

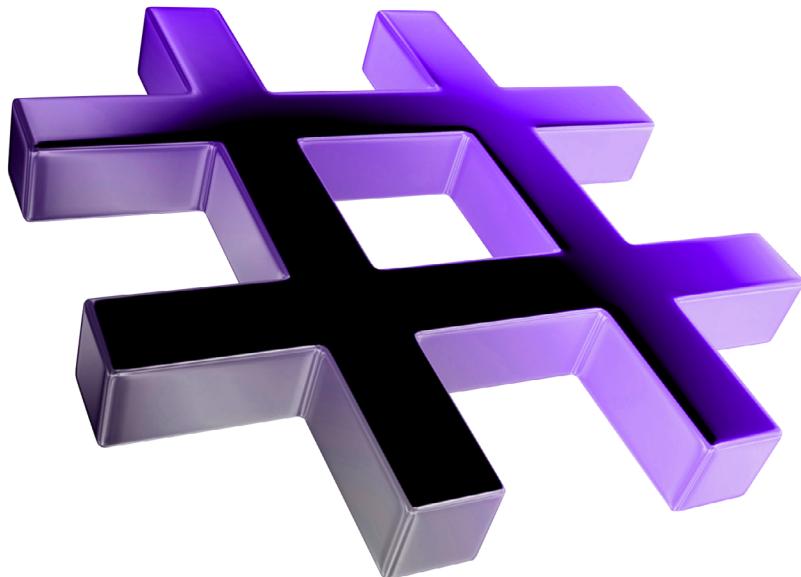
# GUI with Windows Presentation Foundation

# 25

## Objectives

In this chapter you'll:

- Define a WPF GUI with Extensible Application Markup Language (XAML).
- Handle WPF user-interface events.
- Use WPF's commands feature to handle common tasks such as cut, copy and paste.
- Customize the look-and-feel of WPF GUIs using styles and control templates.
- Use data binding to display data in WPF controls.





# Outline

- |   |  |
|---|--|
| <p><b>25.1</b> Introduction</p> <p><b>25.2</b> Windows Presentation Foundation (WPF)</p> <p><b>25.3</b> Declarative GUI Programming Using XAML</p> <p><b>25.4</b> Creating a WPF App</p> <p><b>25.5</b> Laying Out Controls</p> <ul style="list-style-type: none"> <li>25.5.1 General Layout Principles</li> <li>25.5.2 Layout in Action</li> </ul> <p><b>25.6</b> Event Handling</p> | <p><b>25.7</b> Commands and Common Application Tasks</p> <p><b>25.8</b> WPF GUI Customization</p> <p><b>25.9</b> Using Styles to Change the Appearance of Controls</p> <p><b>32.10</b> Customizing Windows</p> <p><b>25.11</b> Defining a Control's Appearance with Control Templates</p> <p><b>25.12</b> Data-Driven GUIs with Data Binding</p> <p><b>25.13</b> Wrap-Up</p> |
|---|--|

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 25.1 Introduction

Microsoft has three active GUI technologies—Windows Forms, Windows Presentation Foundation (WPF) and the Universal Windows Platform (UWP). In Chapters 14–15, you built GUIs using Windows Forms. In this chapter, you’ll build GUIs using **Windows Presentation Foundation (WPF)**, which—unlike Windows Forms—is completely customizable. In Chapter 26, WPF Graphics and Multimedia, you’ll learn how to incorporate 2D graphics, 3D graphics, animation, audio and video in WPF apps. Microsoft’s current and future direction is the Universal Windows Platform (UWP), which is designed to provide a common platform and user experience across all Windows devices, including personal computers, smartphones, tablets, Xbox and even Microsoft’s new HoloLens virtual reality and augmented reality holographic headset—all using nearly identical code. We’re moving to UWP as well and will provide two online UWP chapters. For that reason, we’ve included our WPF chapters *as is* from this book’s previous edition and we will no longer be updating our WPF treatment.

We begin with an introduction to WPF. Next, we discuss an important tool for creating WPF apps called **XAML** (pronounced “zammel”)—**Extensible Application Markup Language**. XAML is an XML vocabulary for defining and arranging GUI controls without any C# code. Because XAML is an XML vocabulary, you should understand the basics of XML before learning XAML and WPF. We introduce XML in Sections 24.2–24.4.

Section 25.3 demonstrates how to define a WPF GUI with XAML. Sections 25.4–25.7 demonstrate the basics of creating a WPF GUI—layout, controls and events. You’ll also learn capabilities of WPF controls and event handling that are different from those in Windows Forms. WPF allows you to easily customize the look-and-feel of a GUI beyond what is possible in Windows Forms. Sections 25.8–25.11 demonstrate several techniques for manipulating the appearance of your GUIs. WPF also allows you to create data-driven GUIs that interact with many types of data. We demonstrate this in Section 25.12.

## 25.2 Windows Presentation Foundation (WPF)

Before WPF, you often had to use multiple technologies to build client apps. If a Windows Forms app required video and audio capabilities, you needed to incorporate an additional technology such as Windows Media Player. Likewise, if your app required 3D graphics ca-

pabilities, you had to incorporate a separate technology such as Direct3D. WPF provides a single platform capable of handling both of these requirements, and more. It enables you to use one technology to build apps containing GUI, images, animation, 2D or 3D graphics, audio and video capabilities. In this chapter and Chapter 33, we demonstrate each of these capabilities.

WPF can interoperate with existing technologies. For example, you can include WPF controls in Windows Forms apps to incorporate multimedia content (such as audio or video) without converting the entire app to WPF, which could be a costly and time-consuming process. You also can use Windows Forms controls in WPF apps.

WPF can use your computer's graphics hardware acceleration capabilities to increase your apps' performance. In addition, WPF generates **vector-based graphics** and is **resolution independent**. Vector-based graphics are defined not by a grid of pixels as **raster-based graphics** are, but rather by mathematical models. An advantage of vector-based graphics is that when you change the resolution, there's no loss of quality. Hence, the graphics become portable to a great variety of devices. Moreover, your apps won't appear smaller on higher-resolution screens. Instead, they'll remain the same size and display sharper. Chapter 33 presents more information about vector-based graphics and resolution independence.

Building a GUI with WPF is similar to building a GUI with Windows Forms—you drag-and-drop predefined controls from the **Toolbox** onto the design area. Many WPF controls correspond directly to those in Windows Forms. Just as in a Windows Forms app, the functionality is event driven. Many of the Windows Forms events you're familiar with are also in WPF. A WPF **Button**, for example, is similar to a Windows Forms **Button**, and both raise **Click** events.

There are several important differences between the two technologies, though. The WPF layout scheme is different. WPF properties and events have more capabilities. Most notably, WPF allows designers to define the appearance and content of a GUI without any C# code by defining it in XAML, a descriptive **markup** language (that is, a text-based notation for describing something).

### ***Introduction to XAML***

In Windows Forms, when you use the designer to create a GUI, the IDE generates code statements that create and configure the controls. In WPF, it generates XAML markup. Because XML is designed to be readable by both humans and computers, you also can manually write XAML markup to define GUI controls. When you compile your WPF app, a XAML compiler generates code to create and configure controls based on your XAML markup. This technique of defining *what* the GUI should contain without specifying *how* to generate it is an example of **declarative programming**.

XAML allows designers and programmers to work together more efficiently. Without writing any code, a graphic designer can edit the look-and-feel of an app using a design tool, such as Microsoft's **Blend for Visual Studio**—a XAML graphic design program that's installed with Visual Studio Community edition. A programmer can import the XAML markup into Visual Studio and focus on coding the logic that gives an app its functionality. Even if you're working alone, however, this separation of front-end appearance from back-end logic improves your program's organization and makes it easier to maintain. XAML is an essential component of WPF programming.

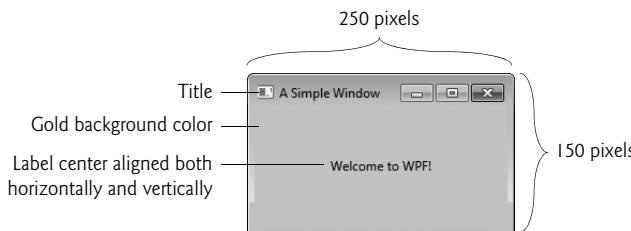
## 25.3 Declarative GUI Programming Using XAML

A XAML document defines the appearance of a WPF app. Figure 25.1 is a simple XAML document that defines a window that displays *Welcome to WPF!* A XAML document consists of many nested elements, delimited by start tags and end tags. As with any other XML document, each XAML document must contain a single root element. Just as in XML, data is placed as nested content or in attributes.

---

```

1  <!-- Fig. 25.1: MainWindow.xaml -->
2  <!-- A simple XAML document. -->
3
4  <!-- the Window control is the root element of the GUI -->
5  <Window x:Class="XAMLIntroduction.MainWindow"
6      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
7      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
8      Title="A Simple Window" Height="150" Width="250">
9
10 <!-- a Layout container -->
11 <Grid Background="Gold">
12     <!-- a Label control -->
13     <Label Content="Welcome to WPF!" HorizontalAlignment="Center"
14         VerticalAlignment="Center"/>
15 </Grid>
16 </Window>
```



**Fig. 25.1** | A simple XAML document.

### Presentation XAML Namespace and Standard XAML Namespace

Two standard namespaces must be defined in every XAML document so that the XAML compiler can interpret your markup—the **presentation XAML namespace**, which defines WPF-specific elements and attributes, and the **standard XAML namespace**, which defines elements and attributes that are standard to all types of XAML documents. Usually, the presentation XAML namespace (<http://schemas.microsoft.com/winfx/2006/xaml/presentation>) is defined as the default namespace (line 6), and the standard XAML namespace (<http://schemas.microsoft.com/winfx/2006/xaml>) is mapped to the namespace prefix x (line 7). These are both automatically included in the **Window** element’s start tag when you create a WPF app.

### Window Control

WPF **controls** are represented by XAML elements. The root element of the XAML document in Fig. 25.1 is a **Window** control (lines 5–16), which defines the app’s window—this

corresponds to a `Form` in Windows Forms. The `Window` start tag `x:Class` attribute (line 5) specifies the name of the associated *code-behind* class that provides the GUI's functionality. The `x:` signifies that the `Class` attribute is located in the standard XAML namespace. A XAML document must have an associated code-behind file to handle events.

Using attributes, you can define a control's properties in XAML. For example, the `Window`'s `Title`, `Width` and `Height` properties are set in line 8. A `Window`'s `Title` specifies the text that's displayed in the title bar. The `Width` and `Height` properties specify a control's width and height, respectively, using machine-independent pixels.

### *Content Controls*

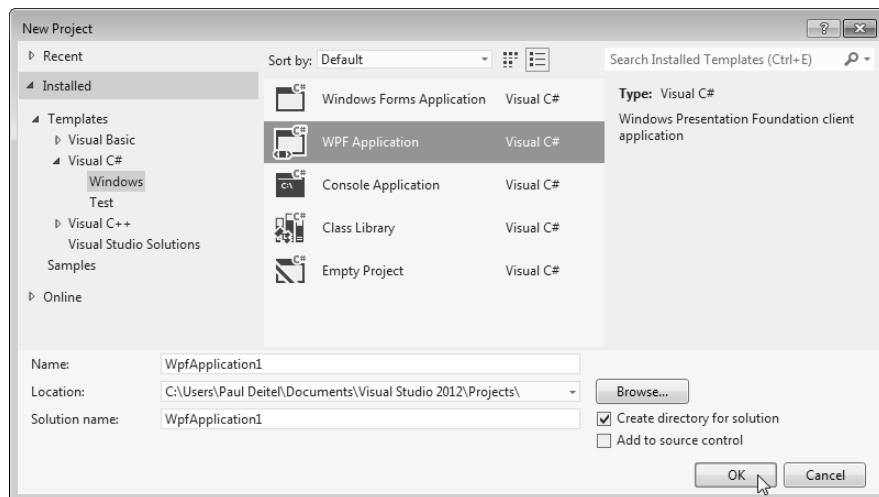
`Window` is a **content control** (a control derived from class `ContentControl`), meaning it can have exactly one child element or text content. You'll almost always set a **layout container** (a control derived from the `Panel` class) as the child element so that you can host multiple controls in a `Window`. A layout container such as a `Grid` (lines 11–15) can have many child elements, allowing it to contain many controls. In Section 25.5, you'll use content controls and layout containers to arrange a GUI.

### *Label Control*

Like `Window`, a `Label` (lines 13–14) is also a `ContentControl`. Labels are generally used to display text.

## 25.4 Creating a WPF App

To create a new WPF app, select `File > New > Project...` to display the `New Project` dialog (Fig. 25.2) and select **WPF Application** from the list of template types under **Visual C# > Windows**. Specify a name and location for your app, then click **OK** to create the project. The IDE for a WPF app looks nearly identical to that of a Windows Forms app. You'll recognize the familiar **Toolbox**, **Design view**, **Solution Explorer** and **Properties** window.



**Fig. 25.2 |** New Project dialog.

### XAML View<sup>1</sup>

There are differences in the IDE, however. One is the new **XAML view** (Fig. 25.3) that appears below the design area when you open a XAML document that represents a window. The **XAML** view is linked to the **Design** view and the **Properties** window. When you edit content in the **Design** view, the **XAML** view automatically updates, and vice versa. Likewise, when you edit properties in the **Properties** window, the **XAML** view automatically updates, and vice versa.



**Fig. 25.3** | XAML view.

### Generated Files

When you create a WPF app, several files are generated and can be viewed in the **Solution Explorer**. **App.xaml** defines the **Application** object and its settings. The most noteworthy setting is the **Application** element's **StartupUri** attribute, which defines the XAML document that executes first when the app loads (**MainWindow.xaml** by default). **App.xaml.cs** contains **App.xaml**'s code-behind class and handles application-level events. **MainWindow.xaml** defines the app's window, and **MainWindow.xaml.cs** contains its code-behind class, which handles the window's events. The file name of the code-behind class is always the file name of the associated XAML document followed by the **.cs** file-name extension.

### Setting XAML Indent Size and Displaying Line Numbers

We use three-space indents in our code. To ensure that your code appears the same as the book's examples, change the tab spacing for XAML documents to three spaces (the default is four). Select **Tools** > **Options...** to display the **Options** dialog, then in **Text Editor** > **XAML** > **Tabs** change the **Tab size** and **Indent size** to 3. You should also configure the **XAML** editor to display line numbers by checking the **Line numbers** checkbox in **Text Editor** > **XAML** > **General**.

### GUI Design

Creating a WPF app is similar to creating a Windows **Forms** app. You can drag-and-drop controls onto the **Design** view of your WPF GUI. A control's properties can be edited in the **Properties** window. Because XAML is easy to understand and edit, some programmers manually edit their GUIs' XAML markup directly rather than doing everything through the IDE's drag-and-drop GUI designer and **Properties** window.

1. Visual-Studio-generated XAML can vary between editions—elements may appear in a different order from what we show or with additional items that we do not discuss.

## 25.5 Laying Out Controls

In Windows Forms, a control's size and location are specified explicitly. In WPF, a control's size should be specified as a *range* of possible values rather than fixed values, and its location specified *relative* to those of other controls. This scheme, in which you specify how controls share the available space, is called **flow-based layout**. Its advantage is that it enables your GUIs, if designed properly, to be aesthetically pleasing, no matter how a user might *resize* the app. Likewise, it enables your GUIs to be resolution independent.

### 25.5.1 General Layout Principles

Layout refers to the *size* and *positioning* of controls. The WPF layout scheme addresses both of these in a flow-based fashion and can be summarized by two fundamental principles with regard to a control's size and position.

#### *Size of a Control*

Unless necessary, a control's size should *not* be defined *explicitly*. Doing so often creates a design that looks pleasing when it first loads, but deteriorates when the app is resized or the content updates. In addition to the `Width` and `Height` properties associated with every control, all WPF controls have the `MinWidth`, `MinHeight`, `MaxHeight` and `MaxWidth` properties. If the `Width` and `Height` properties are both `Auto` (which is the default when they are not specified in the XAML code), you can use `MinWidth`, `MinHeight`, `MaxWidth` and `MaxHeight` to specify a *range* of acceptable sizes for a control as it's resized with its container.

#### *Position of a Control*

A control's position should *not* be defined in absolute terms. Instead, it should be specified based on its position *relative* to the layout container in which it's included and the other controls in the same container. All controls have three properties for this purpose—`Margin`, `HorizontalAlignment` and `VerticalAlignment`. `Margin` specifies how much space to put around a control's edges. The value of `Margin` is a comma-separated list of four integers, representing the left, top, right and bottom margins. Additionally, you can specify two integers—the first represents the value for the left and right margins and the second for the top and bottom margins. If you specify just one integer, it uses the same margin on all four sides.

`HorizontalAlignment` and `VerticalAlignment` specify how to align a control within its layout container. Valid options of `HorizontalAlignment` are `Left`, `Center`, `Right` and `Stretch`. Valid options of `VerticalAlignment` are `Top`, `Center`, `Bottom` and `Stretch`. `Stretch` means that the object will occupy as much space as possible.

#### *Other Layout Properties*

A control can have other layout properties specific to the layout container in which it's contained. We'll discuss these as we examine the specific layout containers. WPF provides many controls for laying out a GUI. Figure 25.4 lists several of them.

