25

Windows 8 UI and XAML

Objectives

In this chapter you'll:

- Define Windows Store app GUIs with Windows 8 UI and Extensible Application Markup Language (XAML).
- Handle Windows 8 UI user-interface events.
- Customize the look-and-feel of Windows 8 GUIs using styles.
- Use data binding to display data in Windows 8 UI controls.
- Be introduced to the Windows Store app lifecycle.

Outline

25.1 Introduction

[Note: *Developing and running the apps in this chapter and Chapter 26 requires Windows 8 and Visual Studio Express 2012 for Windows 8 (or a full version of Visual Studio 2012).*]

There are three technologies for building GUIs in Windows—Windows `Forms`, Windows 8 UI and Windows Presentation Foundation (WPF). In Chapters 14–15, you built GUIs using Windows `Forms`, which is now considered to be a legacy technology. In this chapter, you'll build GUIs using **Windows 8 UI**—Microsoft's Windows 8 framework for GUI, graphics, animation and multimedia. In Chapter 26, you'll learn how to incorporate graphics, animation, audio and video in Windows 8 UI apps. Windows Presentation Foundation (WPF; Chapters 32–33) was introduced in .NET 3.0 as the replacement for Windows `Forms` and various graphics and multimedia capabilities. Many of the concepts you'll learn in this chapter originated in WPF.

Windows Store

Apps that use the Windows 8 UI are known as *Windows Store apps* and can be submitted to the Windows Store (`http://www.windowsstore.com/`) as free or for-sale apps. Before an app can appear in the store, it must be *certified*—you can learn more about this at

```
http://msdn.microsoft.com/library/windows/apps/bg125379
```

Extensible Application Markup Language (XAML)

Throughout this chapter, we discuss an important tool for creating Windows Store apps called **XAML** (pronounced "zammel")—**Extensible Application Markup Language**. XAML is an XML vocabulary that can be used to define and arrange 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 introduced XML in Sections 24.2–24.4.

Building a GUI with Windows 8 UI is similar to building a GUI with Windows `Forms`—you drag-and-drop predefined controls from the **Toolbox** onto the *design area*.

Many Windows 8 UI 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 Windows 8 UI. A Windows 8 UI Button, for example, functions like a Windows Forms Button, and both raise Click events.

There are several important differences between the two technologies. The Windows 8 UI layout scheme is different. Windows 8 UI properties and events have more capabilities. Most notably, Windows 8 UI allows designers to define the appearance and content of a GUI in XAML without any C# code.

As you build a Windows Store app's GUI visually, the IDE generates XAML. This markup is designed to be readable by both humans and computers, so you can also manually write XAML to define your GUI, and in many cases you will. When you compile your 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 is included with full versions of Visual Studio 2012. 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.

Windows 8.1

At time of this writing, Windows 8.1 and Visual Studio 2013 were available as developer previews—even though Windows 8 and Visual Studio 2012 were released only 8 months earlier. Windows 8.1 enhances the Windows 8 operating system and addresses various concerns expressed by users regarding Windows 8, such as the lack of a **Start** button and the requirement to boot to the Windows 8 **Start** screen rather than to the Windows 8 desktop. At Microsoft's 2013 Build Conference (www.buildwindows.com), Microsoft CEO Steve Balmer announced Microsoft's plans for much more frequent software releases, which is now typical of companies (e.g., Apple, Google and many others). To develop this chapter, we used Windows 8 and Visual Studio Express 2012 for Windows 8.

25.2 Welcome App: Introduction to XAML Declarative GUI Programming

A XAML document defines the appearance of a Windows 8 UI app. Figure 25.1 shows a simple XAML document that defines a window that displays Welcome to Windows Store App Development in a **TextBlock**, which is similar to a Label in Windows Forms. We reformatted the XAML generated by the IDE for publication purposes. We discuss the XAML details and how to build this app in following subsections.

```
1  <!-- Fig. 25.1: MainPage.xaml -->
2  <!-- A simple XAML document. -->
3
```

Fig. 25.1 | A simple XAML document. (Part 1 of 2.)

```
<!-- Page control is the root element of the GUI -->
<Page
   x:Class="Welcome.MainPage"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:local="using:Welcome"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   mc:Ignorable="d">

   <!-- Grid is a layout container -->
   <Grid
      Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

      <!-- TextBlock displays text -->
      <TextBlock HorizontalAlignment="Center" Margin="10"
         TextWrapping="Wrap" FontSize="96" TextAlignment="Center"
         Text="Welcome to Windows Store App Development"
         VerticalAlignment="Center" FontFamily="Global User Interface" />
   </Grid>
</Page>
```

Fig. 25.1 | A simple XAML document. (Part 2 of 2.)

25.2.1 Running the Welcome App

You can test your Windows Store apps three ways by using the drop-down list to the right of **Start Debugging** (Fig. 25.2) on the Visual Studio toolbar:

- **Simulator**—Runs the app in a window that simulates a Windows 8 desktop or tablet computer. The simulator allows you to test capabilities such as rotating a device (orientation), touch input, different screen sizes/resolutions and geolocation (e.g., for maps and other location-aware apps). You can also use the simulator to take screen captures of your app for use in the Windows Store.

- **Local Machine** (the default)—Runs the app directly on the Windows 8 computer that's running Visual Studio.
- **Remote Machine**—Runs the app on a Windows 8 device that's connected to the Windows 8 computer that's running Visual Studio via a network or cable. Windows 8 tablets have various sensors (e.g, GPS, gyroscope, etc.) that are not available on most desktop and laptop computers. Running an app on a remote machine allows you to test these capabilities of your app.

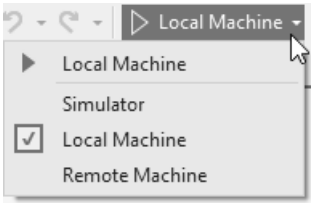

Fig. 25.2 | **Start Debugging** drop-down list on the Visual Studio toolbar.

The screen capture in Fig. 25.1 shows the app running in the Windows 8 simulator. To run the **Welcome** app:

1. Open the `Welcome.sln` file from this chapter's examples folder to load the project into Visual Studio Express 2012 for Windows 8 (or your full version of Visual Studio 2012).
2. Select from the **Start Debugging** drop-down list either **Simulator** or **Local Machine**—whichever item you select is displayed on the toolbar.
3. Click the **Start Debugging** button on the toolbar (or press *F5*) to run the app.

25.2.2 `MainPage.xaml` for the Welcome App

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*. Windows 8 UI **controls** are represented by elements in XAML markup. The root element of the XAML document in Fig. 25.1 is a **Page control** (lines 5–24), which defines the content of the app's page. By default, Windows Store apps use the entire screen. In Windows 8, it's possible to have a maximum of two Windows Store apps on the screen at once. One occupies a column that's 320 pixels wide and as tall as the screen, and the other app occupies the rest. With Windows 8.1, apps must still occupy the screen's height, but they can now be *any* width (the default minimum width is 500 pixels) and there can be more that two apps on the screen at a time.

Page Element

A `Page` represents a screen of content and controls in a Windows Store app. The attribute **`x:Class`** (line 6) of the `Page` start tag (lines 5–12) specifies the name of the class that provides the GUI's functionality. The `x:` signifies that the `Class` attribute is part of the XAML XML vocabulary (discussed in more detail momentarily). A XAML document must have an associated *code-behind file* to handle events. Together, the markup in `Main-`

`Page.xaml` and its code-behind file represent a derived class of `Page` that defines the `Page`'s GUI and functionality.

`Page` (from namespace **`Windows.UI.XAML.Controls`**) is a **content control** that may contain only one **layout container**, which arranges other controls in the `Page`. A layout container such as a `Grid` (lines 15–23) can have many *child elements*, including *nested layout containers*, allowing it to arrange many controls in your GUI.

XAML Namespaces

Several XML namespaces (lines 7–11, Fig. 25.1) are automatically defined as attributes of the `Page` element. This enables the XAML compiler to interpret your markup:

- `xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"`
  This default namespace (line 7) defines Windows 8 UI elements and attributes.
- `xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`
  Defines XAML language elements and attributes. This namespace is mapped to the prefix `x` (line 8).
- `xmlns:local="using:Welcome"`
  This namespace give you access in XAML to parts of the `Page` subclass that represents `MainPage.xaml`. This namespace is mapped to the prefix `local` (line 9).
- `xmlns:d="http://schemas.microsoft.com/expression/blend/2008"`
  This namespace defines elements and attributes that are used by the IDE's GUI designer. This namespace is mapped to the prefix `d` (line 10). Any element or attribute prefixed with `d:` is ignored when the XAML is loaded at execution time.
- `xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"`
  This namespace, which is mapped to the prefix `mc` (line 11), tells the XAML parser which namespaces can be ignored. Line 11 indicates that elements prefixed with `d` can be ignored.

`Grid` Element

A **`Grid`** (from namespace `Windows.UI.XAML.Controls`) is a *flexible, all-purpose layout container* that organizes controls into rows and columns (one row and one column by default). **Layout containers** (discussed in more detail in Section 25.3.1) help you arrange a GUI's controls. Lines 15–23 define a one-cell grid. The **`Background`** attribute specifies the `Grid`'s background color. By default, the IDE sets this `Grid` element's `Background` to a predefined style named `ApplicationPageBackgroundThemeBrush`. Section 25.3.2 discusses using *predefined styles* in detail. In XAML elements, attributes like `Background` that are *not* qualified with an XML namespace (such as `x:`) correspond to *properties* of an object.

