is bound to the **SelectedItem.Content** property of the ComboBox, which was right for the previous **Items** definition of the control using the **ComboBoxItem** elements. Considering that the ComboBox is now bound to a collection of strings, each item of the ComboBox is actually a simple string object, and the **Path** property of the label binding can be simply **SelectedItem**.

```

<Label Name="lblSelectedState"
        Content="{Binding ElementName=cmbStates, Path=SelectedItem}">
</Label>

```

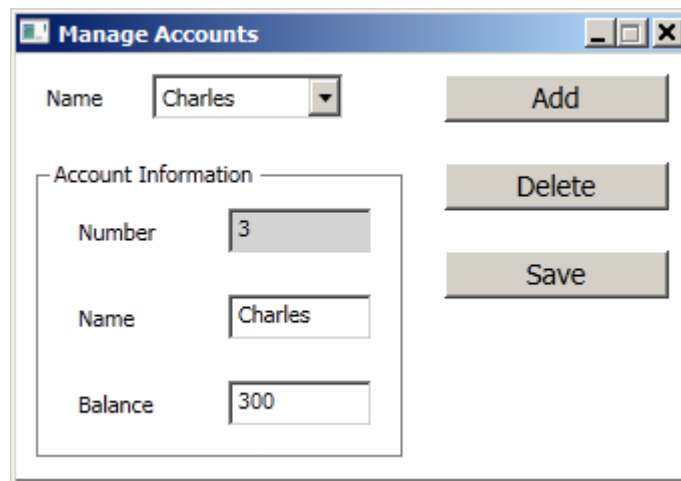
8. Build and run the application. Now the selected item label is working properly, and you can test the other window features to see that the program is fully functional.

## Lab 10B

### An Account Manager Storing Data in XML

#### Introduction

In this lab you will implement an account manager which relies on data from an external XML file. The starter code provides the interface, and you will provide code for XML binding, adding/removing accounts and saving data back to the XML file.