Control	Description
<i>Layout containers (derived from Panel)</i>	
Grid	Layout is defined by a <i>grid of rows and columns</i> , depending on the RowDefinitions and ColumnDefinitions properties. Elements are placed into <i>cells</i> .
Canvas	Layout is <i>coordinate based</i> . Element positions are defined explicitly by their distance from the <i>top</i> and <i>left</i> edges of the Canvas.
StackPanel	Elements are arranged in a <i>single row or column</i> , depending on the Orientation property.
DockPanel	Elements are positioned based on which edge they're <i>docked</i> to. If the LastChildFill property is True, the last element gets the remaining space in the middle.
WrapPanel	A wrapping StackPanel. Elements are arranged <i>sequentially in rows or columns</i> (depending on the Orientation), each row or column wrapping to start a new one when it reaches the WrapPanel's right or bottom edge, respectively.
<i>Content controls (derived from ContentControl)</i>	
Border	Adds a background or a border to the child element.
GroupBox	Surrounds the child element with a titled box.
Window	The app's window. Also the root element.
Expander	Puts the child element in a titled area that collapses to display just the header and expands to display the header and the content.

**Fig. 25.4** | Common controls used for layout.

### 25.5.2 Layout in Action

Figure 25.5 shows the XAML document and the GUI display of a painter app. Note the use of Margin, HorizontalAlignment and VerticalAlignment throughout the markup. This example introduces several WPF controls that are commonly used for layout, as well as a few other basic controls like Buttons and RadioButtons.

---

```

1  <!-- Fig. 25.5: MainWindow.xaml -->
2  <!-- XAML of a painter app. -->
3  <Window x:Class="Painter.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Painter" Height="340" Width="350" Background="Beige">
7
8  <!-- creates a Grid -->
9  <Grid>
10     <Grid.ColumnDefinitions>
11         <ColumnDefinition Width="Auto" /> <!-- defines a column -->
12         <ColumnDefinition Width="*" />
13     </Grid.ColumnDefinitions>
```

---

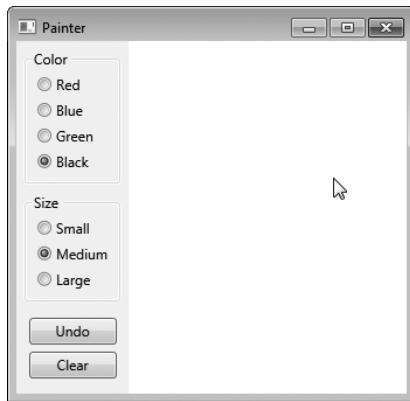
**Fig. 25.5** | XAML of a painter app. (Part 1 of 3.)

```

14      <!-- creates a Canvas -->
15      <Canvas x:Name="paintCanvas" Grid.Column="1" Background="White"
16          Margin="0" MouseMove="paintCanvas_MouseMove"
17          MouseLeftButtonDown="paintCanvas_MouseLeftButtonDown"
18          MouseLeftButtonUp="paintCanvas_MouseLeftButtonUp"
19          MouseRightButtonDown="paintCanvas_MouseRightButtonDown"
20          MouseRightButtonUp="paintCanvas_MouseRightButtonUp" />
21
22
23      <!-- creates a StackPanel-->
24      <StackPanel Margin="3">
25          <!-- creates a GroupBox for color options -->
26          <GroupBox Header="Color" Margin="3">
27              <StackPanel Margin="3" HorizontalAlignment="Left"
28                  VerticalAlignment="Top">
29
30                  <!-- creates RadioButtons for selecting color -->
31                  <RadioButton x:Name="redRadioButton" Content="Red"
32                      Margin="3" Checked="redRadioButton_Checked" />
33                  <RadioButton x:Name="blueRadioButton" Content="Blue"
34                      Margin="3" Checked="blueRadioButton_Checked" />
35                  <RadioButton x:Name="greenRadioButton" Content="Green"
36                      Margin="3" Checked="greenRadioButton_Checked" />
37                  <RadioButton x:Name="blackRadioButton" Content="Black"
38                      IsChecked="True" Margin="3"
39                      Checked="blackRadioButton_Checked" />
40              </StackPanel>
41          </GroupBox>
42
43          <!-- creates GroupBox for size options -->
44          <GroupBox Header="Size" Margin="3">
45              <StackPanel Margin="3" HorizontalAlignment="Left"
46                  VerticalAlignment="Top">
47                  <RadioButton x:Name="smallRadioButton" Content="Small"
48                      Margin="3" Checked="smallRadioButton_Checked" />
49                  <RadioButton x:Name="mediumRadioButton" IsChecked="True"
50                      Checked="mediumRadioButton_Checked" Content="Medium"
51                      Margin="3" />
52                  <RadioButton x:Name="largeRadioButton" Content="Large"
53                      Margin="3" Checked="largeRadioButton_Checked" />
54              </StackPanel>
55          </GroupBox>
56
57          <!-- creates a Button-->
58          <Button x:Name="undoButton" Content="Undo" Width="75"
59              Margin="3,10,3,3" Click="undoButton_Click"/>
60
61          <!-- creates a Button-->
62          <Button x:Name="clearButton" Content="Clear" Width="75"
63              Margin="3,10,3,3" Click="clearButton_Click"/>
64      </StackPanel>
65  </Grid>
66 </Window>

```

**Fig. 25.5** | XAML of a painter app. (Part 2 of 3.)



**Fig. 25.5** | XAML of a painter app. (Part 3 of 3.)

This app's controls look similar to Windows Forms controls. WPF **RadioButtons** function as *mutually exclusive* options, just like their Windows Forms counterparts. However, a WPF RadioButton does not have a **Text** property. Instead, it's a **ContentControl**, meaning it can have *exactly one child* or *text content*. This makes the control more versatile, enabling it to be labeled by an image or other item. In this example, each RadioButton is labeled by plain text specified with the **Content** attribute (for example, lines 31, 33, 35 and 37). A WPF **Button** behaves like a Windows Forms Button but is a **ContentControl**. As such, a WPF Button can display any single element as its content, not just text. Lines 58–59 and 62–63 define the two buttons seen in the Painter app. You can drag and drop controls onto the WPF designer and create their event handlers, just as you do in the Windows Forms designer.

### **GroupBox Control**

A WPF **GroupBox** arranges controls and displays just as a Windows Forms GroupBox would, but using one is slightly different. The **Header** property replaces the Windows Forms version's **Text** property. Also, a **GroupBox** is a **ContentControl**, so to place *multiple* controls in it, you must place them in a layout container such as a **StackPanel** (lines 27–40).

### **StackPanel Control**

In the Painter app, we organized each **GroupBox**'s **RadioButtons** by placing them in **StackPanel**s (for example, lines 27–40). A **StackPanel** is the *simplest* of layout containers. It arranges its content either *vertically* or *horizontally*, depending on its **Orientation** property's setting. The default **Orientation** is **Vertical**, which is used by every **StackPanel** in the Painter example.

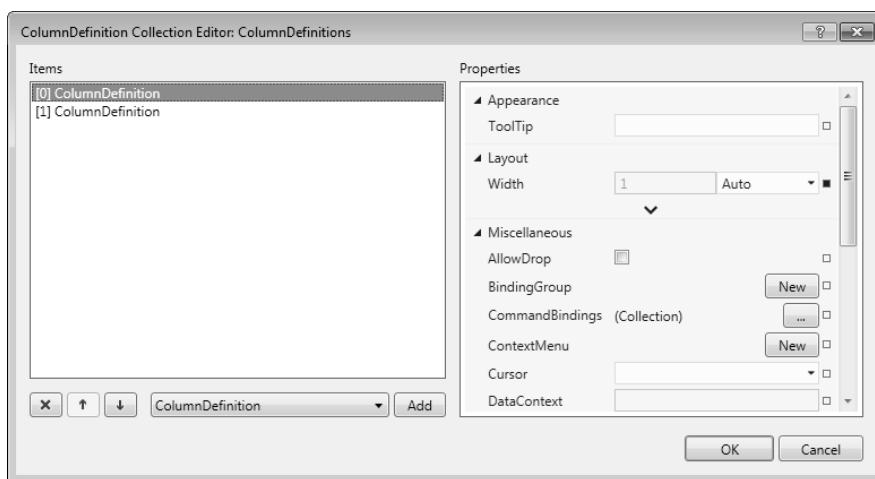
### **Grid Control**

The Painter Window's contents are contained within a **Grid**—a flexible, all-purpose layout container. A **Grid** organizes controls into a user-defined number of *rows* and *columns* (one row and one column by default). You can define a **Grid**'s rows and columns by setting its **RowDefinitions** and **ColumnDefinitions** properties, whose values are a collection of **RowDefinition** and **ColumnDefinition** objects, respectively. Because these properties do not

take `string` values, they cannot be specified as attributes in the `Grid` tag. Another syntax is used instead. A class's property can be defined in XAML as a *nested* element with the name `ClassName.PropertyName`. For example, the `Grid.ColumnDefinitions` element in lines 10–13 sets the `Grid`'s `ColumnDefinitions` property and defines two columns, which separate the options from the painting area, as shown in Fig. 25.5.

You can specify the `Width` of a `ColumnDefinition` and the `Height` of a `RowDefinition` with an *explicit size*, a *relative size* (using `*`) or `Auto`. `Auto` makes the row or column only as big as it needs to be to fit its contents. The setting `*` specifies the size of a row or column with respect to the `Grid`'s other rows and columns. For example, a column with a `Height` of `2*` would be twice the size of a column that's `1*` (or just `*`). A `Grid` first allocates its space to the rows and columns whose sizes are defined explicitly or determined automatically. The remaining space is divided among the other rows and columns. By default, all `Widths` and `Heights` are set to `*`, so every cell in the grid is of equal size. In the `Painter` app, the first column is just wide enough to fit the controls, and the rest of the space is allotted to the painting area (lines 11–12). If you resize the `Painter` window, you'll notice that only the width of the paintable area increases or decreases.

If you click the ellipsis button next to the `RowDefinitions` or `ColumnDefinitions` property in the **Properties** window, the **Collection Editor** window will appear. (If you cannot find a property, type its name in the **Search Properties** text box at the top of the **Properties** window or view the properties by **Name** rather than **Category**.) This tool can be used to add, remove, reorder, and edit the properties of rows and columns in a `Grid`. In fact, any property that takes a collection as a value can be edited in a version of the **Collection Editor** specific to that collection. For example, you could edit the `Items` property of a `ComboBox` (that is, drop-down list) in such a way. The `ColumnDefinitions` **Collection Editor** is shown in Fig. 25.6.



**Fig. 25.6 |** Using the **Collection Editor**.

The control properties we've introduced so far look and function just like their Windows Forms counterparts. To indicate which cell of a `Grid` a control belongs in, however, you

use the **Grid.Row** and **Grid.Column** properties. These are known as **attached properties**—they’re defined by a *different* control than that to which they’re applied. In this case, Row and Column are defined by the Grid itself but applied to the controls contained in the Grid (for example, line 16 in Fig. 25.5). To specify the number of rows or columns that a control spans, you can use the **Grid.RowSpan** or **Grid.ColumnSpan** attached properties, respectively. By default, a control spans the entire Grid, unless the **Grid.Row** or **Grid.Column** property is set, in which case the control spans only the specified row or column by default.

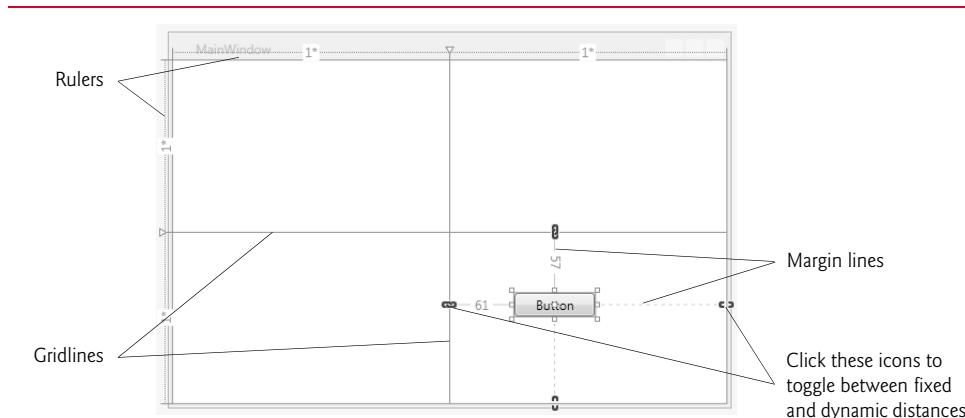
### **Canvas Control**

The painting area of the Painter app is a **Canvas** (lines 16–21), another layout container. A Canvas allows users to position controls by defining explicit coordinates. Controls in a Canvas have the attached properties, **Canvas.Left** and **Canvas.Top**, which specify the control’s coordinate position based on its distance from the Canvas’s *left* and *top* borders, respectively. If two controls *overlap*, the one with the *greater* **Canvas.ZIndex** displays in the *foreground*. If this property is not defined for the controls, then the *last* control added to the canvas displays in the foreground. When you provide a name for a control via the Properties window, the IDE adds an **x:Name** attribute to the controls XAML. This name is used in the C# code as the control’s variable name.

### **Layout in Design Mode**

As you’re creating your GUI in **Design** mode, you’ll notice many helpful layout features. For example, as you *resize* a control, its width and height are displayed. In addition, *snap-lines* appear as necessary to help you align the edges of elements. These lines will also appear when you move controls around the design area.

When you select a control, *margin lines* that extend from the control to the edges of its container appear, as shown in Fig. 25.7. If a solid line containing a number extends to the edge of the container, then the distance between the control and that edge is *fixed*. If a dashed line appears between the edge of the control and the edge of the container, then the distance between the control and that edge of the container is *dynamic*—the distance changes as the container size changes. You can toggle between the two by clicking the icons at the ends of the lines.



**Fig. 25.7** | Margin lines and gridlines in **Design** view.

Furthermore, the **Design** view also helps you use a **Grid**. As shown in Fig. 25.7, when you select a control in a **Grid**, the **Grid**'s rulers appear to the *left* and on *top* of it. The widths and heights of each column and row, respectively, appear on the rulers. Gridlines that outline the **Grid**'s rows and columns also appear, helping you align and position the **Grid**'s elements. You also can create more rows and columns by clicking where you want to separate them on the ruler.

## 25.6 Event Handling

Basic event handling in WPF is almost identical to Windows Forms event handling, but there is a fundamental difference, which we'll explain later in this section. We'll use the **Painter** example to introduce WPF event handling. Figure 25.8 provides the code-behind class for the **Painter** window. As in Windows Forms GUIs, when you double click a control, the IDE automatically generates an event handler for that control's primary event. The IDE also adds an attribute to the control's XAML element specifying the event name and the name of the event handler that responds to the event. For example, in line 32 of Fig. 25.5, the attribute

```
Checked="redRadioButton_Checked"
```

specifies that the **redRadioButton**'s Checked event handler is **redRadioButton\_Checked**.