`TextBlock` Element

A **`TextBlock`** (from namespace `Windows.UI.XAML.Controls`; lines 19–22)—similar to the `Label` control in Windows `Forms`—displays text in a GUI. We specified several `TextBlock` attributes:

- **`HorizontalAlignment`**—How the `TextBlock` aligns horizontally within the layout container. The options are `Left`, `Center`, `Right` and `Stretch` (which stretches the element to *fill all available horizontal space*; this is the default).

- **`Margin`**—The number of pixels separating the control from other controls (or the edge of the screen). If you specify one value, it applies to the left, top, right and bottom edges of the control. If you specify two values—as in `"10, 20"`—the first value applies to the left and right edges, and the second applies to the top and bottom edges. If you specify four values—as in `"10, 20, 30, 40"`—the values apply to the left, top, right and bottom, respectively.
- **`TextWrapping`**—Whether the text wraps to multiple lines if it cannot fit within the `TextBlock`'s width. The options are `Wrap` or `NoWrap` (the default).
- **`FontSize`**—The font size measured in pixels. The default size is 11.
- **`Text`**—The text displayed by the `TextBlock`.
- **`TextAlignment`**—How the text is aligned within the `TextBlock`. The options are `Left` (the default), `Center`, `Right` and `Justify` (i.e., aligned both left and right).
- **`VerticalAlignment`**—How the `TextBlock` is aligned vertically within the layout container. The options are `Top`, `Center`, `Bottom` and `Stretch` (which stretches the element to *fill all available vertical space*; this is the default).
- **`FontFamily`**—Specifies the text's font. We used the default font that was set by the IDE.

These attributes correspond to properties of the `TextBlock` object that's created by the XAML markup in lines 19–22.

25.2.3 Creating the Welcome App's Project

To create a new Windows Store app using Windows 8 UI, open the **New Project** dialog (Fig. 25.3) and select **Templates > Visual C# > Windows Store** from the list of templates.

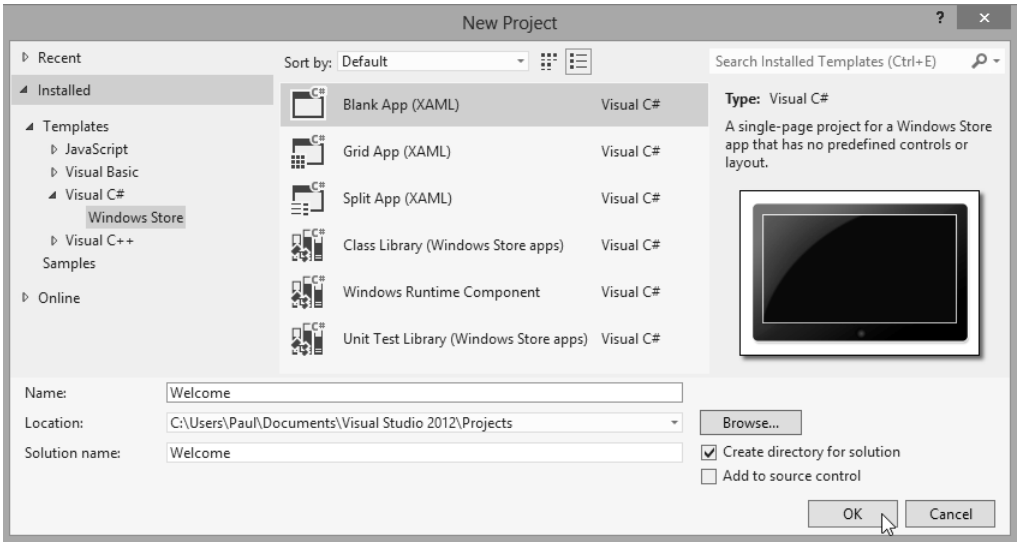

Fig. 25.3 | **New Project** dialog.

You can choose from several project templates, including the ones in Fig. 25.4. For this app, we chose the **Blank App** template. When the project opens in the IDE, the `App.xaml.cs` file is displayed by default. You can close that file for now (we'll discuss it later).

Template	Description
Blank App	A single-page (that is, one screen) app for which you must define your own GUI.
Grid App	A three-page app with predefined layouts and GUI controls. The provided pages allow the user to view a collection of groups of items (e.g., a collection of photo albums) displayed in a grid of rows and columns, view a particular group's details (e.g., a description of one photo album along with a preview of the albums photos) and view a particular item's details (e.g, one photo from a particular photo album). This template also provides support for switching between pages (known as *navigation*) and more.
Split App	A two-page app with predefined layouts and GUI controls. The first page allows the user to view a collection of groups of items displayed in a grid. The second page allows the user to view a particular group *and* a selected item's details. This template also provides support for navigating between pages and more.

Fig. 25.4 | Windows Store app project templates.

Windows Store App Themes

There are two predefined *themes* for Windows Store apps—`Dark` (the default) and `Light`. Microsoft recommends using the `Dark` theme for apps that present mostly images and video and the `Light` theme for apps that display lots of text. In this app, we're displaying only text, so we'll use the `Light` theme. To do so, double click `App.xaml` in the **Solution Explorer**, then add

```
RequestedTheme="Light"
```

to the `application` element's opening tag, then save and close the file.

Design and XAML Views

Double click `MainPage.xaml` in the **Solution Explorer** to open the XAML editor (Fig. 25.5), which has **Design** and **XAML** views. The **Design** view is used for drag-and-drop GUI design and the **XAML** view shows the generated XAML markup. You can also edit the XAML directly (as you'll often do) in the XAML view. The IDE syncs these views and the **Properties** window—when you edit content in the **Design** view, the **XAML** view or the **Properties** window, the IDE automatically updates the others. The bar between the views contains buttons that enable you to show the two views side-by-side (⧈), show the two views one above the other (⊟), swap the views (↑↓) and collapse whichever view is on the bottom or right side (⊠).

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

Fig. 25.5 | **Design** and **XAML** views.

is four). Select **Tools > Options** to display the **Options** dialog. In **Text Editor > XAML > Tabs** change the **Tab size** and the **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**.

25.2.4 Building the App's GUI

Creating a Windows Store app is similar to creating a Windows `Forms` app. You can drag-and-drop controls onto the **Design** view. You can also drag them into the **XAML** view. A control's properties can be edited in the **Properties** window or in the **XAML** view.

Because XAML is easy to understand and edit, it's often less difficult to manually edit your XAML markup. In some cases, you *must* manually write XAML markup. Nevertheless, the visual programming tools in Visual Studio are often handy, and we'll point out the situations in which they're useful.

Creating the `TextBlock` Element and Configuring Its Properties

To add the `TextBlock` element, drag a `TextBlock` from the **Toolbox** and drop it in the **XAML** view between the `Grid` element's start and end tags, then click the `TextBlock` element in the **XAML** view to select it. At the top of the **Properties** window for XAML documents (Fig. 25.6), you can specify the variable name of a XAML element. Below that, the window displays the type of the currently selected object in the **Design** or **XAML** view. By default, the selected object's properties are arranged by category. Within each category, the most commonly used properties are displayed. If a down arrow (⌄) is displayed you can click it to display more of that category's properties.

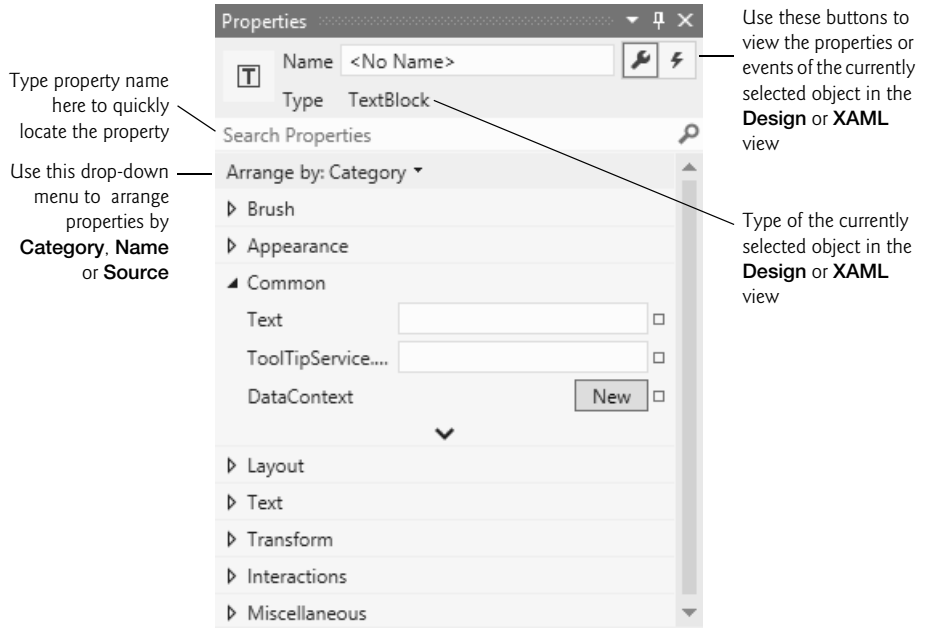

Fig. 25.6 | **Properties** window for XAML documents.