**Suggested Time:** 60 minutes

**Root Directory:** OIC\WpfCs

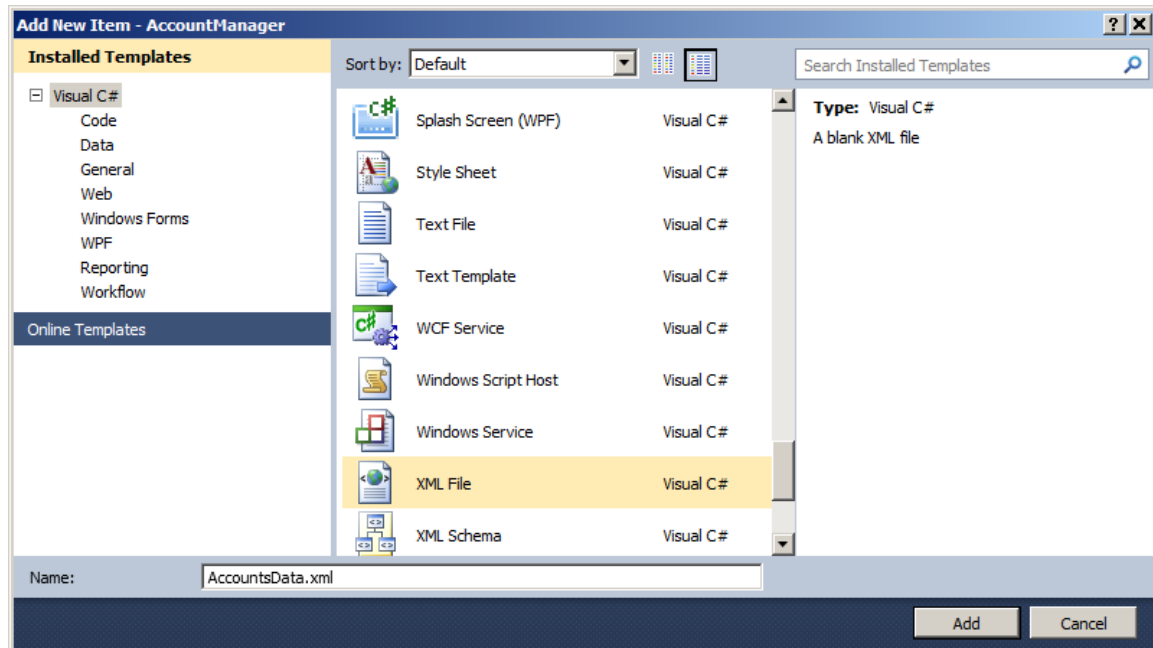
**Directories:**

<b>Labs\Lab10B\AccountManager</b>	(do your work here)
<b>Chap10\AccountManager\Step0</b>	(backup of starter code)
<b>Chap10\AccountManager\Step1</b>	(answer to part 1)
<b>Chap10\AccountManager\Step2</b>	(answer to part 2)
<b>Chap10\AccountManager\Step3</b>	(answer to part 3)

#### Part 1. Bind the ComboBox to the XML file

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. As the first thing we need here is data, let's add an XML file to the project. Right-click the project name in solution explorer and select the Add submenu, then click on the New Item... option.

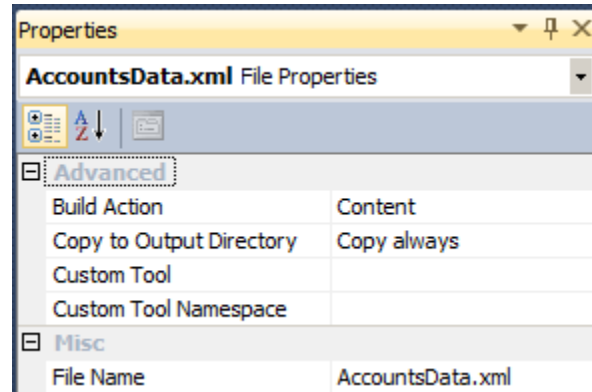
- From the Add New Item dialog, select the XML File template and provide **AccountsData.xml** as the file name. Click Add.



- Provide the following initial data in the new XML file.

```
<?xml version="1.0" encoding="utf-8" ?>
<Accounts xmlns="">
  <Account>
    <Number>1</Number>
    <Name>Amy</Name>
    <Balance>100</Balance>
  </Account>
  <Account>
    <Number>2</Number>
    <Name>Bob</Name>
    <Balance>200</Balance>
  </Account>
  <Account>
    <Number>3</Number>
    <Name>Charles</Name>
    <Balance>300</Balance>
  </Account>
</Accounts>
```

- The purpose of the newly created XML file is to be an external source of data for our program. Thus, it would be nice to have this XML file in the same directory of the application executable, as a content resource (not embedded). To achieve this, modify the file properties by right-clicking it and selecting Properties from the menu. Then, change Build Action to Content and Copy to output Directory to Copy Always.



5. Build the application. Then, go to the **bin\Debug** directory and note that **AccountsData.xml** is there. Now our file is ready to be used!
6. Define a **XmlDataProvider** resource in XAML pointing to the XML file. Give this resource a name using the **x:Key** property so that it can be referenced. Additionally, provide “Accounts” as the value for the **XPath** property, as this is the parent node of the accounts list in our XML file.

```
<Window x:Class="AccountManager.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ManageAccounts" SizeToContent="WidthAndHeight"
  ResizeMode="CanMinimize">
  <Window.Resources>
    <XmlDataProvider x:Key="dataProvider"
      XPath="Accounts"
      Source="AccountsData.xml" />
  </Window.Resources>
  ...