---

```

1 // Fig. 25.8: MainWindow.xaml.cs
2 // Code-behind for MainWindow.xaml.
3 using System.Windows;
4 using System.Windows.Controls;
5 using System.Windows.Input;
6 using System.Windows.Media;
7 using System.Windows.Shapes;
8
9 namespace Painter
10 {
11     public partial class MainWindow : Window
12     {
13         private int diameter = (int) Sizes.MEDIUM; // diameter of circle
14         private Brush brushColor = Brushes.Black; // drawing color
15         private bool shouldErase = false; // specify whether to erase
16         private bool shouldPaint = false; // specify whether to paint
17
18         private enum Sizes // size constants for diameter of the circle
19         {
20             SMALL = 4,
21             MEDIUM = 8,
22             LARGE = 10
23         } // end enum Sizes
24
25         // constructor
26         public MainWindow()
27     {

```

---

**Fig. 25.8** | Code-behind for **MainWindow.xaml**. (Part 1 of 4.)

```
28         InitializeComponent();
29     } // end constructor
30
31     // paints a circle on the Canvas
32     private void PaintCircle( Brush circleColor, Point position )
33     {
34         Ellipse newEllipse = new Ellipse(); // create an Ellipse
35
36         newEllipse.Fill = circleColor; // set Ellipse's color
37         newEllipse.Width = diameter; // set its horizontal diameter
38         newEllipse.Height = diameter; // set its vertical diameter
39
40         // set the Ellipse's position
41         Canvas.SetTop( newEllipse, position.Y );
42         Canvas.SetLeft( newEllipse, position.X );
43
44         paintCanvas.Children.Add( newEllipse );
45     } // end method PaintCircle
46
47     // handles paintCanvas's MouseLeftButtonDown event
48     private void paintCanvas_MouseLeftButtonDown( object sender,
49             MouseButtonEventArgs e )
50     {
51         shouldPaint = true; // OK to draw on the Canvas
52     } // end method paintCanvas_MouseLeftButtonDown
53
54     // handles paintCanvas's MouseLeftButtonUp event
55     private void paintCanvas_MouseLeftButtonUp( object sender,
56             MouseButtonEventArgs e )
57     {
58         shouldPaint = false; // do not draw on the Canvas
59     } // end method paintCanvas_MouseLeftButtonUp
60
61     // handles paintCanvas's MouseRightButtonDown event
62     private void paintCanvas_MouseRightButtonDown( object sender,
63             MouseButtonEventArgs e )
64     {
65         shouldErase = true; // OK to erase the Canvas
66     } // end method paintCanvas_MouseRightButtonDown
67
68     // handles paintCanvas's MouseRightButtonUp event
69     private void paintCanvas_MouseRightButtonUp( object sender,
70             MouseButtonEventArgs e )
71     {
72         shouldErase = false; // do not erase the Canvas
73     } // end method paintCanvas_MouseRightButtonUp
74
75     // handles paintCanvas'sMouseMove event
76     private void paintCanvas_MouseMove( object sender,
77             MouseEventArgs e )
78     {
79         if ( shouldPaint )
80         {
```

---

**Fig. 25.8** | Code-behind for MainWindow.xaml. (Part 2 of 4.)

```
81         // draw a circle of selected color at current mouse position
82         Point.mousePosition = e.GetPosition( paintCanvas );
83         PaintCircle( brushColor,.mousePosition );
84     } // end if
85     else if ( shouldErase )
86     {
87         // erase by drawing circles of the Canvas's background color
88         Point.mousePosition = e.GetPosition( paintCanvas );
89         PaintCircle( paintCanvas.Background,.mousePosition );
90     } // end else if
91 } // end method paintCanvas_MouseMove
92
93 // handles Red RadioButton's Checked event
94 private void redRadioButton_Checked( object sender,
95                                     RoutedEventArgs e )
96 {
97     brushColor = Brushes.Red;
98 } // end method redRadioButton_Checked
99
100 // handles Blue RadioButton's Checked event
101 private void blueRadioButton_Checked( object sender,
102                                     RoutedEventArgs e )
103 {
104     brushColor = Brushes.Blue;
105 } // end method blueRadioButton_Checked
106
107 // handles Green RadioButton's Checked event
108 private void greenRadioButton_Checked( object sender,
109                                     RoutedEventArgs e )
110 {
111     brushColor = Brushes.Green;
112 } // end method greenRadioButton_Checked
113
114 // handles Black RadioButton's Checked event
115 private void blackRadioButton_Checked( object sender,
116                                     RoutedEventArgs e )
117 {
118     brushColor = Brushes.Black;
119 } // end method blackRadioButton_Checked
120
121 // handles Small RadioButton's Checked event
122 private void smallRadioButton_Checked( object sender,
123                                     RoutedEventArgs e )
124 {
125     diameter = ( int ) Sizes.SMALL;
126 } // end method smallRadioButton_Checked
127
128 // handles Medium RadioButton's Checked event
129 private void mediumRadioButton_Checked( object sender,
130                                     RoutedEventArgs e )
131 {
132     diameter = ( int ) Sizes.MEDIUM;
133 } // end method mediumRadioButton_Checked
```

Fig. 25.8 | Code-behind for MainWindow.xaml. (Part 3 of 4.)

```
134      // handles Large RadioButton's Checked event
135      private void largeRadioButton_Checked( object sender,
136          RoutedEventArgs e )
137      {
138          diameter = ( int ) Sizes.LARGE;
139      } // end method largeRadioButton_Checked
140
141      // handles Undo Button's Click event
142      private void undoButton_Click( object sender, RoutedEventArgs e )
143      {
144          int count = paintCanvas.Children.Count;
145
146          // if there are any shapes on Canvas remove the last one added
147          if ( count > 0 )
148              paintCanvas.Children.RemoveAt( count - 1 );
149      } // end method undoButton_Click
150
151      // handles Clear Button's Click event
152      private void clearButton_Click( object sender, RoutedEventArgs e )
153      {
154          paintCanvas.Children.Clear(); // clear the canvas
155      } // end method clearButton_Click
156  } // end class MainWindow
157 } // end namespace Painter
```



---

Fig. 25.8 | Code-behind for MainWindow.xaml. (Part 4 of 4.)

The Painter app “draws” by placing colored circles on the Canvas at the mouse pointer’s position as you drag the mouse. The PaintCircle method (lines 32–45 in Fig. 25.8) creates the circle by defining an Ellipse object (lines 34–38), and positions it using the **Canvas.SetTop** and **Canvas.SetLeft** methods (lines 41–42), which change the circle’s **Canvas.Left** and **Canvas.Top** attached properties, respectively.

The **Children** property stores a list (of type **UIElementCollection**) of a layout container’s child elements. This allows you to edit the layout container’s child elements with C# code as you would any other implementation of the **IEnumerable** interface. You can

add an element to the container by calling the **Add** method of the **Children** list (for example, line 44). The **Undo** and **Clear** buttons work by invoking the **RemoveAt** and **Clear** methods of the **Children** list (lines 149 and 155), respectively.

Just as with a Windows Forms **RadioButton**, a WPF **RadioButton** has a **Checked** event. Lines 94–140 handle the **Checked** event for each of the **RadioButtons** in this example, which change the color and the size of the circles painted on the **Canvas**. The **Button** control's **Click** event also functions the same in WPF as it did in Windows Forms. Lines 143–156 handle the **Undo** and **Clear** Buttons. The event-handler declarations look almost identical to how they would look in a Windows Forms app, except that the event-arguments object (**e**) is a **RoutedEventArgs** object instead of an **EventArgs** object. We'll explain why later in this section.

### *Mouse and Keyboard Events*

WPF has built-in support for keyboard and mouse events that's nearly identical to the support in Windows Forms. Painter uses the **MouseMove** event of the paintable **Canvas** to paint and erase (lines 76–91). A control's **MouseMove** event is triggered whenever the mouse moves within the boundaries of the control. Information for the event is passed to the event handler using a **MouseEventArgs** object, which contains mouse-specific information. The **GetPosition** method of **MouseEventArgs**, for example, returns the current position of the mouse relative to the control that triggered the event (for example, lines 82 and 88). **MouseMove** works the same as it does in Windows Forms. [Note: Much of the functionality in our sample Painter app is already provided by the WPF **InkCanvas** control. We chose not to use this control so we could demonstrate various other WPF features.]

WPF has additional mouse events. Painter also uses the **MouseLeftButtonDown** and **MouseLeftButtonUp** events to toggle painting on and off (lines 48–59), and the **MouseRightButtonDown** and **MouseRightButtonUp** events to toggle erasing on and off (lines 62–73). All of these events pass information to the event handler using the **MouseButtonEventArgs** object, which has properties specific to a mouse button (for example, **ButtonState** or **ClickCount**) in addition to mouse-specific ones. These events are new to WPF and are more specific versions of **MouseUp** and **MouseDown** (which are still available in WPF). A summary of commonly used mouse and keyboard events is provided in Fig. 25.9.

### Common mouse and keyboard events

#### *Mouse Event with an Event Argument of Type **MouseEventArgs***

<b>MouseMove</b>	Raised when you move the mouse within a control's boundaries.
------------------	---

#### *Mouse Events with an Event Argument of Type **MouseButtonEventArgs***

<b>MouseLeftButtonDown</b>	Raised when the left mouse button is pressed.
<b>MouseLeftButtonUp</b>	Raised when the left mouse button is released.
<b>MouseRightButtonDown</b>	Raised when the right mouse button is pressed.
<b>MouseRightButtonUp</b>	Raised when the right mouse button is released.

#### *Mouse Event with an Event Argument of Type **MouseWheelEventArgs***

<b>MouseWheel</b>	Raised when the mouse wheel is rotated.
-------------------	---

**Fig. 25.9** | Common mouse and keyboard events. (Part 1 of 2.)

### Common mouse and keyboard events

#### *Keyboard Events with an Event Argument of Type KeyEventArgs*

KeyDown	Raised when a key is pressed.
KeyUp	Raised when a key is released.

**Fig. 25.9** | Common mouse and keyboard events. (Part 2 of 2.)

### *Routed Events*

WPF events have a significant distinction from their Windows Forms counterparts—they can travel either up (from child to parent) or down (from parent to child) the containment hierarchy—the hierarchy of nested elements defined within a control. This is called **event routing**, and all WPF events are **routed events**.

The event-arguments object that's passed to the event handler of a WPF Button's Click event or a RadioButton's Check event is of the type **RoutedEventArgs**. All event-argument objects in WPF are of type **RoutedEventArgs** or one of its subclasses. As an event travels up or down the hierarchy, it may be useful to stop it before it reaches the end. When the **Handled** property of the **RoutedEventArgs** parameter is set to **true**, event handlers ignore the event. It may also be useful to know the source where the event was first triggered. The **Source** property stores this information. You can learn more about the benefits of routed events at [bit.ly/RoutedEventArgs](http://bit.ly/RoutedEventArgs).

### *Demonstrating Routed Events*

Figures 25.10 and 25.11 show the XAML and code-behind for a program that demonstrates event routing. The program contains two GroupBoxes, each with a Label inside (lines 15–27 in Fig. 25.10). One group handles a left-mouse-button press with MouseLeftButtonUp, and the other with PreviewMouseLeftButtonUp. As the event travels up or down the containment hierarchy, a log of where the event has traveled is displayed in a TextBox (line 29). The WPF TextBox functions just like its Windows Forms counterpart.

---

```

1  <!-- Fig. 25.10: MainWindow.xaml -->
2  <!-- Routed-events example (XAML). -->
3  <Window x:Class="RoutedEvents.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Routed Events" Height="300" Width="300"
7      x:Name="routedEventsWindow">
8      <Grid>
9          <Grid.RowDefinitions>
10             <RowDefinition Height="Auto" />
11             <RowDefinition Height="Auto" />
12             <RowDefinition Height="*" />
13         </Grid.RowDefinitions>
14
15         <GroupBox x:Name="tunnelingGroupBox" Grid.Row="0" Header="Tunneling"
16             Margin="5" PreviewMouseLeftButtonUp="Tunneling">
```

---

**Fig. 25.10** | Routed-events example (XAML). (Part 1 of 2.)

---

```

17      <Label x:Name="tunnelingLabel" Margin="5"
18          HorizontalAlignment="Center"
19          PreviewMouseLeftButtonUp="Tunneling" Content="Click Here"/>
20    </GroupBox>
21
22    <GroupBox x:Name="bubblingGroupBox" Grid.Row="1" Header="Bubbling"
23        Margin="5" MouseLeftButtonUp="Bubbling">
24      <Label x:Name="bubblingLabel" Margin="5"
25          MouseLeftButtonUp="Bubbling" HorizontalAlignment="Center"
26          Content="Click Here"/>
27    </GroupBox>
28
29    <TextBox x:Name="logTextBox" Grid.Row="2" Margin="5" />
30  </Grid>
31 </Window>

```

---

**Fig. 25.10** | Routed-events example (XAML). (Part 2 of 2.)

---

```

1 // Fig. 25.11: MainWindow.xaml.cs
2 // Routed-events example (code-behind).
3 using System.Windows;
4 using System.Windows.Controls;
5 using System.Windows.Input;
6
7 namespace RoutedEvents
8 {
9   public partial class MainWindow : Window
10  {
11    int bubblingEventStep = 1; // step counter for Bubbling
12    int tunnelingEventStep = 1; // step counter for Tunneling
13    string tunnelingLogText = string.Empty; // temporary Tunneling log
14
15    public RoutedEventsWindow()
16    {
17      InitializeComponent();
18    } // end constructor
19
20    // PreviewMouseUp is a tunneling event
21    private void Tunneling( object sender, MouseButtonEventArgs e )
22    {
23      // append step number and sender
24      tunnelingLogText = string.Format( "{0}({1}): {2}\n",
25          tunnelingLogText, tunnelingEventStep,
26          ( ( Control ) sender ).Name );
27      ++tunnelingEventStep; // increment counter
28
29      // execution goes from parent to child, ending with the source
30      if ( e.Source.Equals( sender ) )
31      {
32        tunnelingLogText = string.Format(
33          "This is a tunneling event:\n{0}", tunnelingLogText );

```

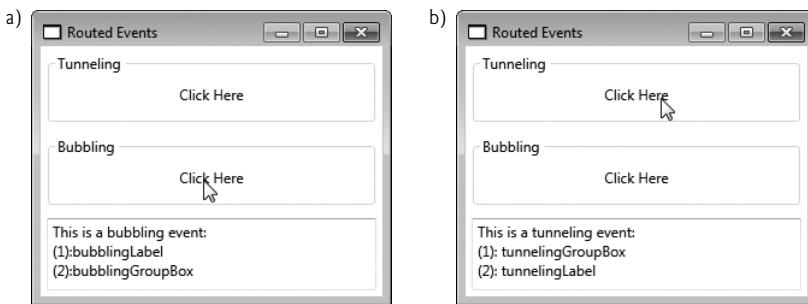
---

**Fig. 25.11** | Routed-events example (code-behind). (Part 1 of 2.)

```

34         logTextBox.Text = tunnelingLogText; // set logTextBox text
35         tunnelingLogText = string.Empty; // clear temporary log
36         tunnelingEventStep = 1; // reset counter
37     } // end if
38 } // end method Tunneling
39
40 // MouseUp is a bubbling event
41 private void Bubbling( object sender, MouseButtonEventArgs e )
42 {
43     // execution goes from child to parent, starting at the source
44     if ( e.Source.Equals( sender ) )
45     {
46         logTextBox.Clear(); // clear the logTextBox
47         bubblingEventStep = 1; // reset counter
48         logTextBox.Text = "This is a bubbling event:\n";
49     } // end if
50
51     // append step number and sender
52     logTextBox.Text = string.Format( "{0}({1}): {2}\n",
53         logTextBox.Text, bubblingEventStep,
54         ( ( Control ) sender ).Name );
55     ++bubblingEventStep;
56 } // end method Bubbling
57 } // end class MainWindow
58 } // end namespace RoutedEvents

```



**Fig. 25.11** | Routed-events example (code-behind). (Part 2 of 2.)

There are three types of routed events—**direct events**, **bubbling events** and **tunneling events**. *Direct events* are like ordinary Windows Forms events—they do *not* travel up or down the containment hierarchy. Bubbling events start at the Source and travel *up* the hierarchy ending at the root (Window) or until you set Handled to true. Tunneling events start at the top and travel *down* the hierarchy until they reach the Source or Handled is true. To help you distinguish tunneling events from bubbling events, WPF prefixes the names of tunneling events with Preview. For example, **PreviewMouseLeftButtonDown** is the tunneling version of **MouseLeftButtonDown**, which is a bubbling event.

If you click the **Click Here** Label in the **Tunneling** GroupBox, the click is handled first by the GroupBox, then by the contained Label. The event handler that responds to the click handles the **PreviewMouseLeftButtonUp** event—a tunneling event. The **Tunneling** method (lines 21–38 in Fig. 25.11) handles the events of both the GroupBox and the

Label. An event handler can handle events for many controls. Simply select each control then use the events tab in the **Properties** window to select the appropriate event handler for the corresponding event of each control. If you click the other Label, the click is handled first by the Label, then by the containing GroupBox. The **Bubbling** method (lines 41–56) handles the **MouseLeftButtonUp** events of both controls.

## 25.7 Commands and Common Application Tasks

In Windows Forms, event handling is the only way to respond to user actions. WPF provides an alternate technique called a **command**—an action or a task that may be triggered by many different user interactions. In Visual Studio, for example, you can cut, copy and paste code. You can execute these tasks through the **Edit** menu, a toolbar or keyboard shortcuts. To program this functionality in WPF, you can define a single command for each task, thus centralizing the handling of common tasks—this is not easily done in Windows Forms.

Commands also enable you to synchronize a task's availability to the state of its corresponding controls. For example, users should be able to copy something only if they have content selected. When you define the copy command, you can specify this as a requirement. As a result, if the user has no content selected, then the menu item, toolbar item and keyboard shortcut for copying are all automatically disabled.

Commands are implementations of the **ICommand** interface. When a command is executed, the **Execute** method is called. However, the command's execution logic is not defined in its Execute method. You must specify this logic when implementing the command. An **ICommand**'s **CanExecute** method works the same way. The logic that specifies when a command is enabled and disabled is not determined by the **CanExecute** method and must instead be specified by responding to an appropriate event. Class **RoutedCommand** is the standard implementation of **ICommand**. Every **RoutedCommand** has a **Name** and a collection of **InputGestures** (that is, keyboard shortcuts) associated with it. **RoutedUICommand** is an extension of **RoutedCommand** with a **Text** property, which specifies the default text to display on a GUI element that triggers the command.

WPF provides a command library of built-in commands. These commands have their standard keyboard shortcuts already associated with them. For example, **Copy** is a built-in command and has **Ctrl-C** associated with it. Figure 25.12 provides a list of some common built-in commands, separated by the classes in which they're defined.