Use the **Properties** window to set the `TextBlock`'s properties to the values in lines 19–22 of Fig. 25.1—as you do, experiment with displaying the properties by **Category** and by **Name**, and use the **Search Properties** box to quickly locate a given property. Below is a list of the properties to set, the category they're located in and the value to specify (we used the default `FontFamily` set by the IDE, so you do not need to change that property):

- `Text` (**Common** category)—`"Welcome to Windows Store App Development"`
- `HorizontalAlignment` (**Layout** category)—`"Center"` (click the ☰ button)
- `VerticalAlignment` (**Layout** category)—`"Center"` (click the ⫼ button)
- `Margin` (**Layout** category)—`"10"` (specify 10 for all four margin values)
- `FontSize` (**Text** category)—`"96"`
- `TextWrapping` (**Text** category)—`"Wrap"`
- `TextAlignment` (**Text** category)—`"Center"`

25.2.5 Overview of Important Project Files and Folders

When you create a Windows Store app, several files and folders are generated and can be viewed in the **Solution Explorer**.

- **`App.xaml`**—Defines the `Application` object and its settings. The most noteworthy element is `Application.Resources`, which specifies resources, such as common styles that define the GUI's look-and-feel, that should be available to all

pages of your app. By default, `StandardStyle.xaml` (from the project's `Common` folder) is specified as a resource.

- **`App.xaml.cs`**—The *code-behind file* for `App.xaml` specifies app-level event handlers, such as the app's `OnLaunched` event handler, which ensures that the app's `MainPage` is displayed when the app starts executing. 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.
- **`MainPage.xaml`**—Defines the first `Page` that's loaded in the app. `MainPage.xaml.cs` is its *code-behind file*, which handles the `Page`'s events.
- **`StandardStyles.xaml`** (in the `Common` folder)—Defines styles (discussed in Section 25.3.2) that enable your app's GUI to have the look-and-feel of Windows Store apps.
- **`Assets`** folder—Typically contains media files, such as images, audio and video files. By default this folder contains four images that are used as logos and a splash screen. We discuss these in Section 25.2.6.
- **`Common`** folder—Contains files typically used by all `Pages` in an app, such as the `StandardStyles.xaml` file.
- **`Package.appxmanifest`**—Defines various settings for your app, such as the app's display name, logo, splash screen and much more. We discuss this file in Section 25.2.7.

25.2.6 Splash Screen and Logos

When you executed this app, you might have noticed that the Deitel logo was displayed briefly before the app appeared on the screen. This was the app's **splash screen**, which provides a visual indication that the app is loading. The splash screen image is one of four PNG images that are provided by default in the project's `Assets` folder. These images are described in Fig. 25.7. You can use any image editing tool to edit the default image files provided by the IDE to incorporate your own logo image. We provide the same versions of these four images in all of this chapter's apps. For more details on these and other standard images used in Windows Store apps, see

```
msdn.microsoft.com/en-us/library/windows/apps/hh846296.aspx
```

Image	Description
`Logo.png`	Shown on the app's tile in the **Start** screen.
`SmallLogo.png`	Displayed with your app's name when your app is shown in Windows 8 search results.
`SplashScreen.png`	Displayed briefly when your app first loads.
`StoreLogo.png`	If you post an app to the Windows Store, this image is shown with your app's description and in any Windows Store search results.

Fig. 25.7 | Images provided by default in a Windows Store app project's `Assets` folder.

25.2.7 Package.appmanifest

The app's manifest file `Package.appmanifest` enables you to quickly configure various app settings, such as the app's display name, its logo images and splash screen, the hardware and software capabilities it uses, information for the Windows Store and more. Complete details of the manifest are discussed at

```
msdn.microsoft.com/en-us/library/windows/apps/
   br230259(v=vs.120).aspx
```

Double clicking the `Package.appmanifest` file in the **Solution Explorer** displays the manifest editor. For this app, we made two changes in the **Application UI** tab's **Tile** section, which specifies information for the app's **Start** screen tile. We set the **Foreground text** value to `Dark` and changed the **Background color** to `#FFFFFF`. This causes Windows 8 to display the app's logo and app name on a tile with a white background and the app name in dark text so that it's readable against the white background. By default, the app's tile is displayed with a dark gray background and light text.

25.3 Painter App: Layouts; Event Handling

This section presents our **Painter** app (Fig. 25.8), which allows you to draw by using the mouse, a stylus (a pen like device for touch screens) or your fingers, depending on the type of Windows 8 device that you have. The app draws colored circles in response to your interactions with the drawing area. We use this app to demonstrate how to lay out controls and handle events in Windows Store apps.

Fig. 25.8 | **Painter** app with a completed drawing running in the Windows 8 simulator.

For this app, we used the **Blank App** template with the default `Dark` theme. In the `Package.appmanifest` file, in addition to the changes we discussed for the **Welcome** app

in Section 25.2.7, we also set the **Supported rotations** to **Landscape** and **Landscape-flipped** so that this app can be used only in landscape orientation on tablet devices. Depending on your app's purpose, you may choose to also support **Portrait** and **Portrait-flipped**. For example, apps in which the user reads lots of text are often better in portrait orientation because it's easier to read short lines of text—this is why books and e-readers are designed for portrait orientation.

25.3.1 General Layout Principles

In Windows `Forms`, a control's size and location are specified explicitly. In Windows 8, apps can execute on a variety of desktop, laptop and tablet computers. Tablets in particular can be held in *portrait* (longer side vertical) or *landscape* (longer side horizontal) orientations, so a Windows Store app's user interface should be able to dynamically adjust based on the device orientation (which can change frequently). For this reason, a control's size is often specified as a range of 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 the app's width and height on the screen. Likewise, it enables your GUIs to be *resolution independent*.

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's width and height changes* or the app's *content updates*. Thus, in addition to the `Width` and `Height` properties associated with every control, all Windows 8 UI controls have the **`MinWidth`**, **`MinHeight`**, **`MaxHeight`** and **`MaxWidth`** properties. If the `Width` and `Height` properties are both `Auto` (which is the *default* when they're *not* specified in the XAML code), you can use these minimum and maximum properties to specify a range of acceptable sizes for a control. Its size will *automatically adjust* as the *size of its container changes*.

Position of a Control

A control's position should be specified relative to the *layout container* in which it's included and the other controls in the same container. All controls have three properties for doing this—**`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, which it interprets as the left–right and top–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*. For example, if a parent layout container is 100 pixels high and a child control's `VerticalAlignment` is set to `Stretch`, then the control will fill the entire 100 pixel height of the layout container.

Layout Containers
Windows 8 UI provides many layout containers. Figure 25.9 lists three of the simpler layout controls that we use in this chapter. These (and many others) are derived from class `Panel` in the `Windows.UI.Xaml.Control` namespace. A control can have other layout properties specific to the *layout container* in which it's contained. We'll discuss these as we examine specific layout containers.

Layout	Description
`Grid`	Layout is defined by a *grid of rows and columns*, depending on the `RowDefinitions` and `ColumnDefinitions` properties. Elements are placed into cells.
`Canvas`	Layout is *coordinate based*. Element positions are defined explicitly by their distance from the `Canvas`'s top and left edges.
`StackPanel`	Elements are arranged in a *single row or column*, depending on the `Orientation` property.

Fig. 25.9 | Common layout containers.

25.3.2 `MainPage.xaml`: Layouts and Controls in the Painter App

Figure 25.10 shows the `MainPage.xaml` document and the GUI display of our **Painter** app. This example uses Windows 8 UI `Button`s and `RadioButton`s, and several layout controls. This section focuses on the XAML and Section 25.3.3 discusses the *code-behind file* that defines the event handlers.

```
1  <!-- Fig. 25.10: MainPage.xaml -->
2  <!-- XAML of the Painter app. -->
3  <Page
4     x:Class="Painter.MainPage"
5     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7     xmlns:local="using:Painter"
8     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9     xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10    mc:Ignorable="d">
11
12    <!-- Defines the AppBar that appears at the bottom of the screen -->
13    <Page.BottomAppBar>
14       <!-- AppBar contains layouts and controls -->
15       <AppBar FontFamily="Global User Interface" IsOpen="True">
16          <!-- AppBar must have one layout element -->
17          <Grid>
18             <!-- StackPanel for left side of AppBar -->
19             <StackPanel Orientation="Horizontal"
20                HorizontalAlignment="Left" VerticalAlignment="Center">
21
22                <!-- StackPanel to group Color RadioButtons -->
23                <StackPanel Orientation="Horizontal" Margin="20,0,20,0">
```