```

7. Now, let’s use the data provider in the ComboBox to obtain the accounts list. Using the ComboBox’s **ItemsSource** property, set a binding to the **dataProvider** resource.

```
<ComboBox
  Name="cmbAccounts"
  Margin="10"
  Width="96"
  ItemsSource="{Binding Source={StaticResource dataProvider},
    XPath=Account}"
>
</ComboBox>

```

8. Build and run the application. If you click on the ComboBox, you’ll notice an unwanted behavior: the information contained in the list seems concatenated, since the ComboBox is trying to show all the text from each item, which are XML nodes. This has happened because we didn’t tell the ComboBox which information from each XML node should be used to display it. Let’s fix this behavior by adding the **DisplayMemberPath** property.

```
<ComboBox
    Name="cmbAccounts"
    Margin="10"
    Width="96"
    ItemsSource="{Binding Source={StaticResource dataProvider},
XPath=Account}"
    DisplayMemberPath="Name"
>
</ComboBox>
```

9. Now, if you build and run the application, the list should be displayed properly by the ComboBox. You have successfully bound the ComboBox to an XML data source!
10. A small tweak is to display the first element in the ComboBox, at index 0. Add a Loaded event handler for the main window and set the SelectedIndex to 0.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    cmbAccounts.SelectedIndex = 0;
}
```

11. Build and run. The program at this point is saved in the **AccountManager\Step1** folder in the chapter directory.

## Part 2. Bind the Text Boxes to the XML File Based on the Selected Item

1. Now we must add logic to populate the text boxes with account information based on the currently selected account from the ComboBox. We'll accomplish this by simply reading the XML node from the **ComboBox.SelectedItem** property. Let's start by defining a function to be responsible for binding the text boxes and the source XML node. This node will be passed as a parameter to this function, and you'll need to import the **System.Xml** namespace into the class file. Note that the binding code for each TextBox does pretty much the same thing as the XAML binding code for the ComboBox.

```
private void BindAccountInformation(XmlNode account)
{
    Binding numberBinding = new Binding();
    numberBinding.Source = account;
    numberBinding.XPath = "Number";
    txtNumber.SetBinding(TextBox.TextProperty, numberBinding);

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

    Binding balanceBinding = new Binding();
    balanceBinding.Source = account;
    balanceBinding.XPath = "Balance";
    txtBalance.SetBinding(TextBox.TextProperty, balanceBinding);
}
```

2. Add a handler for the **SelectionChanged** event of the ComboBox.

```
<ComboBox
  Name="cmbAccounts"
  Margin="10"
  Width="96"
  ItemsSource="{Binding Source={StaticResource dataProvider},
XPath=Account}"
  DisplayMemberPath="Name"
  SelectionChanged="cmbAccounts_SelectionChanged"
>
</ComboBox>
```

3. In procedural code, add implementation to the handler. It must get the Xml node from the **SelectedItem** property of the ComboBox and call the recently created **BindAccountInformation()** method passing it as a parameter. It will be a nice protection if you clear the text boxes and its bindings in case there is no item selected in the ComboBox.

```
private void cmbAccounts_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    int index = cmbAccounts.SelectedIndex;
    if (index != -1)
    {
        XmlElement elem = (XmlElement)cmbAccounts.Items[index];
        BindAccountInformation(elem);
    }
    else
    {
        BindingOperations.ClearAllBindings(txtNumber);
        txtNumber.Text = "";
        BindingOperations.ClearAllBindings(txtName);
        txtName.Text = "";
        BindingOperations.ClearAllBindings(txtBalance);
        txtBalance.Text = "";
    }
}
```

4. Build and run the application. Change the selected account using the ComboBox to see the data changing in the text boxes, which is the expected behavior. However, let's assume that it is important to protect the Number property, avoiding changes to its value. To do so, set the **IsReadOnly** property of the **txtNumber** TextBox to True and the **Background** to LightGray, to give it the visual aspect of a read only field.