### Common built-in commands from the WPF command library

#### *ApplicationCommands properties*

New	Open	Save	Close
Cut	Copy	Paste	

#### *EditingCommands properties*

ToggleBold	ToggleItalic	ToggleUnderline
------------	--------------	-----------------

#### *MediaCommands properties*

Play	Stop	Rewind	FastForward
IncreaseVolume	DecreaseVolume	NextTrack	PreviousTrack

**Fig. 25.12** | Common built-in commands from the WPF command library.

Figures 25.13 and 25.14 are the XAML markup and C# code for a simple text-editor app that allows users to format text into bold and italics, and also to cut, copy and paste text. The example uses the **RichTextBox** control (line 49), which allows users to enter, edit and format text. We use this app to demonstrate several built-in commands from the command library.

```

1  <!-- Fig. 25.13: MainWindow.xaml -->
2  <!-- Creating menus and toolbars, and using commands (XAML). -->
3  <Window x:Class="TextEditor.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Text Editor" Height="300" Width="300">
7
8      <Window.CommandBindings> <!-- define command bindings -->
9          <!-- bind the Close command to handler -->
10         <CommandBinding Command="Close" Executed="closeCommand_Executed" />
11     </Window.CommandBindings>
12
13     <Grid> <!-- define the GUI -->
14         <Grid.RowDefinitions>
15             <RowDefinition Height="Auto" />
16             <RowDefinition Height="Auto" />
17             <RowDefinition Height="*" />
18         </Grid.RowDefinitions>
19
20         <Menu Grid.Row="0"> <!-- create the menu -->
21             <!-- map each menu item to corresponding command -->
22             <MenuItem Header="File">
23                 <MenuItem Header="Exit" Command="Close" />
24             </MenuItem>
25             <MenuItem Header="Edit">
26                 <MenuItem Header="Cut" Command="Cut" />
27                 <MenuItem Header="Copy" Command="Copy" />
28                 <MenuItem Header="Paste" Command="Paste" />
29                 <Separator /> <!-- separates groups of menu items -->
30                 <MenuItem Header="Bold" Command="ToggleBold"
31                     FontWeight="Bold" />
32                 <MenuItem Header="Italic" Command="ToggleItalic"
33                     FontStyle="Italic" />
34             </MenuItem>
35         </Menu>
36
37         <ToolBar Grid.Row="1"> <!-- create the toolbar -->
38             <!-- map each toolbar item to corresponding command -->
39             <Button Command="Cut">Cut</Button>
40             <Button Command="Copy">Copy</Button>
41             <Button Command="Paste">Paste</Button>
42             <Separator /> <!-- separates groups of toolbar items -->
43             <Button FontWeight="Bold" Command="ToggleBold">Bold</Button>
44             <Button FontStyle="Italic" Command="ToggleItalic">
45                 Italic</Button>
46         </ToolBar>
```

**Fig. 25.13** | Creating menus and toolbars, and using commands (XAML). (Part I of 2.)

---

```

47      <!-- display editable, formattable text -->
48      <RichTextBox Grid.Row="2" Margin="5" />
49
50  </Grid>
51 </Window>

```

**Fig. 25.13** | Creating menus and toolbars, and using commands (XAML). (Part 2 of 2.)

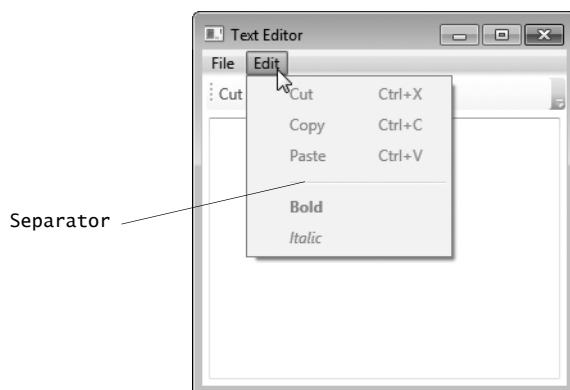
---

```

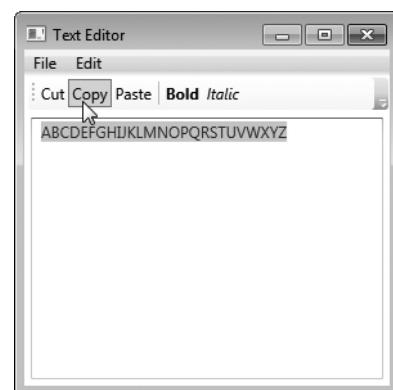
1 // Fig. 25.14: MainWindow.xaml.cs
2 // Code-behind class for a simple text editor.
3 using System.Windows;
4 using System.Windows.Input;
5
6 namespace TextEditor
7 {
8     public partial class MainWindow : Window
9     {
10         public MainWindow()
11         {
12             InitializeComponent();
13         } // end constructor
14
15         // exit the app
16         private void closeCommand_Executed( object sender,
17             ExecutedRoutedEventArgs e )
18         {
19             Application.Current.Shutdown();
20         } // end method closeCommand_Executed
21     } // end class MainWindow
22 } // end namespace TextEditor

```

a) When the app loads



b) After selecting some text



**Fig. 25.14** | Code-behind class for a simple text editor. (Part 1 of 2.)

c) After copying some text

**Fig. 25.14** | Code-behind class for a simple text editor. (Part 2 of 2.)

A command is executed when it's triggered by a *command source*. For example, the `Close` command is triggered by a `MenuItem` (line 23 in Fig. 25.13). The `Cut` command has two sources, a `MenuItem` and a `ToolBar Button` (lines 26 and 39, respectively). A command can have many sources.

To make use of a command, you must create a **command binding**—a link between a command and the methods containing its logic. You can declare a command binding by creating a `CommandBinding` object in XAML and setting its `Command` property to the name of the associated command (line 10). A command binding raises the `Executed` and `PreviewExecuted` events (*bubbling* and *tunneling* versions of the same event) when its associated command is executed. You program the command's functionality into an event handler for one of these events. In line 10, we set the `Executed` attribute to a method name, telling the program that the specified method (`closeCommand_Executed`) handles the command binding's `Executed` event.

In this example, we demonstrate the use of a *command binding* by implementing the `Close` command. When it executes, it shuts down the app. The method that executes this task is `Application.Current.Shutdown`, as shown in line 19 of Fig. 25.14.

You also can use a command binding to specify the logic for determining when a command should be *enabled* or *disabled*. You can do so by handling either the `CanExecute` or `PreviewCanExecute` (*bubbling* and *tunneling* versions of the same events) events in the same way that you handle the `Executed` or `PreviewExecuted` events. Because we do not define such a handler for the `Close` command in its command binding, it's always enabled. Command bindings should be defined within the `Window.CommandBindings` element (for example, lines 8–11 in Fig. 25.13).

The only time a command binding is *not* necessary is when a control has built-in functionality for dealing with a command. A `Button` or `MenuItem` linked to the `Cut`, `Copy`, or `Paste` commands is an example (for example, lines 26–28 and lines 39–41). As Fig. 25.14(a) shows, all three commands are disabled when the app loads. If you select some text, the `Cut` and `Copy` commands are enabled, as shown in Fig. 25.14(b). Once you have copied some text, the `Paste` command is enabled, as evidenced by Fig. 25.14(c). We did not have to define any associated command bindings or event handlers to implement

these commands. The `ToggleBold` and `ToggleItalic` commands are also implemented without any command bindings.

### Menus and Toolbars

The text editor uses menus and toolbars. The `Menu` control creates a menu containing `MenuItem`s. `MenuItem`s can be top-level menus such as `File` or `Edit` (lines 22 and 25 in Fig. 25.13), submenus, or items in a menu, which function like `Buttons` (for example, lines 26–28). If a `MenuItem` has nested `MenuItem`s, then it's a top-level menu or a submenu. Otherwise, it's an item that executes an action via either an event or a command. `MenuItem`s are content controls and thus can display any single GUI element as content.

A `ToolBar` is a single row or column (depending on the `Orientation` property) of options. A `ToolBar`'s `Orientation` is a *read-only* property that gets its value from the parent `ToolBarTray`, which can host multiple `ToolBar`s. If a `ToolBar` has no parent `ToolBarTray`, as is the case in this example, its `Orientation` is `Horizontal` by default. Unlike elements in a `Menu`, a `ToolBar`'s child elements are not of a specific type. A `ToolBar` usually contains `Buttons`, `Checkboxes`, `ComboBoxes`, `RadioButtons` and `Separators`, but any WPF control can be used. `ToolBar`s overwrite the look-and-feel of their child elements with their own specifications, so that the controls look seamless together. You can override the default specifications to create your own look-and-feel. Lines 37–46 define the text editor's `ToolBar`.

Menus and `ToolBar`s can incorporate `Separtors` (for example, lines 29 and 42) that differentiate groups of `MenuItem`s or controls. In a `Menu`, a `Separator` displays as a horizontal bar—as shown between the `Paste` and `Bold` menu options in Fig. 25.14(a). In a horizontal `ToolBar`, it displays as a short vertical bar—as shown in Fig. 25.14(b). You can use `Separtors` in any type of control that can contain multiple child elements, such as a `StackPanel`.

## 25.8 WPF GUI Customization

One advantage of WPF over Windows Forms is the ability to customize controls. WPF provides several techniques to customize the look and behavior of controls. The simplest takes full advantage of a control's properties. The value of a control's `Background` property, for example, is a *brush* (i.e., `Brush` object). This allows you to create a *gradient* or an *image* and use it as the background rather than a solid color. For more information about brushes, see Section 26.5. In addition, many controls that allowed only text content in Windows Forms are `ContentControls` in WPF, which can host any type of content—including other controls. The caption of a WPF `Button`, for example, could be an *image* or even a *video*.

In Section 25.9, we demonstrate how to use styles in WPF to achieve a uniform look-and-feel. In Windows Forms, if you want to make all your `Buttons` look the same, you have to manually set properties for every `Button`, or copy and paste. To achieve the same result in WPF, you can define the properties once as a style and apply the style to each `Button`. This is similar to the CSS/HTML implementation of styles. HTML specifies the content and structure of a website, and CSS defines styles that specify the presentation of elements in a website. For more information on CSS and HTML, see our Resource Centers at [www.deitel.com/ResourceCenters.html](http://www.deitel.com/ResourceCenters.html).

Styles are limited to modifying a control's look-and-feel through its properties. In Section 25.11, we introduce control templates, which offer you the freedom to define a control's appearance by modifying its visual structure. With a custom control template,

you can completely strip a control of all its visual settings and rebuild it to look exactly the way you like, while maintaining its existing functionality. A **Button** with a custom control template might look structurally different from a default **Button**, but it still functions the same as any other **Button**.

If you want to change only the appearance of an element, a style or control template should suffice. However, you also can create entirely new custom controls that have their own functionality, properties, methods and events.

## 25.9 Using Styles to Change the Appearance of Controls

Once defined, a **WPF style** is a collection of property-value and event-handler definitions that can be reused. Styles enable you to eliminate repetitive code or markup. For example, if you want to change the look-and-feel of the standard **Button** throughout a section of your app, you can define a style and apply it to all the **Buttons** in that section. Without styles, you have to set the properties for each individual **Button**. Furthermore, if you later decided that you wanted to tweak the appearance of these **Buttons**, you would have to modify your markup or code several times. By using a style, you can make the change only once in the style and it's automatically be applied to any control which uses that style.

Styles are **WPF resources**. A resource is an object that's defined for an entire section of your app and can be reused multiple times. A resource can be as simple as a property or as complex as a control template. Every WPF control can hold a collection of resources that can be accessed by any element down the containment hierarchy. In a way, this is similar in approach to the concept of variable scope that you learned about in Chapter 7. For example, if you define a style as a resource of a **Window**, then any element in the **Window** can use that style. If you define a style as a resource of a layout container, then only the elements of the layout container can use that style. You also can define application-wide resources for an **Application** object in the **App.xaml** file. These resources can be accessed in any file in the app.

### *Color Chooser App*

Figure 25.15 provides the XAML markup and Fig. 25.16 provides the C# code for a color-chooser app. This example demonstrates styles and introduces the **Slider** user input control.

---

```

1  <!-- Fig. 25.15: MainWindow.xaml -->
2  <!-- Color chooser app showing the use of styles (XAML). -->
3  <Window x:Class="ColorChooser.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Color Chooser" Height="150" Width="500">
7
8  <Window.Resources> <!-- define Window's resources -->
9      <Style x:Key="SliderStyle"> <!-- define style for Sliders -->
10
11         <!-- set properties for Sliders -->
12         <Setter Property="Slider.Width" Value="256" />
13         <Setter Property="Slider.Minimum" Value="0" />
14         <Setter Property="Slider.Maximum" Value="255" />

```

---

**Fig. 25.15** | Color-chooser app showing the use of styles (XAML). (Part 1 of 3.)

```
15   <Setter Property="Slider.IsSnapToTickEnabled" Value="True" />
16   <Setter Property="Slider.VerticalAlignment" Value="Center" />
17   <Setter Property="Slider.HorizontalAlignment" Value="Center" />
18   <Setter Property="Slider.Value" Value="0" />
19   <Setter Property="Slider.AutoToolTipPlacement"
20     Value="TopLeft" />
21
22   <!-- set event handler for ValueChanged event -->
23   <EventSetter Event="Slider.ValueChanged"
24     Handler="slider_ValueChanged" />
25   </Style>
26 </Window.Resources>
27
28 <Grid Margin="5"> <!-- define GUI -->
29   <Grid.RowDefinitions>
30     <RowDefinition />
31     <RowDefinition />
32     <RowDefinition />
33     <RowDefinition />
34   </Grid.RowDefinitions>
35   <Grid.ColumnDefinitions>
36     <ColumnDefinition Width="Auto" />
37     <ColumnDefinition Width="Auto" />
38     <ColumnDefinition Width="50" />
39     <ColumnDefinition />
40   </Grid.ColumnDefinitions>
41
42   <!-- define Labels for Sliders -->
43   <Label Grid.Row="0" Grid.Column="0" HorizontalAlignment="Right"
44     VerticalAlignment="Center" Content="Red:"/>
45   <Label Grid.Row="1" Grid.Column="0" HorizontalAlignment="Right"
46     VerticalAlignment="Center" Content="Green:"/>
47   <Label Grid.Row="2" Grid.Column="0" HorizontalAlignment="Right"
48     VerticalAlignment="Center" Content="Blue:"/>
49   <Label Grid.Row="3" Grid.Column="0" HorizontalAlignment="Right"
50     VerticalAlignment="Center" Content="Alpha:"/>
51
52   <!-- define Label that displays the color -->
53   <Label x:Name="colorLabel" Grid.RowSpan="4" Grid.Column="3"
54     Margin="10" />
55
56   <!-- define Sliders and apply style to them -->
57   <Slider x:Name="redSlider" Grid.Row="0" Grid.Column="1"
58     Style="{StaticResource SliderStyle}"
59     Value="{Binding Text, ElementName=redBox}" />
60   <Slider x:Name="greenSlider" Grid.Row="1" Grid.Column="1"
61     Style="{StaticResource SliderStyle}"
62     Value="{Binding Text, ElementName=greenBox1}" />
63   <Slider x:Name="blueSlider" Grid.Row="2" Grid.Column="1"
64     Style="{StaticResource SliderStyle}"
65     Value="{Binding Text, ElementName=blueBox}" />
```

**Fig. 25.15** | Color-chooser app showing the use of styles (XAML). (Part 2 of 3.)

```
66      <Slider x:Name="alphaSlider" Grid.Row="3" Grid.Column="1"
67          Style="{StaticResource SliderStyle}"
68          Value="{Binding Text, ElementName=alphaBox}" />
69
70      <TextBox x:Name="redBox" Grid.Row="0" Grid.Column="2"
71          Text="{Binding Value, ElementName=redSlider}"/>
72      <TextBox x:Name="greenBox" Grid.Row="1" Grid.Column="2"
73          Text="{Binding Value, ElementName=greenSlider}"/>
74      <TextBox x:Name="blueBox" Grid.Row="2" Grid.Column="2"
75          Text="{Binding Value, ElementName=blueSlider}"/>
76      <TextBox x:Name="alphaBox" Grid.Row="3" Grid.Column="2"
77          Text="{Binding Value, ElementName=alphaSlider}"/>
78  </Grid>
79 </Window>
```

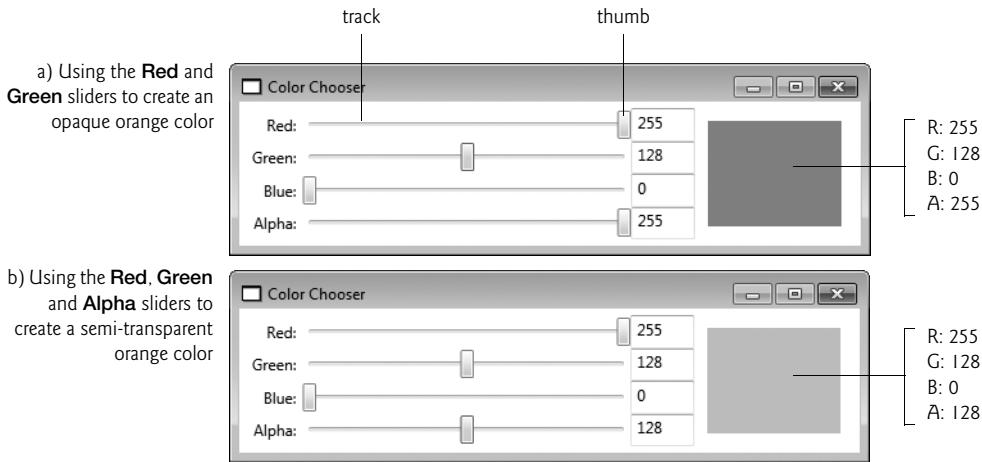
**Fig. 25.15** | Color-chooser app showing the use of styles (XAML). (Part 3 of 3.)