Fig. 25.10 | XAML of the **Painter** app. (Part 1 of 3.)

```
24                      <TextBlock Text="Color:" FontSize="20"
25                         VerticalAlignment="Center"/>
26                      <RadioButton x:Name="blackRadioButton" Content="Black"
27                         GroupName="color" IsChecked="True"
28                         Checked="blackRadioButton_Checked"/>
29                      <RadioButton x:Name="redRadioButton" Content="Red"
30                         GroupName="color" Checked="redRadioButton_Checked"/>
31                      <RadioButton x:Name="blueRadioButton" Content="Blue"
32                         GroupName="color" Checked="blueRadioButton_Checked"/>
33                      <RadioButton x:Name="greenRadioButton" Content="Green"
34                         GroupName="color" Checked="greenRadioButton_Checked"/>
35                   </StackPanel>
36
37                   <!-- StackPanel to group Brush Size RadioButtons -->
38                   <StackPanel Orientation="Horizontal">
39                      <TextBlock Text="Brush Size:" FontSize="20"
40                         VerticalAlignment="Center"/>
41                      <RadioButton x:Name="smallRadioButton" Content="Small"
42                         GroupName="size" IsChecked="True"
43                         Checked="smallRadioButton_Checked"/>
44                      <RadioButton x:Name="mediumRadioButton"
45                         Content="Medium" GroupName="size"
46                         Checked="mediumRadioButton_Checked"/>
47                      <RadioButton x:Name="largeRadioButton" Content="Large"
48                         GroupName="size" Checked="largeRadioButton_Checked"/>
49                   </StackPanel>
50                </StackPanel>
51
52                <!-- StackPanel for right side of AppBar -->
53                <StackPanel Orientation="Horizontal"
54                   HorizontalAlignment="Right">
55                   <Button x:Name="undoButton" Click="undoButton_Click"
56                      Style="{StaticResource UndoAppBarButtonStyle}" />
57                   <Button x:Name="clearButton" Click="clearButton_Click"
58                      Style="{StaticResource DeleteAppBarButtonStyle}" />
59                </StackPanel>
60             </Grid>
61          </AppBar>
62       </Page.BottomAppBar>
63
64       <!-- Grid layout for the Page's conent -->
65       <Grid
66          Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
67          <!-- Canvas for displaying graphics -->
68          <Canvas x:Name="paintCanvas" Margin="10" Background="White"
69             PointerPressed="paintCanvas_PointerPressed"
70             PointerMoved="paintCanvas_PointerMoved"
71             PointerReleased="paintCanvas_PointerReleased"
72             PointerCanceled="paintCanvas_PointerCanceled"
73             PointerCaptureLost="paintCanvas_PointerCaptureLost"
74             PointerExited="paintCanvas_PointerExited"/>
75       </Grid>
76    </Page>
```

Fig. 25.10 | XAML of the **Painter** app. (Part 2 of 3.)

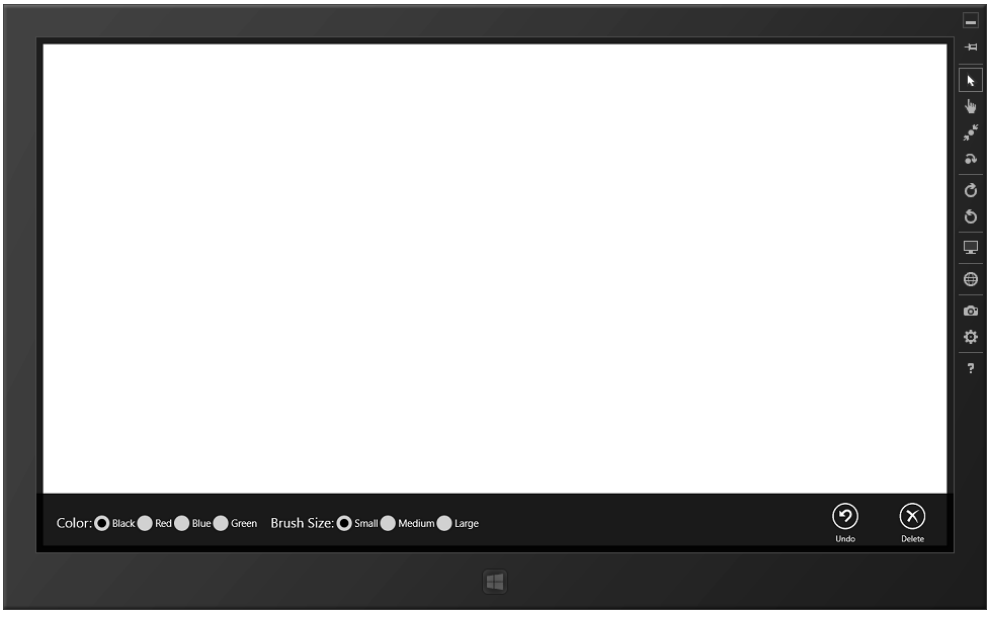

Fig. 25.10 | XAML of the **Painter** app. (Part 3 of 3.)

Bottom `AppBar`

At the bottom of the **Painter** app is an **`AppBar`** (lines 13–62) containing the controls that you use to interact with the **Painter** app. Each `Page` can have `AppBar`s at the top and bottom of the screen for quick access to commands. By default, the `AppBar`s are hidden. You can show them by *swiping from the screen's bottom edge* on a touch screen or by *right clicking* with the mouse on a desktop computer. They're dismissed by interacting with the app's content—in this app's case, by interacting with the drawing area. In this app, we set the `AppBar`'s **`IsOpen`** attribute to `True` to indicate that it should be displayed when the app first executes.

Creating an `AppBar`

You create an `AppBar` by clicking the `Page` element in the XAML view, then clicking the **New** button to the right of the **`BottomAppBar`** or **`TopAppBar`** property in the **Properties** window. This creates a `Page.BottomAppBar` or `Page.TopAppBar` element containing an `AppBar` control. You can then drag layouts and controls into the `AppBar`'s area in **Design** view or edit the `AppBar` element's XAML directly in **XAML** view.

Laying Out the `AppBar`

To lay out the `AppBar`'s controls, we added a `Grid` layout to the `AppBar`, then added *nested* `StackPanel` layouts within the `Grid`—in this case, we edited the XAML directly. A **`StackPanel`** arranges its content either *vertically* or *horizontally*, depending on its **`Orientation`** value. The default is `Vertical`, but `AppBar`s are arranged horizontally, so we used `Horizontal` for each `StackPanel` (as specified in lines 19, 23, 38 and 53).

The `StackPanel` at lines 19–50 arranges the controls at the `AppBar`'s left side (Fig. 25.11). We set the `StackPanel`'s `HorizontalAlignment` property to `Left` to left align

the controls on the `AppBar`'s left side, and set its `VerticalAlignment` property to `Center`, which centers the controls vertically within the `AppBar`.

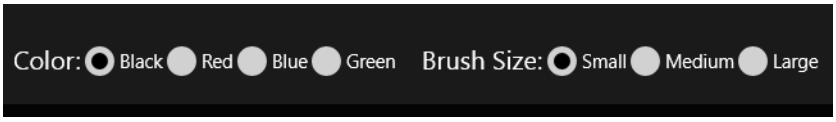

Fig. 25.11 | Left side of the **Painter** app's `AppBar`.

The `StackPanel` at lines 53–59 (of Fig. 25.10) arranges the controls at the `AppBar`'s right side (Fig. 25.12). We set the `StackPanel`'s `HorizontalAlignment` property to `Right`, which right aligns the controls on the `AppBar`'s right side.

Fig. 25.12 | Right side of the **Painter** app's `AppBar`.

`RadioButton`s

Windows 8 UI **`RadioButton`s** function as *mutually exclusive options*, just like their Windows `Forms` counterparts. However, a Windows 8 UI `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—for example, it can be labeled by an image or other item. In this app, each `RadioButton` is labeled by plain text that's specified with the control's **`Content` property** (for example, line 26 of Fig. 25.10).

Unlike Windows `Forms`, Windows 8 UI does *not* provide a `GroupBox` control for maintaining the mutually exclusive relationship between `RadioButton`s. Instead, this relationship is based on the `GroupName` property—`RadioButton`s with the same `GroupName` are mutually exclusive. Many layouts, including `StackPanel`, are subclasses of `Panel`. The `StackPanel` at lines 23–35 contains a `TextBlock` and `RadioButton`s for choosing the *drawing color*. The one at lines 38–49 contains a `TextBlock` and `RadioButton`s for choosing the *brush size*. In addition to the `GroupName`, we set the following properties for each `RadioButton` in this example:

- **`x:Name`** (set by the `Name` property at the top of the **Properties** window)—Specifies the variable name used in C# code to interact with the control programmatically.
- **`Content`** (**Common** category)—Specifies the text to display (to the right of the `RadioButton`, by default).
- **`Checked`**—Specifies the event handler that's called when the user interacts with the control. You can create the event handler either by double clicking the control in **Design** view or by using the **Event handlers** tab (⚡) in the **Properties** window.

For the **Black** and **Small** `RadioButton`s we also set the **`IsChecked` property** to `True` to indicate that those `RadioButton`s should be *selected* when the app begins executing.

Buttons

A Windows 8 UI **Button** behaves like a Windows Forms Button but is a ContentControl. As such, *a Windows 8 UI Button can display any single element as its content, not just text.* Lines 53–59 define a StackPanel containing two Buttons—an Undo button that removes the last circle that was drawn and a Delete button that removes the entire drawing. For each Button in this example, we set the following properties:

- **x:Name** (set by the Name property at the top of the **Properties** window)—Specifies the variable name used in C# code to interact with the control programmatically.
- **Click**—Specifies the event handler that's called when the user interacts with the control. You can create the event handler either by double clicking the control in **Design** view or by using the **Event handlers** tab (⚡) in the **Properties** window.
- **Style**—Specifies the look-and-feel of the Button, which we discuss below.

Styling the Buttons

Previously, we mentioned that each Windows Store app project also has a StandardStyles.xaml file (in the project's Common folder) that defines the common look-and-feel of Windows Store apps and their controls. There are many pre-configured AppBar button styles provided in StandardStyles.xaml. To use a style for a Button, you set the **Style** property. Later in this chapter, you'll see how to create your own styles.