```
<TextBox
  Name="txtNumber"
  Margin="10"
  Width="72"
  IsReadOnly="True"
  Background="LightGray"
>
</TextBox>
```



### Part 3. Implement the Action Buttons

1. In Part 2, you provided binding for the text boxes and an XML node that we got from the ComboBox, which has the entire XML document as data source. The interesting thing is that the WPF binding mechanism takes care of updating this XML data source when we change the data in the text boxes. However, these changes are not saved to the external XML file because the data source keeps only an in-memory copy of the document. To save this copy into the external XML file, we can add code to the Save button in our program by using the **XmlDataProvider.Document.Save()** method. Double-click the Save button to add an event handler and provide code for saving the file.

```
private void btnSave_Click(object sender, RoutedEventArgs e)
{
    try
    {
        XmlDataProvider dp =
            this.FindResource("dataProvider") as XmlDataProvider;
        dp.Document.Save("AccountsData.xml");
        MessageBox.Show("The data was successfully saved" , "Accounts");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Accounts");
    }
}
```

2. Build and run the application. Try modifying some values in the text boxes for some accounts, and click the Save button. Close the application and go to the **bin\Debug** folder in the project directory, and open the **AccountsData.xml** file and note that your changes were saved! If you open the application again, you'll see that your modified data was persisted. (Don't rebuild the application, or the starting data will be copied over the modified data.)
3. Now let's add code for the Add button. Before creating a new account, we'll need to provide a valid account number, which can be an increment of the biggest account number that currently exists in the list. Provide a method that searches the accounts list for the biggest account number and return this number plus 1 as a suggestion for the new account number. The method name should be **GetNextAccountNumber()**.

```
private int GetNextAccountNumber()
{
    int maxAccountNumberFound = 0;

    foreach (XmlNode item in cmbAccounts.Items)
    {
        int currentNumber =
            Convert.ToInt32(item.SelectSingleNode("Number").InnerText);
        if (currentNumber > maxAccountNumberFound)
            maxAccountNumberFound = currentNumber;
    }
    return maxAccountNumberFound + 1;
}
```

```
}
```

4. To create a new account, what you need to do is basically create a new XML node with a blank account and add it to the XML document contained in the data provider used by the ComboBox. By doing this, the ComboBox will be updated with this new item automatically, and your last step will be to select this new item's index so that the user can edit the data using the text boxes.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    try
    {
        XmlDataProvider dp =
            this.FindResource("dataProvider") as XmlDataProvider;
        XmlNode accountNode =
            dp.Document.CreateNode(XmlNodeType.Element, "Account", "");

        XmlNode numberNode =
            dp.Document.CreateNode(XmlNodeType.Element, "Number", "");
        numberNode.InnerText = GetNextAccountNumber().ToString();
        accountNode.AppendChild(numberNode);

        XmlNode nameNode =
            dp.Document.CreateNode(XmlNodeType.Element, "Name", "");
        nameNode.InnerText = "New name";
        accountNode.AppendChild(nameNode);

        XmlNode balanceNode =
            dp.Document.CreateNode(XmlNodeType.Element, "Balance", "");
        balanceNode.InnerText = "0";
        accountNode.AppendChild(balanceNode);

        dp.Document.DocumentElement.AppendChild(accountNode);

        cmbAccounts.SelectedIndex = cmbAccounts.Items.Count - 1;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Accounts");
    }
}
```

5. Finally, add code for the Delete button. One solution you could use to delete an item is to get the XML node contained in the **SelectedItem** property and then call the **RemoveChild()** method in the XML Document, which will result in an immediate update in the ComboBox.

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
{
    try
    {
        int index = cmbAccounts.SelectedIndex;
        if (index != -1)
        {
            XmlDataProvider dp =
```

```
        this.FindResource("dataProvider") as XmlDataProvider;  
        XmlElement elem = (XmlElement)cmbAccounts.Items[index];  
        dp.Document.DocumentElement.RemoveChild(elem);  
        cmbAccounts.SelectedIndex = 0;  
    }  
}  
catch (Exception ex)  
{  
    MessageBox.Show(ex.Message, "Accounts");  
}  
}
```

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