---

```
1 // Fig. 25.16: MainWindow.xaml.cs
2 // Color chooser app showing the use of styles (code-behind).
3 using System.Windows;
4 using System.Windows.Media;
5
6 namespace ColorChooser
7 {
8     public partial class MainWindow : Window
9     {
10         public MainWindow()
11         {
12             InitializeComponent();
13             alphaSlider.Value = 255; // override Value from style
14         } // constructor
15
16         // handles the ValueChanged event for the Sliders
17         private void slider_ValueChanged( object sender,
18             RoutedPropertyChangedEventArgs< double > e )
19         {
20             // generates new color
21             SolidColorBrush backgroundColor = new SolidColorBrush();
22             backgroundColor.Color = Color.FromArgb(
23                 ( byte ) alphaSlider.Value, ( byte ) redSlider.Value,
24                 ( byte ) greenSlider.Value, ( byte ) blueSlider.Value );
25
26             // set colorLabel's background to new color
27             colorLabel.Background = backgroundColor;
28         } // end method slider_ValueChanged
29     } // end class MainWindow
30 } // end namespace ColorChooser
```

**Fig. 25.16** | Color-chooser app showing the use of styles (code-behind). (Part 1 of 2.)

---



**Fig. 25.16** | Color-chooser app showing the use of styles (code-behind). (Part 2 of 2.)

### *RGBA Colors*

This app uses the RGBA color system. Every color is represented by its red, green and blue color values, each ranging from 0 to 255, where 0 denotes no color and 255 full color. For example, a color with a red value of 0 would contain no red component. The alpha value (A)—which also ranges from 0 to 255—represents a color’s *opacity*, with 0 being completely *transparent* and 255 completely *opaque*. The two colors in Fig. 25.16’s sample outputs have the same RGB values, but the color displayed in Fig. 25.16(b) is *semitransparent*.

### *Slider Controls*

The color-chooser GUI uses four **Slider** controls that change the RGBA values of a color displayed by a **Label**. Next to each **Slider** is a **TextBox** that displays the **Slider**’s current value. You also can type a number in a **TextBox** to update the value of the corresponding **Slider**. A **Slider** is a numeric user input control that allows users to drag a “thumb” along a track to select the value. Whenever the user moves a **Slider**, the app generates a new color, the corresponding **TextBox** is updated and the **Label** displays the new color as its background. The new color is generated by using class **Color**’s **FromArgb** method, which returns a color based on the four RGBA byte values you pass it (Fig. 25.16, lines 22–24). The color is then applied as the **Background** of the **Label**. Similarly, changing the value of a **TextBox** updates the thumb of the corresponding **Slider** to reflect the change, which then updates the **Label** with the new color. We discuss the updates of the **TextBoxes** shortly.

### *Style for the Sliders*

Styles can be defined as a resource of any control. In the color-chooser app, we defined the style as a resource of the entire **Window**. We also could have defined it as a resource of the **Grid**. To define resources for a control, you set a control’s **Resources** property. Thus, to define a resource for a **Window**, as we did in this example, you would use **Window.Resources** (lines 8–26 in Fig. 25.15). To define a resource for a **Grid**, you’d use **Grid.Resources**.

Style objects can be defined in XAML using the **Style** element. The **x:Key** attribute (i.e., attribute **Key** from the standard XAML namespace) must be set in every style (or

other resource) so that it can be referenced later by other controls (line 9). The children of a `Style` element set properties and define event handlers. A **Setter** sets a property to a specific value (e.g., line 12, which sets the styled `Slider`'s `Width` property to 256). An **EventSetter** specifies the method that responds to an event (e.g., lines 23–24, which specifies that method `slider_ValueChanged` handles the `Slider`'s `ValueChanged` event).

The `Style` in the color-chooser example (`SliderStyle`) primarily uses `Setters`. It lays out the color `Sliders` by specifying the `Width`, `VerticalAlignment` and `HorizontalAlignment` properties (lines 12, 16 and 17). It also sets the `Minimum` and `Maximum` properties, which determine a `Slider`'s range of values (lines 13–14). In line 18, the default `Value` is set to 0. `IsSnapToTickEnabled` is set to `True`, meaning that only values that fall on a “tick” are allowed (line 15). By default, each tick is separated by a value of 1, so this setting makes the styled `Slider` accept only integer values. Lastly, the style also sets the `AutoToolTipPlacement` property, which specifies where a `Slider`'s tooltip should appear, if at all.

Although the `Style` defined in the color-chooser example is clearly meant for `Sliders`, it can be applied to any control. Styles are not control specific. You can make all controls of one type use the same default style by setting the style's **TargetType** attribute to the control type. For example, if we wanted all of the Window's `Sliders` to use a `Style`, we would add `TargetType="Slider"` to the `Style`'s start tag.

### *Using a Style*

To apply a style to a control, you create a **resource binding** between a control's `Style` property and the `Style` resource. You can create a resource binding in XAML by specifying the resource in a **markup extension**—an expression enclosed in curly braces ({}). The form of a markup extension calling a resource is `{ ResourceType ResourceKey }` (for example, `{StaticResource SliderStyle}` in Fig. 25.15, line 58).

### *Static and Dynamic Resources*

There are two types of resources. **Static resources** are applied only at initialization time. **Dynamic resources** are applied every time the resource is modified by the app. To use a style as a *static* resource, use `StaticResource` as the type in the markup extension. To use a style as a *dynamic* resource, use `DynamicResource` as the type. Because styles don't normally change during runtime, they are usually used as static resources. However, using one as a dynamic resource is sometimes necessary, such as when you wish to enable users to customize a style at runtime.

In this app, we apply `SliderStyle` as a static resource to each `Slider` (lines 58, 61, 64 and 67). Once you apply a style to a control, the **Design** view and **Properties** window update to display the control's new appearance settings. If you then modify the control through the **Properties** window, the control itself is updated, not the style.

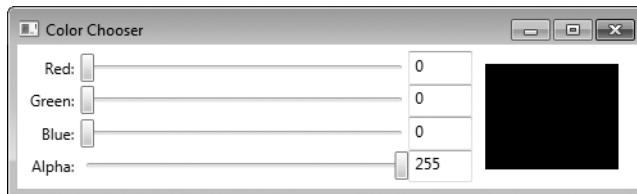
### *Element-to-Element Bindings*

In this app, we use a new feature of WPF called **element-to-element binding** in which a property of one element is always equal to a property of another element. This enables us to declare in XAML that each `TextBox`'s `Text` property should always have the value of the corresponding `Slider`'s `Value` property, and that each `Slider`'s `Value` property should always have the value of the corresponding `TextBox`'s `Text` property. Once these bindings are defined, changing a `Slider` updates the corresponding `TextBox` and vice versa. In Fig. 25.15, lines 59, 62, 65 and 68 each use a `Binding` markup extension to bind a `Slider`'s `Value` property to a `TextBox`'s `Text` property.

er's `Value` property to the `Text` property of the appropriate `TextBox`. Similarly, lines 71, 73, 75 and 77 each use a `Binding` markup extension to bind a `TextBox`'s `Text` property to the `Value` property of the appropriate `Slider`.

### *Programmatically Changing the Alpha Slider's Value*

As shown in Fig. 25.17, the `Slider` that adjusts the alpha value in the color-chooser example starts with a value of 255, whereas the R, G and B `Sliders`' values start at 0. The `Value` property is defined by a `Setter` in the style to be 0 (line 18 in Fig. 25.15). This is why the R, G and B values are 0. The `Value` property of the alpha `Slider` is programmatically defined to be 255 (line 13 in Fig. 25.16), but it could also be set locally in the XAML. Because a local declaration takes precedence over a style setter, the alpha `Slider`'s value would start at 255 when the app loads.



**Fig. 25.17** | GUI of the color-chooser app at initialization.

### *Dependency Properties*

Most WPF properties, though they might look and behave exactly like ordinary ones, are in fact **dependency properties**. Such properties have built-in support for change notification—that is, an app knows and can respond to changes in property values. In addition, they support inheritance down the control-containment hierarchy. For example, when you specify `FontSize` in a `Window`, every control in the `Window` inherits it as the default `FontSize`. You also can specify a control's property in one of its child elements. This is how *attached properties* work.

A control's properties may be set at many different levels in WPF, so instead of holding a fixed value, a *dependency property's value* is determined during execution by a value-determination system. If a property is defined at several levels at once, then the current value is the one defined at the level with the highest precedence. A style, for example, overwrites the default appearance of a control, because it takes higher precedence. A summary of the levels, in order from highest to lowest precedence, is shown in Fig. 25.18.

Levels of value determination system	
Animation	The value is defined by an active animation. For more information about animation, see Chapter 33.
Local declaration	The value is defined as an attribute in XAML or set in code. This is how ordinary properties are set.

**Fig. 25.18** | Levels of value determination from highest to lowest precedence. (Part 1 of 2.)

Levels of value determination system	
Trigger	The value is defined by an active trigger. For more information about triggers, see Section 25.11.
Style	The value is defined by a setter in a style.
Inherited value	The value is inherited from a definition in a containing element.
Default value	The value is not explicitly defined.

**Fig. 25.18** | Levels of value determination from highest to lowest precedence. (Part 2 of 2.)

## 25.10 Customizing Windows

For over a decade, the standard design of an app window has remained practically the same—a framed rectangular box with a header in the top left and a set of buttons in the top right for minimizing, maximizing and closing the window. Cutting-edge apps, however, have begun to use *custom windows* that diverge from this standard to create a more interesting look.

WPF lets you do this more easily. To create a custom window, set the `WindowStyle` property to `None`. This removes the standard frame around your `Window`. To make your `Window` irregularly shaped, you set the `AllowsTransparency` property to `True` and the `Background` property to `Transparent`. If you then add controls, only the space within the boundaries of those controls behaves as part of the window. This works because a user cannot interact with any part of a `Window` that's *transparent*. You still define your `Window` as a rectangle with a width and a height, but when a user clicks in a transparent part of the `Window`, it behaves as if the user clicked *outside* the `Window`'s boundaries—that is, the window does not respond to the click.

Figure 25.19 is the XAML markup that defines a GUI for a circular digital clock. The `Window`'s `WindowStyle` is set to `None` and `AllowsTransparency` is set to `True` (line 7). In this example, we set the background to be an image using an `ImageBrush` (lines 10–12). The background image is a *circle* with a *drop shadow* surrounded by *transparency*. Thus, the `Window` appears *circular*.

---

```

1  <!-- Fig. 25.19: MainWindow.xaml -->
2  <!-- Creating a custom window and using a timer (XAML). -->
3  <Window x:Class="Clock.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Clock" Name="clockWindow" Height="118" Width="118"
7      WindowStyle="None" AllowsTransparency="True"
8      MouseLeftButtonDown="clockWindow_MouseLeftButtonDown">
9
10     <Window.Background> <!-- Set background image -->
11         <ImageBrush ImageSource="images/circle.png" />
12     </Window.Background>
13

```

---

**Fig. 25.19** | Creating a custom window and using a timer (XAML). (Part 1 of 2.)

```
14    <Grid>
15        <TextBox x:Name="timeTextBox" Margin="0,42,0,0"
16            Background="Transparent" TextAlignment="Center"
17            FontWeight="Bold" Foreground="White" FontSize="16"
18            BorderThickness="0" Cursor="Arrow" Focusable="False" />
19    </Grid>
20 </Window>
```



**Fig. 25.19** | Creating a custom window and using a timer (XAML). (Part 2 of 2.)

The time is displayed in the center of the window in a `TextBox` (lines 15–18). Its `Background` is set to `Transparent` so that the text displays directly on the circular background (line 16). We configured the text to be size 16, bold, and white by setting the `FontSize`, `FontWeight`, and `Foreground` properties. The `Cursor` property is set to `Arrow`, so that the mouse cursor doesn't change when it moves over the time (line 18). Setting `Focusable` to `False` disables the user's ability to select the text (line 18).

When you create a custom window, there's no built-in functionality for doing the simple tasks that normal windows do. For example, there is no way for the user to move, resize, minimize, maximize, or close a window unless you write the code to enable these features. You can move the clock around, because we implemented this functionality in the `Window`'s code-behind class (Fig. 25.20). Whenever the left mouse button is held down on the clock (handled by the `MouseLeftButtonDown` event), the `Window` is dragged around using the `DragMove` method (lines 27–31). Because we did not define how to close or minimize the `Window`, you can shut down the clock by pressing *Alt-F4*—this is a feature built into Windows—or by right clicking its icon on the taskbar and selecting **Close window**.

```
1 // Fig. 25.20: MainWindow.xaml.cs
2 // Creating a custom window and using a timer (code-behind).
3 using System;
4 using System.Windows;
5 using System.Windows.Input;
6
7 namespace Clock
8 {
9     public partial class MainWindow : Window
10    {
11        // create a timer to control clock
12        private System.Windows.Threading.DispatcherTimer timer =
13            new System.Windows.Threading.DispatcherTimer();
14    }
}
```

**Fig. 25.20** | Creating a custom window and using a timer (code-behind). (Part 1 of 2.)

```
15    // constructor
16    public MainWindow()
17    {
18        InitializeComponent();
19
20        timer.Interval = TimeSpan.FromSeconds( 1 ); // tick every second
21        timer.IsEnabled = true; // enable timer
22
23        timer.Tick += timer_Tick;
24    } // end constructor
25
26    // drag Window when the left mouse button is held down
27    private void clockWindow_MouseLeftButtonDown( object sender,
28        MouseButtonEventArgs e )
29    {
30        this.DragMove(); // moves the window
31    } // end method clockWindow_MouseLeftButtonDown
32
33    // update the time when the timer ticks
34    private void timer_Tick( object sender, EventArgs e )
35    {
36        DateTime currentTime = DateTime.Now; // get the current time
37
38        // display the time as hh:mm:ss
39        timeTextBox.Text = currentTime.ToString("hh:mm:ss");
40    } // end method timer_Tick
41 } // end class MainWindow
42 } // end namespace Clock
```

---

**Fig. 25.20** | Creating a custom window and using a timer (code-behind). (Part 2 of 2.)

The clock works by getting the current time every second and displaying it in the `TextBox`. To do this, the clock uses a `DispatcherTimer` object (of the `Windows.Threading` namespace), which raises the `Tick` event repeatedly at a prespecified time interval. Since the `DispatcherTimer` is defined in the C# code rather than the XAML, we need to specify the method to handle the `Tick` event in the C# code. Line 23 assigns method `timer_Tick` to the `Tick` event's delegate. This adds the `timer_Tick` method as an `EventHandler` for the specified event. After it's declared, you must specify the interval between `Ticks` by setting the `Interval` property, which takes a `TimeSpan` as its value. `TimeSpan` has several class methods for instantiating a `TimeSpan` object, including `FromSeconds`, which defines a `TimeSpan` lasting the number of seconds you pass to the method. Line 20 creates a one-second `TimeSpan` and sets it as the `DispatcherTimer`'s `Interval`. A `DispatcherTimer` is disabled by default. Until you enable it by setting the `IsEnabled` property to `true` (line 21), it will not `Tick`. In this example, the `Tick` event handler gets the current time and displays it in the `TextBox`.

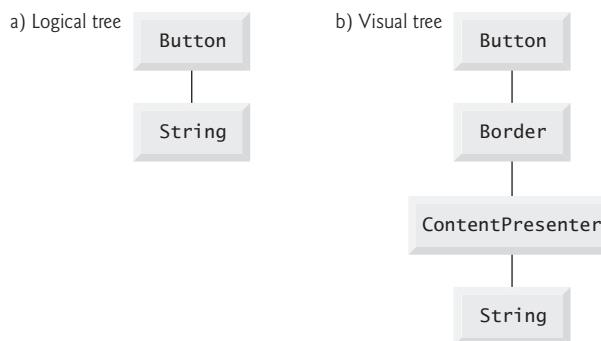
You may recall that the `Timer` component provided the same capabilities in Windows Forms. A similar object that you can drag-and-drop onto your GUI doesn't exist in WPF. Instead, you must create a `DispatcherTimer` object, as illustrated in this example.

## 25.11 Defining a Control's Appearance with Control Templates

We now update the clock example to include buttons for minimizing and closing the app. We also introduce **control templates**—a powerful tool for customizing the look-and-feel of your GUIs. As previously mentioned, a custom control template can redefine the appearance of any control *without* changing its functionality. In Windows Forms, if you want to create a round button, you have to create a new control and simulate the functionality of a **Button**. With control templates, you can simply redefine the visual elements that compose the **Button** control and still use the preexisting functionality.

All WPF controls are **lookless**—that is, a control's properties, methods and events are coded into the control's class, but its *appearance* is *not*. Instead, the appearance of a control is determined by a *control template*, which is a hierarchy of visual elements. Every control has a built-in default control template. All of the GUIs discussed so far in this chapter have used these default templates.

The hierarchy of visual elements defined by a control template can be represented as a tree, called a control's **visual tree**. Figure 25.21(b) shows the visual tree of a default **Button** (Fig. 25.22). This is a more detailed version of the same **Button**'s **logical tree**, which is shown in Fig. 25.21(a). A logical tree depicts how a control is defined, whereas a visual tree depicts how a control is graphically rendered.