To apply a style to a control, you create a **resource binding** between a control's Style property and a Style resource. You create a resource binding in XAML by specifying the resource in a **markup extension**—an expression enclosed in curly braces ({}) of the form {*ResourceType ResourceKey*}. Line 56

```
Style="{StaticResource UndoAppBarButtonStyle}"
```

applies the UndoAppBarButtonStyle, and line 58

```
Style="{StaticResource DeleteAppBarButtonStyle}"
```

applies the DeleteAppBarButtonStyle. These can be created by directly modifying the XAML. You can also click the control's Style property in the **Properties** window, select **Local Resource**, then select **UndoAppBarButtonStyle** or **DeleteAppBarButtonStyle** from the drop-down menu.

There are **static resources** that are applied at initialization time only and **dynamic resources** that 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. Styles typically do not change at runtime, so they're normally 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.

By default, the AppBar button styles are located in XML comments within the StandardStyles.xaml file. To use these styles you must first *remove them from the comments*. The easiest way to do this is to either locate the comment and remove its delimiters (<!-- and -->) or copy the corresponding Style element and paste it into the file outside the comment's delimiters.

***Canvas* Control**

The painting area of the `Painter` app is a **`Canvas`** contained in a one-cell `Grid` (lines 68–74). A `Canvas` is a *layout container* that allows you to *position controls* by *defining explicit coordinates* from the `Canvas`'s upper-left corner. For this canvas, we set the following properties:

- **`x:Name`**—Specifies the variable name used in C# code to interact with the control programmatically.
- **`Margin`** (**Layout** category)—Specifies the spacing around the `Canvas`.
- **`Background`** (**Brush** category)—Specifies the background color of the `Canvas`.

We also used the **Event handlers** tab (⚡) to create event handlers for the Canvas's `PointerPressed`, `PointerMoved`, `PointerReleased`, `PointerCanceled`, `PointerCaptureLost` and `PointerExited` events, which Section 25.3.3 discusses in detail.

25.3.3 Event Handling

Event handling in Windows 8 UI is similar to Windows `Forms` event handling. Figure 25.13 shows `MainPage.xaml`'s code-behind file. As in Windows `Forms`, when you double click a control in **Design** view, the IDE *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 method name of the event handler that responds to the event. For example, in line 30 of Fig. 25.10, the attribute

```
Checked="redRadioButton_Checked"
```

specifies that the `redRadioButton`'s `Checked` event handler is `redRadioButton_Checked`.

```
1  // Fig. 25.13: MainPage.xaml.cs
2  // Code-behind for MainPage.xaml.
3  using System;
4  using Windows.UI; // Colors class with predefined colors
5  using Windows.UI.Input; // types related to event handling
6  using Windows.UI.Xaml; // types that support XAML
7  using Windows.UI.Xaml.Controls; // XAML GUI controls and supporting types
8  using Windows.UI.Xaml.Input; // types related to event handling
9  using Windows.UI.Xaml.Media; // graphics and multimedia capabilities
10 using Windows.UI.Xaml.Shapes; // Ellipse class and other shapes
11
12 namespace Painter
13 {
14    public sealed partial class MainPage : Page
15    {
16       private Sizes diameter = Sizes.MEDIUM; // set diameter of circle
17       private Brush brushColor =
18          new SolidColorBrush(Colors.Black); // set the drawing color
19       private bool shouldPaint = false; // specify whether to paint
20
```

Fig. 25.13 | Code-behind for `MainPage.xaml`. (Part 1 of 4.)

```
21         private enum Sizes // size constants for diameter of the circle
22         {
23            SMALL = 5,
24            MEDIUM = 15,
25            LARGE = 30
26         } // end enum Sizes
27
28         public MainPage()
29         {
30            this.InitializeComponent();
31         } // end constructor
32
33         // paints a circle on the Canvas
34         private void PaintCircle(Brush circleColor, PointerPoint point)
35         {
36            Ellipse newEllipse = new Ellipse(); // create an Ellipse
37
38            newEllipse.Fill = circleColor; // set Ellipse's color
39            newEllipse.Width =
40               Convert.ToInt32(diameter); // set its horizontal diameter
41            newEllipse.Height =
42               Convert.ToInt32(diameter); // set its vertical diameter
43
44            // set the Ellipse's position
45            Canvas.SetTop(newEllipse, point.Position.Y);
46            Canvas.SetLeft(newEllipse, point.Position.X);
47
48            paintCanvas.Children.Add(newEllipse);
49         } // end method PaintCircle
50
51         // user chose red
52         private void redRadioButton_Checked(object sender,
53            RoutedEventArgs e)
54         {
55            brushColor = new SolidColorBrush(Colors.Red);
56         } // end method redRadioButton_Checked
57
58         // user chose blue
59         private void blueRadioButton_Checked(object sender,
60            RoutedEventArgs e)
61         {
62            brushColor = new SolidColorBrush(Colors.Blue);
63         } // end method blueRadioButton_Checked
64
65         // user chose green
66         private void greenRadioButton_Checked(object sender,
67            RoutedEventArgs e)
68         {
69            brushColor = new SolidColorBrush(Colors.Green);
70         } // end method greenRadioButton_Checked
71
```

Fig. 25.13 | Code-behind for `MainPage.xaml`. (Part 2 of 4.)

```
72          // user chose black
73          private void blackRadioButton_Checked(object sender,
74             RoutedEventArgs e)
75          {
76             brushColor = new SolidColorBrush(Colors.Black);
77          } // end method blackRadioButton_Checked
78
79          // user chose small brush size
80          private void smallRadioButton_Checked(object sender,
81             RoutedEventArgs e)
82          {
83             diameter = Sizes.SMALL;
84          } // end method smallRadioButton_Checked
85
86          // user chose medium brush size
87          private void mediumRadioButton_Checked(object sender,
88             RoutedEventArgs e)
89          {
90             diameter = Sizes.MEDIUM;
91          } // end method mediumRadioButton_Checked
92
93          // user chose large brush size
94          private void largeRadioButton_Checked(object sender,
95             RoutedEventArgs e)
96          {
97             diameter = Sizes.LARGE;
98          } // end method largeRadioButton_Checked
99
100         // remove last ellipse that was added to the paintCanvas
101         private void undoButton_Click(object sender, RoutedEventArgs e)
102         {
103            int count = paintCanvas.Children.Count;
104
105            // if there are any shapes on Canvas remove the last one added
106            if (count > 0)
107               paintCanvas.Children.RemoveAt(count - 1);
108         } // end method undoButton_Click
109
110         // deletes the entire drawing
111         private void deleteButton_Click(object sender, RoutedEventArgs e)
112         {
113            paintCanvas.Children.Clear(); // clear the canvas
114         } // end method deleteButton_Click
115
116         // handles paintCanvas's PointerPressed event
117         private void paintCanvas_PointerPressed(object sender,
118            PointerRoutedEventArgs e)
119         {
120            shouldPaint = true; // the user is drawing
121         } // end method paintCanvas_PointerPressed
122
```

Fig. 25.13 | Code-behind for `MainPage.xaml`. (Part 3 of 4.)

```
123         // handles paintCanvas's PointerMoved event
124         private void paintCanvas_PointerMoved(object sender,
125            PointerRoutedEventArgs e)
126         {
127            if (shouldPaint)
128            {
129               // draw a circle of selected color at current pointer position
130               PointerPoint pointerPosition = e.GetCurrentPoint(paintCanvas);
131               PaintCircle(brushColor, pointerPosition);
132            } // end if
133         } // end method paintCanvas_PointerMoved
134
135         // handles paintCanvas's PointerReleased event
136         private void paintCanvas_PointerReleased(object sender,
137            PointerRoutedEventArgs e)
138         {
139            shouldPaint = false; // the user finished drawing
140         } // end method paintCanvas_PointerReleased
141
142         // handles paintCanvas's PointerCanceled event
143         private void paintCanvas_PointerCanceled(object sender,
144            PointerRoutedEventArgs e)
145         {
146            shouldPaint = false; // the user finished drawing
147         } // end method paintCanvas_PointerCanceled
148
149         // handles paintCanvas's PointerCaptureLost event
150         private void paintCanvas_PointerCaptureLost(object sender,
151            PointerRoutedEventArgs e)
152         {
153            shouldPaint = false; // the user finished drawing
154         } // end method paintCanvas_PointerCaptureLost
155
156         // handles paintCanvas's PointerExited event
157         private void paintCanvas_PointerExited(object sender,
158            PointerRoutedEventArgs e)
159         {
160            shouldPaint = false; // the user finished drawing
161         } // end method paintCanvas_PointerExited
162      } // end class MainPage
163   } // end namespace Painter
```

Fig. 25.13 | Code-behind for `MainPage.xaml`. (Part 4 of 4.)

Instance Variables and `Sizes` *enum of Class* `MainPage`

Lines 16–19 declare the classes instance variables that maintain the current *brush size* (`diameter`), the *brush color* (`brushColor`) and *whether or not the user is currently drawing* (`shouldPaint`). The `Sizes` enum (lines 21–26) defines the diameters of the `SMALL`, `MEDIUM` and `LARGE` brush sizes.