**Fig. 25.21** | The logical and visual trees for a default **Button**.

A control's logical tree always mirrors its definition in XAML. For example, you'll notice that the **Button**'s logical tree, which comprises only the **Button** and its **string** caption, exactly represents the hierarchy outlined by its XAML definition, which is

```
<Button>
    Click Me
</Button>
```

To actually render the **Button**, WPF displays a **ContentPresenter** with a **Border** around it. These elements are included in the **Button**'s visual tree. A **ContentPresenter** is an object used to display a single element of content on the screen. It's often used in a template to specify where to display content.



Fig. 25.22 | The default Button.

In the updated clock example, we create a custom control template (named **ButtonTemplate**) for rendering Buttons and apply it to the two Buttons in the app. The XAML markup is shown in Fig. 25.23. Like a style, a control template is usually defined as a resource, and applied by binding a control's **Template** property to the control template using a *resource binding* (for example, lines 47 and 52). After you apply a control template to a control, the **Design** view will update to display the new appearance of the control. The **Properties** window remains unchanged, since a control template does *not* modify a control's properties.

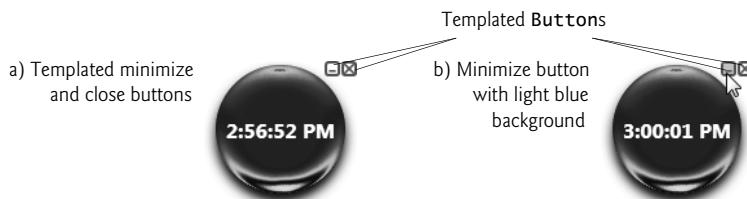
---

```
1  <!-- Fig. 25.23: MainWindow.xaml -->
2  <!-- Using control templates (XAML). -->
3  <Window x:Class="Clock.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Clock" Name="clockWindow" Height="118" Width="118"
7      WindowStyle="None" AllowsTransparency="True"
8      MouseLeftButtonDown="clockWindow_MouseLeftButtonDown">
9
10 <Window.Resources>
11     <!-- control template for Buttons -->
12     <ControlTemplate x:Key="ButtonTemplate" TargetType="Button">
13         <Border Name="Border" BorderThickness="2" CornerRadius="2"
14             BorderBrush="RoyalBlue">
15
16             <!-- Template binding to Button.Content -->
17             <ContentPresenter Margin="0" Width="8"
18                 Content="{TemplateBinding Content}" />
19         </Border>
20
21         <ControlTemplate.Triggers>
22             <!-- if mouse is over the button -->
23             <Trigger Property="IsMouseOver" Value="True">
24                 <!-- make the background blue -->
25                 <Setter TargetName="Border" Property="Background"
26                     Value="LightBlue" />
27             </Trigger>
28         </ControlTemplate.Triggers>
29     </ControlTemplate>
30 </Window.Resources>
31
32     <Window.Background> <!-- Set background image -->
33         <ImageBrush ImageSource="images/circle.png" />
34     </Window.Background>
35
```

Fig. 25.23 | Using control templates (XAML). (Part 1 of 2.)

---

```
36     <Grid>
37         <Grid.RowDefinitions>
38             <RowDefinition Height="Auto" />
39             <RowDefinition />
40         </Grid.RowDefinitions>
41
42         <StackPanel Grid.Row="0" Orientation="Horizontal"
43                     HorizontalAlignment="Right">
44
45             <!-- these buttons use the control template -->
46             <Button x:Name="minimizeButton" Margin="0" Focusable="False"
47                         IsTabStop="False" Template="{StaticResource ButtonTemplate}">
48                 Click="minimizeButton_Click">
49                 <Image Source="images/minimize.png" Margin="0" />
50             </Button>
51             <Button x:Name="closeButton" Margin="1,0,0,0" Focusable="False"
52                         IsTabStop="False" Template="{StaticResource ButtonTemplate}">
53                 Click="closeButton_Click">
54                 <Image Source="images/close.png" Margin="0"/>
55             </Button>
56         </StackPanel>
57
58         <TextBox x:Name="timeTextBox" Grid.Row="1" Margin="0,30,0,0"
59                         Background="Transparent" TextAlign="Center"
60                         FontWeight="Bold" Foreground="White" FontSize="16"
61                         BorderThickness="0" Cursor="Arrow" Focusable="False" />
62     </Grid>
63 </Window>
```



**Fig. 25.23** | Using control templates (XAML). (Part 2 of 2.)

To define a control template in XAML, you create a **ControlTemplate** element. Just as with a style, you must specify the control template's **x:Key** attribute so you can reference it later (line 12). You must also set the **TargetType** attribute to the type of control for which the template is designed (line 12). Inside the **ControlTemplate** element, you can build the control using any WPF visual element (lines 13–19). In this example, we replace the default **Border** and **ContentPresenter** with our own custom ones.

Sometimes, when defining a control template, it may be beneficial to use the value of one of the templated control's properties. For example, if you want several controls of different sizes to use the same control template, you may need to use the values of their **Width** and **Height** properties in the template. WPF allows you to do this with a **template binding**, which can be created in XAML with the markup extension, **{TemplateBinding}**.

*PropertyName}*. To bind a property of an element in a control template to one of the properties of the templated control (that is, the control that the template is applied to), you need to set the appropriate markup extension as the value of that property. In **ButtonTemplate**, we bind the **Content** property of a **ContentPresenter** to the **Content** property of the templated **Button** (line 18). The nested element of a **ContentControl** is the value of its **Content** property. Thus, the images defined in lines 49 and 54 are the **Content** of the **Buttons** and are displayed by the **ContentPresenters** in their respective control templates. You also can create template bindings to a control's events.

Often you'll use a combination of control templates, styles and local declarations to define the appearance of your app. Recall that a control template defines the default appearance of a control and thus has a lower precedence than a style in dependency property-value determination.

### Triggers

The control template for **Buttons** used in the updated clock example defines a **trigger**, which changes a control's appearance when that control enters a certain state. For example, when your mouse is over the clock's minimize or close Buttons, the **Button** is highlighted with a light blue background. This simple change in appearance is caused by a trigger that fires whenever the **IsMouseOver** property becomes **True**.

A trigger must be defined in the **Style.Triggers** or **ControlTemplate.Triggers** element of a style or a control template, respectively (for example, lines 21–28). You can create a trigger by defining a **Trigger** object. The **Property** and **Value** attributes define the state when a trigger is active. Setters *nested* in the **Trigger** element are carried out when the trigger is fired. When the trigger no longer applies, the changes are removed. A Setter's **TargetName** property specifies the name of the element that the **Setter** applies to (for example, line 25).

Lines 23–27 define the **IsMouseOver** trigger for the minimize and close **Buttons**. When the mouse is over the **Button**, **IsMouseOver** becomes **True**, and the trigger becomes active. The trigger's **Setter** makes the background of the **Border** in the control template temporarily light blue. When the mouse exits the boundaries of the **Button**, **IsMouseOver** becomes **False**. Thus, the **Border**'s background returns to its *default setting*, which in this case is transparent.

### Functionality

Figure 25.24 shows the code-behind class for the clock app. Although the custom control template makes the **Buttons** in this app look different, it doesn't change how they behave. Lines 3–40 remain unchanged from the code in the first clock example (Fig. 25.20). The functionality for the minimize and close **Buttons** is implemented in the same way as any other button—by handling the **Click** event (lines 43–47 and 50–53 of Fig. 25.24, respectively). To minimize the window, we set the **WindowState** of the **Window** to **WindowState.Minimized** (line 46).

---

```

1 // Fig. 25.24: MainWindow.xaml.cs
2 // Using control templates (code-behind).
3 using System;

```

---

**Fig. 25.24** | Using control templates (code-behind). (Part 1 of 2.)

```
4  using System.Windows;
5  using System.Windows.Input;
6
7  namespace Clock
8  {
9      public partial class MainWindow : Window
10     {
11         // creates a timer to control clock
12         private System.Windows.Threading.DispatcherTimer timer =
13             new System.Windows.Threading.DispatcherTimer();
14
15         // constructor
16         public MainWindow()
17         {
18             InitializeComponent();
19
20             timer.Interval = TimeSpan.FromSeconds( 1 ); // tick every second
21             timer.IsEnabled = true; // enable timer
22
23             timer.Tick += timer_Tick;
24         } // end constructor
25
26         // drag Window when the left mouse button is held down
27         private void clockWindow_MouseLeftButtonDown( object sender,
28             MouseButtonEventArgs e )
29         {
30             this.DragMove();
31         } // end method clockWindow_MouseLeftButtonDown
32
33         // update the time when the timer ticks
34         private void timer_Tick( object sender, EventArgs e )
35         {
36             DateTime currentTime = DateTime.Now; // get the current time
37
38             // display the time as hh:mm:ss
39             timeTextBox.Text = currentTime.ToString("hh:mm:ss");
40         } // end method timer_Tick
41
42         // minimize the app
43         private void minimizeButton_Click( object sender,
44             RoutedEventArgs e )
45         {
46             this.WindowState = WindowState.Minimized; // minimize window
47         } // end method minimizeButton_Click
48
49         // close the app
50         private void closeButton_Click( object sender, RoutedEventArgs e )
51         {
52             Application.Current.Shutdown(); // shut down app
53         } // end method closeButton_Click
54     } // end class MainWindow
55 } // end namespace Clock
```

---

**Fig. 25.24** | Using control templates (code-behind). (Part 2 of 2.)

## 25.12 Data-Driven GUIs with Data Binding

WPF provides a comprehensive model for allowing GUIs to interact with data.

### Bindings

A **data binding** is a pointer to data, represented by a **Binding** object. WPF allows you to create a *binding* to a broad range of data types. At the simplest level, you could create a binding to a single property. Often, however, it's useful to create a binding to a data object—an object of a class with properties that describe the data. You also can create a binding to objects like arrays, collections and data in an XML document. The versatility of the WPF data model even allows you to bind to data represented by LINQ statements.

Like other binding types, a *data binding* can be created declaratively in XAML markup with a *markup extension*. To declare a data binding, you must specify the data's *source*. If it's another element in the XAML markup, use property **ElementName**. Otherwise, use **Source**. Then, if you're binding to a specific data point of the source, such as a property of a control, you must specify the **Path** to that piece of information. Use a comma to separate the binding's property declarations. For example, to create a binding to a control's property, you would use {Binding ElementName=ControlName, Path=PropertyName}.

Figure 25.25 presents the XAML markup of a book-cover viewer that lets the user select from a list of books, and displays the cover of the currently selected book. The list of books is presented in a **ListView** control (lines 15–24), which displays a set of data as items in a selectable list. Its current selection can be retrieved from the **SelectedItem** property. A large image of the currently selected book's cover is displayed in an **Image** control (lines 27–28), which automatically updates when the user makes a new selection. Each book is represented by a **Book** object, which has four **string** properties:

1. **ThumbImage**—the full path to the small cover image of the book.
2. **LargeImage**—the full path to the large cover image of the book.
3. **Title**—the title of the book.
4. **ISBN**—the 10-digit ISBN of the book.

Class **Book** also contains a constructor that initializes a **Book** and sets each of its properties. The full source code of the **Book** class is not presented here but you can view it in the IDE by opening this example's project.

---

```

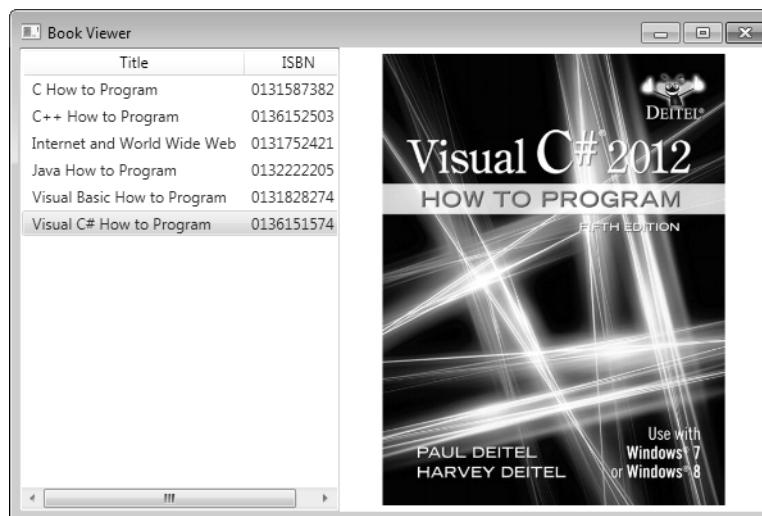
1  <!-- Fig. 25.25: MainWindow.xaml -->
2  <!-- Using data binding (XAML). -->
3  <Window x:Class="BookViewer.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Book Viewer" Height="400" Width="600">
7
8      <Grid> <!-- define GUI -->
9          <Grid.ColumnDefinitions>
10             <ColumnDefinition Width="Auto" />
11             <ColumnDefinition />
12         </Grid.ColumnDefinitions>

```

---

**Fig. 25.25** | Using data binding (XAML). (Part 1 of 2.)

```
13      <!-- use ListView and GridView to display data -->
14      <ListView x:Name="booksListView" Grid.Column="0" MaxWidth="250">
15          <ListView.View>
16              <GridView>
17                  <GridViewColumn Header="Title" Width="100"
18                      DisplayMemberBinding="{Binding Path=Title}" />
19                  <GridViewColumn Header="ISBN" Width="80"
20                      DisplayMemberBinding="{Binding Path=ISBN}" />
21              </GridView>
22          </ListView.View>
23      </ListView>
24
25
26      <!-- bind to selected item's full-size image -->
27      <Image Grid.Column="1" Source="{Binding ElementName=booksListView,
28          Path=SelectedItem.LargeImage}" Margin="5" />
29
30  </Grid>
</Window>
```



**Fig. 25.25** | Using data binding (XAML). (Part 2 of 2.)

To synchronize the book cover that's being displayed with the currently selected book, we bind the `Image`'s `Source` property to the file location of the currently selected book's large cover image (lines 27–28). The `Binding`'s `ElementName` property is the name of the selector control, `booksListView`. The `Path` property is `SelectedItem.LargeImage`. This indicates that the binding should be linked to the `LargeImage` property of the `Book` object that's currently `booksListView`'s `SelectedItem`.

Some controls have built-in support for data binding, and a separate `Binding` object doesn't need to be created. A `ListView`, for example, has a built-in `ItemsSource` property that specifies the *data source* from which the items of the list are determined. There is no need to create a binding—instead, you can just set the `ItemsSource` property as you would any other property. When you set `ItemsSource` to a collection of data, the objects in the

collection automatically become the items in the list. Figure 25.26 presents the code-behind class for the book-cover viewer. When the `Window` is created, a collection of six `Book` objects is initialized (lines 17–29) and set as the `ItemsSource` of the `booksListView`, meaning that each item displayed in the selector is one of the `Books`.

---

```

1 // Fig. 25.26: MainWindow.xaml.cs
2 // Using data binding (code-behind).
3 using System.Collections.Generic;
4 using System.Windows;
5
6 namespace BookViewer
7 {
8     public partial class MainWindow : Window
9     {
10         private List< Book > books = new List< Book >();
11
12         public MainWindow()
13         {
14             InitializeComponent();
15
16             // add Book objects to the List
17             books.Add( new Book( "C How to Program", "013299044X",
18                 "images/small/chtp.jpg", "images/large/chtp.jpg" ) );
19             books.Add( new Book( "C++ How to Program", "0133378713",
20                 "images/small/cpphtp.jpg", "images/large/cpphtp.jpg" ) );
21             books.Add( new Book(
22                 "Internet and World Wide Web How to Program", "0132151006",
23                 "images/small/iw3htp.jpg", "images/large/iw3htp.jpg" ) );
24             books.Add( new Book( "Java How to Program", "0132940949",
25                 "images/small/jhttp.jpg", "images/large/jhttp.jpg" ) );
26             books.Add( new Book( "Visual Basic How to Program", "0133406954",
27                 "images/small/vbhtp.jpg", "images/large/vbhtp.jpg" ) );
28             books.Add( new Book( "Visual C# How to Program", "0133379337",
29                 "images/small/vcshtp.jpg", "images/large/vcshtp.jpg" ) );
30
31             booksListView.ItemsSource = books; // bind data to the list
32         } // end constructor
33     } // end class MainWindow
34 } // end namespace BookViewer

```

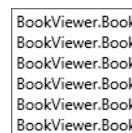
---

**Fig. 25.26** | Using data binding (code-behind).

### Displaying Data in the ListView

For a `ListView` to display objects in a useful manner, you must specify how. For example, if you don't specify how to display each `Book`, the `ListView` simply displays the result of the item's `ToString` method, as shown in Fig. 25.27.

There are many ways to format the display of a `ListView`. One such method is to display each item as a row in a *tabular grid*, as shown in Fig. 25.25. This can be achieved by setting a `GridView` as the `View` property of a `ListView` (lines 16–23). A `GridView` consists of many `GridViewColumns`, each representing a property. In this example, we define two columns, one for `Title` and one for `ISBN` (lines 18–19 and 20–21, respectively). A `GridViewColumn`'s `Header` property specifies what to display as its header. The values displayed



**Fig. 25.27** | ListView display with no data template.

in each column are determined by its **DisplayMemberBinding** property. We set the **Title** column's **DisplayMemberBinding** to a **Binding** object that points to the **Title** property (line 19), and the **ISBN** column's to one that points to the **ISBN** property (line 21). Neither of the **Bindings** has a specified **ElementName** or **Source**. Because the **ListView** has already specified the data source (line 31 of Fig. 25.26), the two data bindings inherit this source, and we do not need specify it again.

### Data Templates

A much more powerful technique for formatting a **ListView** is to specify a template for displaying each item in the list. This template defines how to display bound data and is called a **data template**. Figure 25.28 is the XAML markup that describes a modified version of the book-cover viewer GUI. Each book, instead of being displayed as a row in a table, is represented by a small thumbnail of its cover image with its title and ISBN. Lines 11–32 define the data template (that is, a **DataTemplate** object) that specifies how to display a **Book** object. Note the similarity between the structure of a data template and that of a control template. If you define a data template as a resource, you apply it by using a resource binding, just as you would a style or control template. To apply a data template to items in a **ListView**, use the **ItemTemplate** property (for example, line 43).

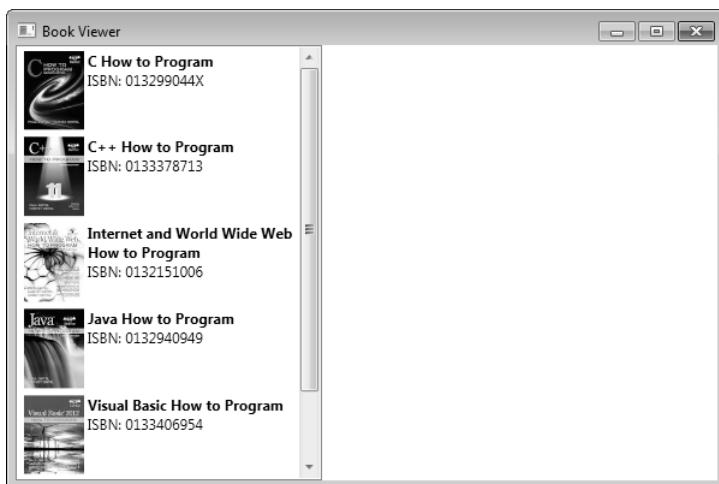
---

```
1  <!-- Fig. 25.28: MainWindow.xaml -->
2  <!-- Using data templates (XAML). -->
3  <Window x:Class="BookViewer.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Book Viewer" Height="400" Width="600" Name="bookViewerWindow">
7
8  <Window.Resources> <!-- Define Window's resources -->
9
10 <!-- define data template -->
11 <DataTemplate x:Key="BookTemplate">
12     <Grid MaxWidth="250" Margin="3">
13         <Grid.ColumnDefinitions>
14             <ColumnDefinition Width="Auto" />
15             <ColumnDefinition />
16         </Grid.ColumnDefinitions>
17
18         <!-- bind image source -->
19         <Image Grid.Column="0" Source="{Binding Path=ThumbImage}"
20             Width="50" />
21     </Grid>
```

**Fig. 25.28** | Using data templates (XAML). (Part 1 of 3.)

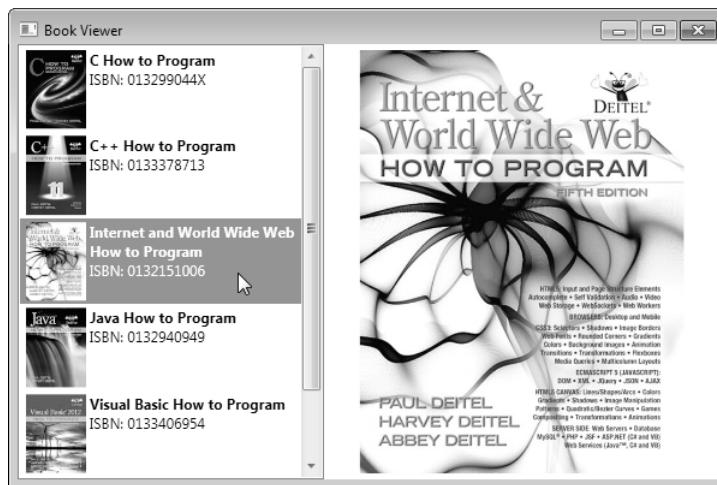
```
22     <StackPanel Grid.Column="1">
23         <!-- bind Title and ISBN -->
24         <TextBlock Margin="3,0" Text="{Binding Path=Title}"
25             FontWeight="Bold" TextWrapping="Wrap" />
26         <StackPanel Margin="3,0" Orientation="Horizontal">
27             <TextBlock Text="ISBN: " />
28             <TextBlock Text="{Binding Path=ISBN}" />
29         </StackPanel>
30     </StackPanel>
31 </DataTemplate>
32 </Window.Resources>
33
34 <Grid> <!-- define GUI -->
35     <Grid.ColumnDefinitions>
36         <ColumnDefinition Width="Auto" />
37         <ColumnDefinition />
38     </Grid.ColumnDefinitions>
39
40     <!-- use ListView and template to display data -->
41     <ListView x:Name="booksListView" Grid.Column="0"
42         ItemTemplate="{StaticResource BookTemplate}" />
43
44     <!-- bind to selected item's full-size image -->
45     <Image Grid.Column="1" Source="{Binding ElementName=booksListView,
46         Path=SelectedItem.LargeImage}" Margin="5" />
47 </Grid>
48 </Window>
```

- a) App showing the `ListView` with the `DataTemplate` applied to its items



**Fig. 25.28** | Using data templates (XAML). (Part 2 of 3.)

b) Selecting an item from the ListView



**Fig. 25.28** | Using data templates (XAML). (Part 3 of 3.)

A data template uses data bindings to specify how to display data. Once again, we can omit the data binding's `ElementName` and `Source` properties, because its source has already been specified by the `ListView` (line 31 of Fig. 25.26). The same principle can be applied in other scenarios as well. If you bind an element's `DataContext` property to a data source, then its child elements can access data within that source without your having to specify it again. In other words, if a binding already has a context (i.e., a `DataContext` has already been defined by a parent), it automatically inherits the data source. For example, if you bind a data source to the `DataContext` property of a `Grid`, then any data binding created in the `Grid` uses that source by default. You can, however, *override* this source by explicitly defining a new one when you define a binding.

In the `BookTemplate` data template, lines 19–20 of Fig. 25.28 define an `Image` whose `Source` is bound to the `Book`'s `Thumbnail` property, which stores the relative file path to the thumbnail cover image. The `Book`'s `Title` and `ISBN` are displayed to the right of the book using `TextBlocks`—lightweight controls for displaying text. The `TextBlock` in lines 24–25 displays the `Book`'s `Title` because the `Text` property is bound to it. Because some of the books' titles are long, we set the `TextWrapping` property to `Wrap` (line 25) so that, if the title is too long, it will wrap to multiple lines. We also set the `FontWeight` property to `Bold`. Lines 26–29 display two additional `TextBlocks`, one that displays `ISBN:`, and another that's bound to the `Book`'s `ISBN` property.

Figure 25.28(a) shows the book-viewer app when it first loads. Each item in the `ListView` is represented by a thumbnail of its cover image, its title and its `ISBN`, as specified in the data template. As illustrated by Fig. 25.28(b), when you select an item in the `ListView`, the large cover image on the right automatically updates, because it's bound to the `SelectedItem` property of the list.

### Collection Views

A **collection view** (of class type **CollectionView**) is a *wrapper* around a collection of data and can provide multiple “views” of the data based on how it’s filtered, sorted and grouped. A *default view* is created in the background every time a *data binding* is created. To retrieve the collection view, use the **CollectionViewSource.GetDefaultView** method and pass it the source of your data binding. For example, to retrieve the default view of bookListView, you’d use `CollectionViewSource.GetDefaultView(bookListView.ItemsSource)`.

You can then modify the view to create the exact view of the data that you want to display. The methods of filtering, sorting and grouping data are beyond the scope of this book. For more information, see [msdn.microsoft.com/en-us/library/ms752347.aspx#what\\_are\\_collection\\_views](http://msdn.microsoft.com/en-us/library/ms752347.aspx#what_are_collection_views).

### Asynchronous Data Binding

Sometimes you may wish to create *asynchronous data bindings* that don’t hold up your app while data is being transmitted. To do this, you set the **IsAsync** property of a data binding to True (it’s False by default). Often, however, it’s not the transmission but the instantiation of data that’s the most expensive operation. An asynchronous data binding does not provide a solution for instantiating data asynchronously. To do so, you must use a **data provider**, a class that can create or retrieve data. There are two types of data providers, **XmlDataProvider** (for XML) and **ObjectDataProvider** (for data objects). Both can be declared as resources in XAML markup. If you set a data provider’s **IsAsynchronous** property to True, the provider will run in the background. Creating and using data providers is beyond the scope of this book. See [msdn.microsoft.com/en-us/library/aa480224.aspx](http://msdn.microsoft.com/en-us/library/aa480224.aspx) for more information.

## 25.13 Wrap-Up

Many of today’s commercial apps provide GUIs that are easy to use and manipulate. The demand for sophisticated and user-friendly GUIs makes GUI design an essential programming skill. In Chapters 14–15, we showed you how to create GUIs with Windows Forms. In this chapter, we demonstrated how to create GUIs with WPF. You learned how to design a WPF GUI with XAML markup and how to give it functionality in a C# code-behind class. We presented WPF’s new flow-based layout scheme, in which a control’s size and position are both defined relatively. You learned not only to handle events just as you did in a Windows Forms app, but also to implement WPF commands when you want multiple user interactions to execute the same task. We demonstrated the flexibility WPF offers for customizing the look-and-feel of your GUIs. You learned how to use styles, control templates and triggers to define a control’s appearance. The chapter concluded with a demonstration of how to create data-driven GUIs with data bindings and data templates.

But WPF is not merely a GUI-building platform. Chapter 26 explores some of the many other capabilities of WPF, showing you how to incorporate 2D and 3D graphics, animation and multimedia into your WPF apps.

## Summary

### Section 25.2 Windows Presentation Foundation (WPF)

- Windows Presentation Foundation (WPF) is a single platform capable of integrating GUI components, images, animation, 2D graphics, 3D graphics, audio and video capabilities.
- WPF generates vector-based graphics and is resolution independent.
- Just as in Windows Forms, you use predefined controls to create GUI elements in WPF, and the functionality of a WPF GUI is event driven.
- In WPF, you can write XAML markup to define the layout and appearance of a GUI.
- Declarative programming is defining *what* something is without specifying *how* to generate it. Writing XAML markup to define GUI controls is an example of declarative programming.
- XAML separates front-end appearance from back-end logic, allowing designers and programmers to work together more efficiently.

### Section 25.3 Declarative GUI Programming Using XAML

- WPF controls are represented by elements in XAML markup.
- Every XAML document must have a single root element.
- The presentation XAML namespace and the standard XAML namespace must be defined with the `Window` control in every XAML document so that the XAML compiler can interpret your markup.
- A XAML document must have an associated code-behind class in order to handle events. The `x:Class` attribute of `Window` specifies the class name of the code-behind class.
- A `Window` is a content control that can have exactly one child element or text. To place multiple controls in a `Window`, you must set a layout container as the `Window`'s child element.
- WPF `Labels` are content controls that typically display a small amount of data.

### Section 25.4 Creating a WPF App

- When you open a XAML document in Visual Studio, a `XAML view` that allows you to edit XAML markup appears in your IDE.
- Every WPF project has an `App.xaml` document that defines the `Application` object and its settings and an `App.xaml.cs` code-behind class that handles application-wide events.
- The `StartupUri` attribute in `App.xaml` specifies the XAML document that executes first.
- The file name of a code-behind file is the file name of the XAML document followed by `.cs`.

### Section 25.5.1 General Layout Principles

- WPF layout is flow based and defined relatively.
- A control's size should be defined as a range of acceptable sizes using the `MinHeight`, `MinWidth`, `MaxHeight` and `MaxWidth` properties.
- A control's position should be defined based on its position relative to the layout container in which it's included and the other controls in the same container.
- A control's `Margin` specifies how much space to put around the edges of a control.
- A control's `HorizontalAlignment` and `VerticalAlignment` specify how to align a control within its layout container.

### Section 25.5.2 Layout in Action

- WPF `RadioButtons` represent mutually exclusive options.

- WPF Buttons are objects that respond to mouse clicks.
- A WPF GroupBox surrounds its child element with a titled box.
- A StackPanel is a layout container that arranges its content vertically or horizontally.
- A Grid is a flexible, all-purpose layout container that organizes controls into a user-defined grid of rows and columns.
- The RowDefinitions and ColumnDefinitions properties of Grid define its rows and columns. They take collections of RowDefinition and ColumnDefinition objects, respectively. You can specify the Width of a ColumnDefinition and the Height of a RowDefinition to be an explicit size, a relative size (using \*) or Auto.
- You can use the Collection Editor to edit properties that take collection values, such as RowDefinitions or ColumnDefinitions.
- The Grid.Row and Grid.Column properties specify a control's location in the Grid.
- To specify the number of rows or columns of a Grid that a contained control spans, use the Grid.RowSpan or Grid.ColumnSpan attached properties, respectively.
- Attached properties are defined by a parent control and used by its children.
- A Canvas is a layout container that allows users to position controls in absolute terms.
- The Canvas.Left and Canvas.Top properties specify a control's coordinate position based on its distance from the left and top borders, respectively, of the Canvas in which it's contained. If two controls overlap, the one with the greater Canvas.ZIndex displays in front.
- In Design mode, snaplines, margin lines, and gridlines help you lay out your GUI.

### *Section 25.6 Event Handling*

- WPF event handling is similar to Windows Forms event handling.
- All layout containers have a Children property that stores all of a layout container's children. You can manipulate the content of a layout container as you would any other IEnumerable.
- A WPF RadioButton raises the Checked event every time it becomes checked.
- A WPF Button raises the Click event every time it's clicked.
- Just as in Windows Forms, WPF has built-in support for keyboard and mouse events.
- Class MouseEventArgs contains mouse-specific information.
- Class MouseButtonEventArgs contains information specific to a mouse button.
- The MouseLeftButtonDown and MouseRightButtonDown events are raised when the user presses the left and right mouse buttons, respectively.
- The MouseLeftButtonUp and MouseRightButtonUp events are raised when the user releases the left and right mouse buttons, respectively.
- WPF events can travel either up or down the containment hierarchy of controls. This is called event routing, and all WPF events are routed events. A RoutedEventArgs parameter contains information specific to routed events. If its Handled property is True, then event handlers ignore the event. The Source property specifies where the event occurred.
- WPF TextBoxes display editable unformatted text.
- There are three types of routed events. Direct events do not travel up or down the containment hierarchy. Bubbling events start at the Source and travel up the hierarchy, ending at the Window or until you set Handled to true. Tunneling events start at the top and travel down the hierarchy until they reach the Source or Handled is true.
- A tunneling event has the same name as the corresponding bubbling event prefixed by Preview.

### **Section 25.7 Commands and Common Application Tasks**

- A WPF command is an action or a task that may be triggered by many different user interactions.
- Commands enable you to synchronize the availability of a task to the state of its corresponding controls and keyboard shortcuts.
- Commands are implementation of the `ICommand` interface. Every `ICommand` has a `Name` and a collection of `InputGestures` (keyboard shortcuts) associated with it.
- When an `ICommand` executes, its `Execute` method is called. The command's execution logic is not defined in its `Execute` method.
- An `ICommand`'s `CanExecute` method is called to determine when the command should be enabled. The logic of the `ICommand` is not defined in this method.
- WPF provides a command library of built-in commands. These commands have their standard keyboard shortcuts already associated with them.
- A command executes when it's triggered by a command source.
- To use a command, you must specify a method that handles either the `Executed` or the `PreviewExecuted` event in a command binding. You can declare a command binding by creating a `CommandBinding` object in XAML, setting the `Command` attribute to the command, and setting the value of either the `Executed` or `PreviewExecuted` attribute to the event-handler name.
- You can declare an event handler for any event in XAML by setting the event's attribute to be the name of the event-handler method.
- `Application.Current.Shutdown` shuts down an app.
- Command bindings should be defined within the `Window.CommandBindings` element.
- Some controls have built-in functionality for using certain commands. In these cases, a command binding is not necessary.
- A `RichTextBox` allows users to enter, edit and format text.
- A `Menu` creates a menu containing `MenuItem`s. If a `MenuItem` contains additional nested `MenuItem`s, then it's a top-level menu or a submenu. Otherwise, it's a menu option, and it executes an action via either an event or a command.
- A `ToolBar` is a single row or column of controls. `ToolBars` overwrite the look-and-feel of their child elements with their own specifications so that the controls look seamless together. You can place multiple `ToolBars` inside a `ToolBarTray`.
- A `Separator` creates a divider to separate content into groups. You can use `Separators` in any control that can contain multiple child elements.

### **Section 25.8 WPF GUI Customization**

- You can change the appearance of a WPF control just by adjusting its properties.
- You can define the appearance of many controls at once by creating and applying a style.
- With a customized control template, you can completely redefine the appearance of a control without changing its functionality.
- You can create a custom control with its own functionality, properties, methods and events.