Method `PaintCircle`

The **Painter** app draws by placing colored circles on the `Canvas` at the pointer position as you *drag the pointer*. The `PaintCircle` method (lines 34–49 in Fig. 25.13) creates the cir-

cle by defining an **Ellipse** object (lines 36–42), and positions it using the **Canvas.SetTop** and **Canvas.SetLeft** methods (lines 45–46), which change the Ellipse object's **Canvas.Left** and **Canvas.Top** properties, respectively. These two properties are actually defined by a *different* control than that to which they're applied. In this case, Left and Top are defined by the Canvas but applied to the objects *contained* in the Canvas. Such properties are known as **attached properties**, because they're attached to the child objects.

The Canvas's **Children** property stores a list (of type **UIElementCollection**) of a layout container's child elements that you can manipulate programmatically. You can add an element to the list by calling the **Add** method on the Children property (for example, line 48). The **Undo** and **Delete** Buttons invoke the **RemoveAt** and **Clear** methods on the Children property (lines 107 and 113, respectively) to remove one child and to clear all the children, respectively.

RadioButton Events

Just as with a Windows Forms RadioButton, a Windows 8 UI RadioButton has a Checked event. Lines 52–98 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 event-handler methods 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.

Button Events

The Button control's **Click** event also functions as in Windows Forms. Lines 101–114 handle the **Undo** and **Delete** Buttons. The **Undo** Button's event handler determines whether the Canvas has any children (i.e., paintCanvas.Children.Count is greater than 0), and if so removes the last child element from the Children collection. The **Delete** Button's event handler clears the entire Children collection, thus deleting the drawing.

User Input Events

Windows 8 UI has built-in support for keyboard and mouse events that's similar to the support in Windows Forms. Because Windows 8 runs on both tablets and desktop computers, some users will interact with your app by touching the screen, some will use a stylus (a pen-like device) and some will use a mouse. In Windows 8 a mouse, a stylus or a finger are known generically as *pointers*, and pointer events enable your apps to respond to these types of input in a uniform manner. Windows 8 also supports keyboard events. For devices that do not have hardware keyboards, "soft" (on-screen) keyboards are displayed when the user must supply keyboard input. The pointer and keyboard events are shown in Fig. 25.14. Handling user interactions in Windows Store apps is discussed in detail at:

```
msdn.microsoft.com/en-us/library/windows/apps/hh465397.aspx
```

Pointer and keyboard events	
Pointer Events with an Event Argument of Type **PointerRoutedEventArgs**	
PointerEntered	Pointer entered an element's bounds.

Fig. 25.14 | Pointer (touch/mouse) and keyboard events. (Part 1 of 2.)

Pointer and keyboard events	
`PointerExited`	Pointer exited an element's bounds.
`PointerMoved`	Pointer moved within an element's bounds.
`PointerPressed`	Mouse button pressed with the mouse pointer inside an element's bounds or finger/stylus touched an element.
`PointerReleased`	Mouse button released with the mouse pointer inside an element's bounds or finger/stylus removed from an element.
`PointerWheelChanged`	Mouse wheel scrolled.
`PointerCaptureLost`	A pointer can be captured in a `PointerPressed` event handler by calling the control's `CapturePointer` method, at which point only that control can generate pointer events until capture is lost. `PointerCaptureLost` can occur when the pointer moves to another app or when the pointer is released.
`PointerCanceled`	This occurs when a pointer unintentionally makes contact with an element, then the user removes the pointer from that element to cancel the interaction—for example, when the user touches a `Button` then moves the mouse or drags a finger/stylus out of the `Button`'s bounds to indicate that the `Button`'s action should *not* be performed.
Keyboard Events with an Event Argument of Type **`KeyRoutedEventArgs`**	
`KeyDown`	Key was pressed.
`KeyUp`	Key was released.

Fig. 25.14 | Pointer (touch/mouse) and keyboard events. (Part 2 of 2.)

The **Painter** app uses the `Canvas`'s `PointerMoved` event handler (lines 124–133) to paint when `shouldPaint` is `true`. A control's `PointerMoved` event is triggered whenever a *pointer (i.e., finger, stylus or mouse) moves* while *within the boundaries of the control*. Information for the event is passed to the event handler using a **`PointerRoutedEventArgs`** object, which contains pointer-specific information. `PointerRoutedEventArgs` method **`GetCurrentPoint`** (line 130), for example, returns the *current position of the pointer relative to the upper-left corner of the control that triggered the event*.

The app draws only when the user *drags a pointer* on the `Canvas`—that is, when the user presses and holds a mouse button while moving the mouse or drags a finger or stylus without removing it from the screen. To ensure this, instance variable `shouldPaint` is set to `true` when the `Canvas`'s `PointerPressed` event occurs (handled in lines 117–121).

The app sets `shouldPaint` to `false` for any event that might cause the drag operation to terminate (lines 136–161). In Windows `Forms`, mouse press and release events occur in pairs; however, according to Microsoft's documentation, this is *not guaranteed* for `PointerPressed` and `PointerReleased` events. For this reason, if your app needs to know when a mouse button is released or a finger/stylus is removed from the screen, Microsoft recommends that you also handle the `PointerCanceled`, `PointerCaptureLost` and `PointerExited` events (lines 143–161).

25.4 CoverViewer App: Data Binding, Data Templates and Styles

The **CoverViewer** app (Fig. 25.15) allows you to browse through book-cover images for several recent Deitel textbooks, by selecting a book from a **`ListView` control** (similar to the Windows `Forms` `ListBox` control) at the app's left side. The large book cover image is then displayed in an **`Image` control** (similar to a Windows `Forms` `PictureBox` control). You can easily adapt this program to display any type of image.

This app demonstrates several new concepts:

- We use a collection of objects as the source of the data that's displayed in the `ListView` control. Specifying a data source for a `ListView` (or other control) so that it can automatically display the data is known as **data binding**.
- We customize how the data is displayed in the `ListView`'s items by defining our own `Style` for the text and using a **`DataTemplate`** to define how the data should be presented in each `ListView`'s item.
- We use nested `Grid` layouts with multiple rows and columns to organize the app's GUI.

For this app, we use the **Blank App** template with its default `Dark` theme. Recall that the `Dark` theme is recommended for apps that display lots of images and video. This app's images are located in the `images` folder with the chapter's examples. To add images to a new project, you can simply drag this folder from the **File Explorer** window, into the project's `Assets` folder in the **Solution Explorer** window.

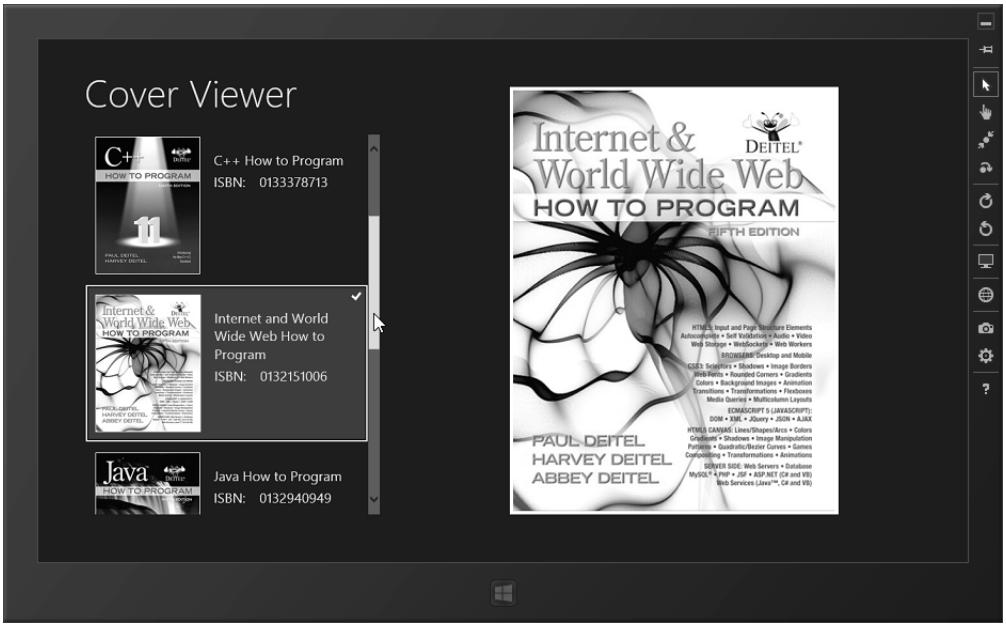

Fig. 25.15 | `CoverViewer` app running in the Windows 8 simulator.

Class ***Book***

Each book in this app is represented by a `Book` object, which has four `string` properties:

1. `ThumbImage`—the path to the small cover image of the book.
2. `LargeImage`—the 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. Because this class is straightforward, its full source code is not presented here. You can view it in the IDE by opening the `Book.cs` file in this example's project. As you'll see in Section 25.4.3, we create a collection of `Book` objects when this app executes, then use the collection as the data source for the app's `ListView`.

25.4.1 `MainPage.xaml` for the CoverViewer App

Figure 25.16 presents the **CoverViewer** app's XAML markup. Lines 13–54 (discussed in the next section) define a `Style` and a `DataTemplate` in the `Page`'s `Page.Resources` element—which contains resources that are used only in the current `Page`. In this section, we present the `Page`'s `Grid`-based layout (lines 56–89) and show how to bind data to the `Image` element's `Source` attribute so that the `Image` can automatically display the selected book's large cover.

```
1  <!-- Fig. 25.16: MainPage.xaml -->
2  <!-- Data binding, data templates and custom styles -->
3  <Page
4      x:Class="CoverViewer.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:CoverViewer"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12    <!-- Define Page's resources -->
13    <Page.Resources>
14       <!-- text style for TextBlocks in the ListView -->
15       <Style x:Key="TextStyle" TargetType="TextBlock"
16          BasedOn="{StaticResource BasicTextStyle}">
17          <Setter Property="TextBlock.FontSize" Value="20" />
18          <Setter Property="TextBlock.TextWrapping" Value="Wrap" />
19       </Style>
20
21       <!-- define data template for displaying Book data in ListView -->
22       <DataTemplate x:Key="BookTemplate">
23          <Grid Margin="10">
24             <Grid.ColumnDefinitions>
25                <ColumnDefinition Width="auto" />
26                <ColumnDefinition Width="*" />
27             </Grid.ColumnDefinitions>
```

Fig. 25.16 | Data binding, data templates and custom styles. (Part 1 of 3.)

```
28
29              <!-- bind book's thumb image to Image element's Source -->
30              <Border BorderBrush="White" BorderThickness="1">
31                 <Image Grid.Column="0" Source="{Binding Path=ThumbImage}"
32                    Stretch="Uniform" MinHeight="75" MaxHeight="200" />
33              </Border>
34
35              <!-- layout the book's title and ISBN text-->
36              <StackPanel Grid.Column="1" Margin="10">
37                 <!-- bind book's Title to TextBlock's Text -->
38                 <TextBlock Margin="10, 10, 10, 5"
39                    Text="{Binding Path=Title}"
40                    Style="{StaticResource TextStyle}"/>
41
42                 <StackPanel Orientation="Horizontal">
43                    <TextBlock Text="ISBN: " Margin="10, 0, 10, 10"
44                       Style="{StaticResource TextStyle}" />
45
46                    <!-- bind book's ISBN to TextBlock's Text -->
47                    <TextBlock Text="{Binding Path=ISBN}"
48                       Margin="10, 0, 10, 10"
49                       Style="{StaticResource TextStyle}" />
50                 </StackPanel>
51              </StackPanel>
52           </Grid>
53        </DataTemplate>
54     </Page.Resources>
55
56     <Grid
57        Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
58
59        <!-- define two rows -->
60        <Grid.RowDefinitions>
61           <RowDefinition Height="140"/>
62           <RowDefinition Height="*"/>
63        </Grid.RowDefinitions>
64
65        <!-- define three columns -->
66        <Grid.ColumnDefinitions>
67           <ColumnDefinition Width="70"/>
68           <ColumnDefinition Width="1*"/>
69           <ColumnDefinition Width="2*"/>
70        </Grid.ColumnDefinitions>
71
72        <!-- page title displayed in row 0, column 1-->
73        <TextBlock Grid.Column="1" Text="Cover Viewer"
74           Style="{StaticResource PageHeaderTextStyle}"/>
75
76        <!-- use ListView and data template to display book information -->
77        <ListView x:Name="booksListView" Grid.Row="1" Grid.Column="1"
78           Margin="0,0,0,70" ItemTemplate="{StaticResource BookTemplate}" />
79
```

Fig. 25.16 | Data binding, data templates and custom styles. (Part 2 of 3.)

```
80          <!-- Border around Image -->
81          <Border Grid.Row="0" Grid.Column="2" Grid.RowSpan="2"
82             HorizontalAlignment="Center" VerticalAlignment="Center"
83             BorderBrush="White" BorderThickness="5" Margin="70">
84
85             <!-- bind Image's Source to selected item's full-size image -->
86             <Image Source="{Binding ElementName=booksListView,
87                Path=SelectedItem.LargeImage}" />
88          </Border>
89       </Grid>
90    </Page>
```

Fig. 25.16 | Data binding, data templates and custom styles. (Part 3 of 3.)

`Grid` with Rows and Columns

By default, a `Grid` layout contains only *one* cell unless you define rows and columns for it. Lines 56–89 define the app's GUI in a `Grid` layout with two rows and three columns. You define the rows and columns by setting the `Grid`'s **`RowDefinitions`** and **`ColumnDefinitions`** properties (located in the **Layout** section of the **Properties** window). These contain collections of **`RowDefinition`** and **`ColumnDefinition`** objects, respectively. Because these properties do not take `string` values, they cannot be specified as attributes in the `Grid` tag. In such cases, a class's property can be defined in XAML as a *nested element* with the name *ClassName.PropertyName*. For example, the `Grid.RowDefinitions` element in lines 60–63 sets the `Grid`'s `RowDefinitions` property and defines two rows—one for the app name at the top of the UI and one for the `ListView` and `Image`, as shown in Fig. 25.15. Rows and columns are each indexed from 0.

You can specify the `Height` of a `RowDefinition` and the `Width` of a `ColumnDefinition` 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 is `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 `Width`s and `Height`s are set to `*`, so every cell in the grid is of *equal size*. In the **CoverViewer** app, the first column (line 67) is 70 pixels wide and the *remaining space* is allocated to the two other columns. The second and third column sizes are specified as 1* and 2*, respectively, so these columns use 1/3 and 2/3 of the remaining space, respectively.

If you click the ellipsis button next to the `RowDefinitions` or `ColumnDefinitions` property in the **Properties** window, the **Collection Editor** window (Fig. 25.17) will appear. 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 `ListView` to specify predefined `ListView` items.

Placing Controls in the `Grid` Using Attached Properties

To indicate a control's location in the `Grid`, you use the **`Grid.Row`** and **`Grid.Column`** attached properties. For the selected control in **Design** or **XAML** view, these appear with the names **Row** and **Column** in the **Layout** section of the **Properties** window.

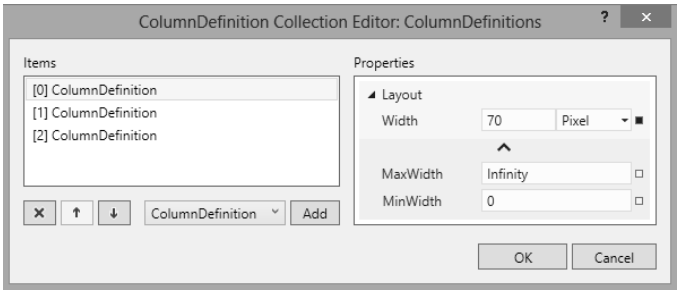

Fig. 25.17 | **Collection Editor** for a `Grid`'s `ColumnDefinitions` property.

`TextBlock` Containing the Page Header

Lines 73–74 define a `TextBlock` that displays the app name. Many of the *preconfigured `Page` templates* provided by Visual Studio show a page header that describes the page as a `TextBlock` in a 140-pixel row at the top of the `Page`, so we chose to mimic that in the **CoverViewer** app. `StandardStyles.xaml` provides the `PageHeaderTextStyle` (used in line 74) for this text. When defining a control in a `Grid`, the control is placed in row 0 and column 0 unless you specify otherwise. Since the `TextBlock` specifies only the `Grid.Column` attribute with the value `1`, the `TextBlock` is located in row 0 and column 1.

`ListView` for Displaying a `Book`'s Thumbnail Cover Image, Title and ISBN

Lines 77–78 define the `ListView` that displays the collection of `Books`, which appears in row 1 and column 1 of the `Grid`. The **`ItemTemplate`** property specifies how each item in the `ListView` is displayed. By default, items are displayed as text, but you can customize this with a `DataTemplate`. In this case, the `ItemTemplate` is a `StaticResource` named `BookTemplate`, which we discuss in detail in Section 25.4.2.

`Border` and `Image` for Displaying the Selected Book's Large Cover Image

Lines 81–88 define a 5-pixel, white **`Border`** that contains an `Image` (lines 86–87) in which the selected `Book`'s large cover image is displayed. The `Border` is placed in row 0 and column 2, and *spans* both row 0 and row 1, as specified with the attached property **`Grid.RowSpan`**. Similarly, there is a **`Grid.ColumnSpan`** property to indicate that a control spans more than one column. The `HorizontalAlignment` and `VerticalAlignment` properties (line 82) indicate that this element will be centered within column 2.

Binding the `Image`'s `Source` Property to the `ListView`'s `SelectedItem`

Often, an app needs to edit and display data. Windows 8 UI enables GUIs to easily interact with data. A *data binding* is a pointer to data that's represented by a **`Binding`** object. You can create such bindings to a broad range of data types, including objects, collections, data in XML documents, data in databases and LINQ query results.

Like resource bindings, data bindings can be created declaratively in XAML markup with *markup extensions*. To declare a data binding, you must specify the *data's source*. If it's another element in the XAML markup, you use property **`ElementName`**; otherwise, you use **`Source`**. Then, if you're binding to a specific property of a control or other object, you must specify the **`Path`** to that piece of information.

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. In lines 86–87

```
<Image Source="{Binding ElementName=booksListView,
   Path=SelectedItem.LargeImage}" />
```

the `Binding`'s `ElementName` property is the name of the selector control, `booksListView`. Each item in this `ListView` is a `Book` object, so the `ListView`'s `SelectedItem` represents one `Book`. Setting the `Binding`'s `Path` to `SelectedItem.LargeImage` indicates the specific `Book` property (`LargeImage`) to use as the `Image`'s `Source`.

Many controls have *built-in support for data binding*, and do *not* require a separate `Binding` object. A `ListView`, for example, has an **`ItemsSource`** property that specifies the data source of the `ListView`'s items. There is *no need to create a binding*—instead, you can just set the `ItemsSource` property (which we do in Fig. 25.18, line 39) 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.