### **Section 25.9 Using Styles to Change the Appearance of Controls**

- Styles can define property values and event handlers.
- A resource is an object that's defined for an entire section of your app and can be reused multiple times. Every WPF control can hold a collection of resources that can be accessed by its children. To define resources for a control, you use a control's `Resources` property.

- A **Slider** is a numeric user input control that allows users to drag a “thumb” along a track to select the value.
- Style objects are easily defined in XAML using the **Style** element.
- A **Setter** sets a property to a certain value.
- An **EventSetter** specifies the method that responds to an event.
- You can make all controls of one type automatically use the same default style by setting the style’s **TargetType** attribute to the control type.
- You can create a resource binding in XAML by calling the resource in a markup extension. The form of a markup extension calling a resource is `{Resource Type ResourceKey}`.
- There are two types of resources. Static resources are applied at initialization time only. Dynamic resources are applied every time the resource is modified by the app.
- Most WPF properties are dependency properties. A dependency property’s value is determined during execution by a value-determination system. If a property is defined at several levels, then the current value is the one defined at the level with the highest precedence.

### ***Section 25.10 Customizing Windows***

- To create a custom window, set the **WindowStyle** property to **None**. This removes the standard frame around your **Window**.
- To make an irregularly shaped **Window**, set the **AllowsTransparency** property to **True** and the **Background** property to **Transparent**.
- A **Window** can be dragged around the screen with the **DragMove** method.
- A **DispatcherTimer** object is designed to raise the **Tick** event once every prespecified time interval. To use it, you must specify the **Interval** between **Ticks** and set **.IsEnabled** to **True**.

### ***Section 25.11 Defining a Control’s Appearance with Control Templates***

- All WPF controls are lookless. A control’s properties, methods and events are coded into the control’s class, but its appearance is not.
- A control’s appearance is determined by a control template.
- A control’s logical tree depicts how it’s defined, and a control’s visual tree depicts how it’s graphically rendered. A control template defines a control’s visual tree.
- A control template is usually defined as a resource and applied by binding a control’s **Template** property to the control template using a resource binding.
- To define a control template in XAML, you create a **ControlTemplate** element. You must set the **TargetType** attribute to the type of control for which the template is designed. Inside the **ControlTemplate** element, you have the freedom to build the control using any WPF visual element.
- You can use a control’s property within the definition of its control template by using a template binding, which can be created in XAML with the markup extension, `{TemplateBinding PropertyName}`.
- A trigger changes the appearance of a control when that control enters a certain state.
- A trigger must be contained within a style or a control template. You can create a trigger with a **Trigger** object. Setters nested in the **Trigger** element are carried out when the trigger is fired. When the trigger no longer applies, the changes are removed.
- To minimize a window, set the **WindowState** property to **Windows.WindowState.Minimized**.

### ***Section 25.12 Data-Driven GUIs with Data Binding***

- A data binding is a pointer to some type of data, represented by a **Binding** object.

- A data binding can be created declaratively in XAML markup using a markup extension. You must specify the data source by setting either the `Source` or `ElementName` property. To obtain a specific data point, you must specify the `Path` relative to the data source.
- A `ListView` displays data in a selectable list format.
- A `ListView` has a built-in `ItemsSource` property that specifies the data source from which the items of the list are determined.
- A `GridView` displays each item of a `ListView` as a row in a grid.
- A `GridViewColumn` represents a column of a `GridView`. The `DisplayMemberBinding` property specifies the data field that's displayed.
- A `DataTemplate` object that defines how to display data can be created and used as a resource.
- To apply a data template to a `ListView`, bind the `ItemsTemplate` property to the template.
- If you bind an element's `DataContext` to a data object, then its child elements can access specific data points within that object without respecifying it as the source.
- A collection view (of class type `CollectionView`) is a wrapper around a collection of data that can provide us with multiple “views” of the same data based on how we filter, sort and group the data.
- A data provider is a class that can create or retrieve data. There are two types of data providers, `XmlDataProvider` (for XML) and `ObjectDataProvider` (for data objects). If you set a data provider's `IsAsynchronous` property to `True`, the provider will run in the background.

## Terminology

Add method of <code>UIElementCollection</code> class	<code>CommandBindings</code> property of <code>Window</code> control
<code>AllowsTransparency</code> property of <code>Window</code> control	<code>commands</code>
<code>Application.Current.Shutdown</code> method	<code>content control</code>
<code>attached</code> property	<code>Content</code> property of <code>ContentPresenter</code> class
<code>Background</code> property	<code>ContentControl</code> class
<code>Binding</code> class	<code>ContentPresenter</code> class
<code>Border</code> control	<code>control</code>
<code>Button</code> control	<code>control template</code>
bubbling events	<code>ControlTemplate</code> class
<code>CanExecute</code> event of <code>CommandBinding</code> class	<code>data binding</code>
<code>CanExecute</code> method of <code>ICommand</code> interface	<code>data provider</code>
<code>Canvas</code> control	<code>data template</code>
<code>Checked</code> event of <code>RadioButton</code> control	<code>DataContext</code> of WPF controls
<code>Children</code> property of <code>Panel</code> control	<code>DataTemplate</code> class
<code>Clear</code> method of <code>UIElementCollection</code> class	declarative programming
<code>Click</code> event of <code>Button</code> control	<code>dependency property</code>
collection view	direct events
<code>CollectionView</code> class	<code>DispatcherTimer</code> class
<code>CollectionViewSource.GetDefaultView</code> method	<code>DisplayMemberBinding</code> property of <code>GridView-Column</code> class
<code>Column</code> attached property of <code>Grid</code> control	<code>DockPanel</code> control
<code>ColumnDefinition</code> class associated with <code>Grid</code> control	<code>DragMove</code> method of <code>Window</code> control
<code>ColumnDefinitions</code> property of <code>Grid</code> control	dynamic resource
<code>ColumnSpan</code> attached property of <code>Grid</code> control	<code>ElementName</code> property of <code>Binding</code> class
command binding	element-to-element binding
<code>CommandBinding</code> class	<code>EventSetter</code> class
	<code>Execute</code> method of <code>ICommand</code> interface

Executed event of `CommandBinding` class  
Expander control  
Extensible Application Markup Language (XAML)  
flow-based layout  
`GetDefaultView` method of `CollectionViewSource` class  
`GetPosition` method of `MouseEventArgs` class  
Grid control  
GridView control  
GridViewColumn class  
GroupBox control  
Handled property of `RoutedEventArgs` class  
Header property of `GroupBox` control  
HorizontalAlignment property of WPF controls  
ICommand interface  
`InputGestures` property of `ICommand` interface  
Interval property of `DispatcherTimer` class  
IsAsync property of `Binding` class  
IsEnabled property of `DispatcherTimer` class  
IsMouseOver property of WPF controls  
`ItemsSource` property of `ListView`  
`ItemTemplate` property of `ListView` control  
Label control  
layout container  
Left attached property of `Canvas` control  
ListView control  
logical tree  
lookless control  
Margin property of WPF controls  
markup extension (XAML)  
MaxHeight property of WPF controls  
MaxWidth property of WPF controls  
Menu control  
MenuItem control  
MinHeight property of WPF controls  
Minimized constant of `Windows.WindowState` enum  
MinWidth property of WPF controls  
`MouseButtonEventArgs` class  
`MouseEventArgs` class  
MouseLeftButtonDown event  
MouseLeftButtonUp event  
MouseMove event  
MouseRightButtonDown event  
MouseRightButtonUp event  
naming collision  
ObjectDataProvider class  
Orientation property of `StackPanel` control  
Panel class  
Path property of `Binding` class  
presentation XAML namespace  
PreviewCanExecute event of `CommandBinding` class  
PreviewExecuted event of `CommandBinding` class  
PreviewMouseLeftButtonDown event  
PreviewMouseLeftButtonUp event  
Property property of `Trigger` class  
RadioButton control  
raster-based graphics  
resolution independence  
resource  
resource binding  
Resources property of WPF controls  
RichTextBox control  
routed events  
RoutedEventArgs class  
Row attached property of `Grid` control  
RowDefinition class associated with `Grid` control  
RowDefinitions property of `Grid` control  
RowSpan attached property of `Grid` control  
SelectedItem property of `ListView` control  
Separator control  
SetLeft method of `Canvas` control  
Setter class  
SetTop method of `Canvas` control  
Shutdown method of `Application.Current`  
Slider control  
Source property of `Binding` class  
Source property of `RoutedEventArgs` class  
StackPanel control  
standard XAML namespace  
StartupUri property of `App.xaml`  
static resource  
style  
Style class  
style sheet  
TargetName property of `Setter` class  
TargetType property of `ControlTemplate` class  
TargetType property of `Style` class  
template binding  
Template property of WPF controls  
TextBlock control  
TextBox control  
Tick event of `DispatcherTimer` class  
TimeSpan struct  
ToolBar control  
ToolBarTray control

Top attached property of Canvas control	WindowState.Minimized constant
trigger	WindowState property of Window control
Trigger class	WindowStyle property of Window control
Triggers property of ControlTemplate class	WPF (Windows Presentation Foundation)
Triggers property of Style class	WrapPanel control
tunneling events	x:Class attribute
UIElementCollection class	x:Name attribute
Value property of Trigger class	XAML (Extensible Application Markup Language)
vector-based graphics	XAML view
VerticalAlignment property of WPF controls	XmlDataProvider class
visual tree	XmlReader class
Window control	ZIndex attached property of Canvas control
Windows Presentation Foundation	

## Self-Review Exercises

- 25.1** State whether each of the following is *true* or *false*. If *false*, explain why.
- WPF has GUI, animation, 2D graphics, 3D graphics and multimedia capabilities.
  - All of a WPF app's Visual C# code can be replaced by XAML markup.
  - You lay out a WPF GUI in the same way you lay out a Windows Forms GUI.
  - Events in WPF are the same as they are in Windows Forms.
  - A WPF command can be executed through many user interactions.
  - If a WPF resource is defined for a Window, then it can be used by any of the Window's child elements.
  - A WPF style must be defined for a specific control type.
  - A WPF control template must be defined for a specific control type.
  - A WPF control's logical and visual trees are the same.
  - Data bindings in WPF can be specified in XAML by markup extensions.
- 25.2** Fill in the blanks in each of the following statements:
- XAML documents consists of a hierarchy of \_\_\_\_\_.
  - A Window is a(n) \_\_\_\_\_, meaning it can have exactly one child element or text.
  - Properties such as Grid.Row or Canvas.Left are examples of \_\_\_\_\_ properties.
  - MouseLeftButtonDown is a bubbling event. The name of its corresponding tunneling event is \_\_\_\_\_.
  - When a WPF command is executed, it raises the \_\_\_\_\_ and \_\_\_\_\_ events.
  - A menu usually has only \_\_\_\_\_ and \_\_\_\_\_ as children.
  - \_\_\_\_\_ resources are applied at initialization only, whereas \_\_\_\_\_ resources are reapplied every time the resource is modified.
  - A WPF control template defines a control's \_\_\_\_\_ tree.
  - To format the display of items in a ListView, you would use a(n) \_\_\_\_\_.

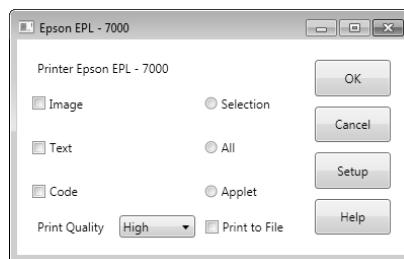
## Answers to Self-Review Exercises

- 25.1** a) True. b) False. XAML is designed to be used alongside Visual C#, not to replace it. c) False. The layout in WPF is primarily flow based, whereas layout in Windows Forms is primarily coordinate based. d) False. Events in WPF are routed events. e) True. f) True. g) False. A style may be applied to any type of control. h) True. i) False. A control's visual tree represents how a control is rendered, whereas a logical tree represents how it's defined. j) True.
- 25.2** a) elements. b) content control. c) attached. d) PreviewMouseLeftButtonDown. e) Executed, PreviewExecuted. f) MenuItems, Separators. g) Static, dynamic. h) visual. i) data template.

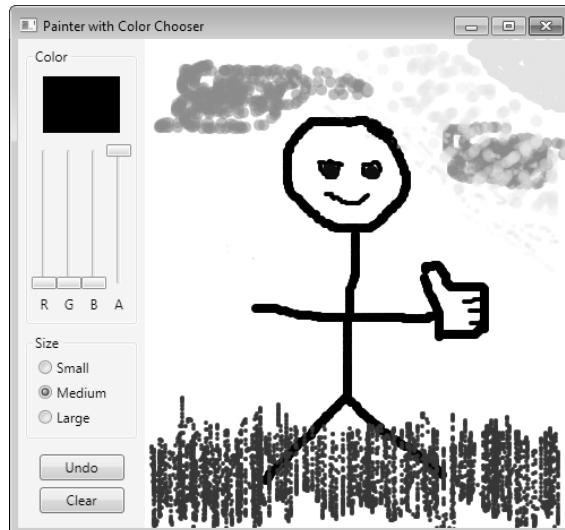
## Exercises

**25.3** Create the GUI in Fig. 25.29 (you do not have to provide functionality) using WPF. Do not use a Canvas. Do not use explicit sizing or positioning.

**25.4** Incorporate an RGBA color chooser into the Painter example to look like Fig. 25.30. Let the user select the brush color using the color chooser instead of the group of RadioButtons. You should use a style to make all the sliders look the same.



**Fig. 25.29** | Printer GUI.



**Fig. 25.30** | Painter with color-chooser GUI.

**25.5** Use WPF to create a cash-register app modeled after the one presented in Fig. 25.31. It should allow users to enter a series of prices, then obtain the total. The **Delete** button should clear the current entry, and the **Clear** button should reset the app.

**25.6** Use WPF to create an app that quizzes users on their knowledge of national flags. The app should display a series of flags in random order (with no repeats). As each flag is displayed, the user should be able to select the flag's country from a drop-down list and submit an answer (Fig. 25.32). The app should keep a running tally of how the user has performed. The flag images are available in the `exerciseImages` folder.

Because the flag `Image` needs to change from country to country, its `Source` property needs to be set in Visual C# code. In XAML, you can set a `string` file path to be an `Image`'s `Source`, but the `Source` property actually takes an `ImageSource` object as its value. When your XAML document is compiled, the `string` file path is converted. To set an `Image`'s `Source` in Visual C#, you must specify an `ImageSource` object. To create an `ImageSource` object from a `string` file path, you can write

```
ImageSourceConverter converter = new ImageSourceConverter();
ImageSource source = ( ImageSource ) converter.ConvertFromString( path );
```

where `path` is the `string` file path.



**Fig. 25.31** | Cash-register GUI.



**Fig. 25.32** | Flag-quiz GUI.

**25.7** Using WPF, create a GUI that represents a simple microwave, as shown in Fig. 25.33 (you do not have to provide functionality). To create the `Start`, `Clear` and numerical `Buttons`, you'll need to make use of control templates. To apply a control template automatically for a control type, you can create a style (with a `TargetType`) that sets the `Template` property.

**25.8** WPF allows two-way data bindings. In a normal data binding, if the data source is updated, the binding's target will update, but not vice versa. In a two-way binding, if the value is changed in either the binding's source or its target, the other will be automatically updated. To create a two-way binding, set the `Mode` property to `TwoWay` at the `Binding`'s declaration. Create a phone-book app modeled after the one shown in Fig. 25.34. When the user selects a contact from the contacts list, its information should display in a `Grid` of `TextBoxes`. As the information is modified, the contacts list should display each change.



Fig. 25.33 | Microwave GUI.

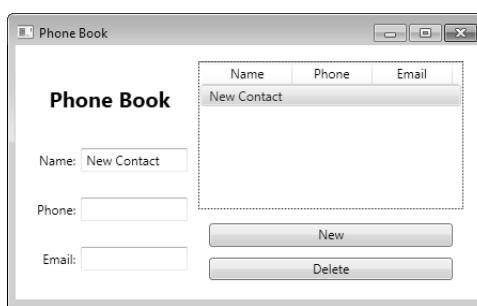


Fig. 25.34 | Phone-book GUI.

**25.9** You can create data bindings to LINQ queries. Modify the example given in Fig. 9.4 to display a list of all employees' earnings and a list of employees earning between \$4000 and \$6000 per month (Fig. 25.35). Allow users to set a different salary range and update the filtered list of employees. Use a data template to display each employee in the format *LastName, FirstName: \$Per-Month Salary*.

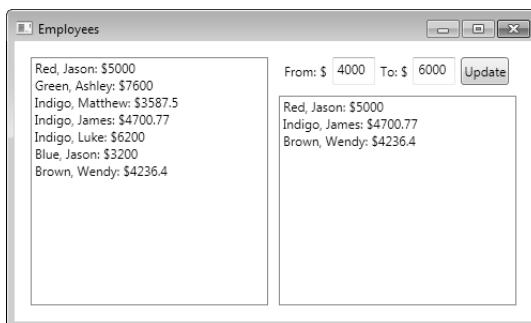


Fig. 25.35 | Lists-of-employees GUI.