25.4.2 Defining Styles and Data Templates

Throughout this chapter, we've used various standard `Style`s that are defined in an app's `StandardStyles.xaml` file. One advantage of Windows 8 UI over Windows `Forms` is that you can *customize the look-and-feel of controls* to meet your app's needs. In addition, many controls that allowed only text content in Windows `Forms` are `ContentControl`s in Windows 8 UI. Such controls can host *any* type of content—including other controls. The caption of a Windows 8 UI `Button`, for example, could be an image or even a video.

In this section, we demonstrate how to use styles to achieve a *uniform look-and-feel*. In Windows `Forms`, if you want to make all your `Label`s look the same, you have to manually set properties for every `Label`, or copy and paste. To achieve the same result in Windows 8 UI, you can define the properties once as a `Style` and apply the style to each `TextBlock`.

Normally, a `ListView` presents its items as text. This section also introduces *data templates*, which enable you to define how data is presented in a data-bound control. We'll use this capability to display each `Book` object's `ListView` item as a thumbnail image, a title and an ISBN number.

Defining Styles and Data Templates as `Page` *Resources*

Styles and data templates are **Windows 8 UI resources**. A *resource* is an object that is defined for one or more `Page`s of your app and can be *reused* multiple times. Every Windows 8 UI control can hold a collection of resources that can be accessed by any of the control's *nested* elements. If you define a style as a `Page` resource, then any element in the `Page` 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 can also define app-level resources for an `Application` object in the `App.xaml` file—the contents of `StandardStyles.xaml` are specified in the `App.xaml` file as app-level resource. These resources can be accessed in any file in the app. A `Page`'s resources are defined in the **`Page.Resources` element** (lines 13–54, Fig. 25.16).

`TextBlock` Style

Lines 15–19 (Fig. 25.16) define a **`Style`** element that's applied to the `TextBlock`s in our `ListView`'s items. The required `x:Key` attribute must be set in every `Style` (or other resource) so that it can be referenced later by other controls. The `Style`'s required attribute **`x:TargetType`** indicates the type of control that the style can be applied to. The children of a `Style` element can set properties and define event handlers. A **`Setter`** element sets a property to a specific value (e.g., line 17, which sets a `TextBlock`'s `FontSize` property to 20). An **`EventSetter`** element specifies the method that responds to an event, as in

```
<EventSetter Event="ControlName.EventName" Handler="EventHandlerName"/>
```

The `Style` in lines 15–19 (named `TextStyle`) uses two `Setter`s to specify a `TextBlock`'s `FontSize` and `TextWrapping` values. Because some of the book titles are long, we set the `TextWrapping` property to `Wrap`.

`DataTemplate` for `ListView` Items

Lines 22–53 (Fig. 25.16) use a **`DataTemplate`** element to define how to display *bound data* in the `ListView`'s items. Each book, instead of being displayed as text in a `ListView` item, is represented by a small thumbnail of its cover image with its title and ISBN text displayed to the right of the image. You apply a *data template* by using a *resource binding*. To apply a data template to items in a `ListView`, use the **`ItemTemplate`** property (as in line 78).

A *data template* uses *data bindings* (lines 31, 39 and 47) to specify which data to display. In each case, we can omit the data binding's `ElementName` property, because the `ListView`'s data source is specified when the app begins executing (as you'll see in Fig. 25.18).

The `BookTemplate` *data template* arranges controls in a two-column `Grid`. The left column displays a `Border` containing an `Image` and the right column uses nested `StackPanel`s to display a `Book`'s `Title` and `ISBN`. Line 31 binds the `Image`'s `Source` to the `Book`'s `ThumbImage` property, which specifies the location of the thumbnail cover image in the project. The `Book`'s `Title` and `ISBN` are displayed to the right of the book using `TextBlock`s. The `TextBlock` in lines 38–40 binds the `Book`'s `Title` to the `Text` property. Lines 43–49 display two additional `TextBlock`s, one that displays `ISBN:`, and another that's bound to the `Book`'s `ISBN`.

25.4.3 `MainPage.xaml.cs`: Binding Data to the `ListView`'s `ItemSource`

Figure 25.18 presents the code-behind class for the `Cover Viewer`. When the `Page` is created, a collection of six `Book` objects is initialized (lines 19–37) and set as the `ItemsSource` of the `booksListView`, meaning that each item displayed in the `ListView` is one of the `Book`s.

```
1  // Fig. 25.18: MainPage.xaml.cs
2  // Using data binding
3  using System;
4  using System.Collections.Generic;
5  using Windows.UI.Xaml.Controls;
```

Fig. 25.18 | Using data binding. (Part 1 of 2.)

```
6
7  namespace CoverViewer
8  {
9     public sealed partial class MainPage : Page
10    {
11       // list of Book objects
12       private List<Book> books = new List<Book>();
13
14       public MainPage()
15       {
16          this.InitializeComponent();
17
18          // add Book objects to the List
19          books.Add(new Book("C How to Program", "013299044X",
20             "assets/images/small/chtp.jpg",
21             "assets/images/large/chtp.jpg"));
22          books.Add(new Book("C++ How to Program", "0133378713",
23             "assets/images/small/cpphtp.jpg",
24             "assets/images/large/cpphtp.jpg"));
25          books.Add(new Book(
26             "Internet and World Wide Web How to Program", "0132151006",
27             "assets/images/small/iw3htp.jpg",
28             "assets/images/large/iw3htp.jpg"));
29          books.Add(new Book("Java How to Program", "0132940949",
30             "assets/images/small/jhtp.jpg",
31             "assets/images/large/jhtp.jpg"));
32          books.Add(new Book("Visual Basic How to Program", "0133406954",
33             "assets/images/small/vbhtp.jpg",
34             "assets/images/large/vbhtp.jpg"));
35          books.Add(new Book("Visual C# How to Program", "0133379337",
36             "assets/images/small/vcshtp.jpg",
37             "assets/images/large/vcshtp.jpg"));
38
39          booksListView.ItemsSource = books; // bind data to the list
40          booksListView.SelectedIndex = 0; // select first item in ListView
41       } // end constructor
42    } // end class MainPage
43 } // end namespace CoverViewer
```

Fig. 25.18 | Using data binding. (Part 2 of 2.)

25.5 App Lifecycle

So far, you've learned how to create GUIs for Windows Store apps. There is much more to Windows Store app development than we've shown here. For example there is a well-defined app lifecycle in which an app at any given time is *not running*, *running*, *suspended*, *terminated* or *closed by the user*:

- An app that's *not running* has not yet been launched by the user.
- An app that's *running* is on the screen so that the user can interact with it.
- An app that's *suspended* is not currently on the screen, so Windows 8 suspends its execution to conserve power and reduce the load on the CPU.

- An app that's *terminated* was shut down by Windows 8—typically to recover resources like memory that are needed by other apps.
- An app that's *closed by the user* was shut down explicitly by the user by swiping from the top of the screen to the bottom.

Various events occur during an app's lifecycle:

- The `Suspending` event occurs when an app transitions to the *suspended* state—usually because the app is no longer visible on the screen. This typically occurs when the user runs another app.
- The `Resuming` event occurs when the app transitions from *suspended* to *running*.

The complete details of the Windows Store app lifecycle can be found at

```
msdn.microsoft.com/en-us/library/windows/apps/hh464925.aspx
```

Articles and tutorials about Windows Store app concepts can be found at:

```
msdn.microsoft.com/en-us/library/windows/apps/hh750302.aspx
```

25.6 Wrap-Up

In this chapter, you created several GUI-based apps with Windows 8 UI. You learned how to design a Windows 8 UI GUI with XAML markup and how to give it functionality in a C# code-behind class. We presented various Windows 8 UI layouts, in which we specified a control's size and position relative to other controls. We demonstrated the flexibility Windows 8 UI offers for customizing the look-and-feel of your GUIs with styles and data templates. You learned how to create data-driven GUIs with data bindings. Finally, we overviewed the Windows Store app lifecycle and provided links to resources where you can learn more about Windows Store app development.

Windows 8 UI is not merely a GUI-building platform. Chapter 26 explores some of the many other capabilities of Windows 8 UI, showing you how to incorporate 2D graphics, animation and multimedia into your Windows 8 UI apps.

Summary

Section 25.1 Introduction

- There are three technologies for building GUIs in Windows—Windows `Forms`, Windows 8 UI and Windows Presentation Foundation (WPF).
- Windows 8 UI is Microsoft's Windows 8 framework for GUI, graphics, animation and multimedia.
- Windows Presentation Foundation (WPF) was introduced in .NET 3.0 as the replacement for Windows `Forms` and various graphics and multimedia capabilities.
- Apps that use the Windows 8 UI are known as *Windows Store apps* and can be submitted to the Windows Store as free or for-sale apps.
- XAML (pronounced "zammle")—Extensible Application Markup Language—is an XML vocabulary that can be used to define and arrange GUI controls without any C# code.

- As you build a Windows Store app's GUI visually, the IDE generates XAML. This markup is designed to be readable by both humans and computers, so you can also manually write XAML to define your GUI, and in many cases you will.
- When you compile your app, a XAML compiler generates code to create and configure controls based on your XAML markup.
- The technique of defining what the GUI should contain without specifying how to generate it is an example of declarative programming.

Section 25.2 Welcome App: Introduction to XAML Declarative GUI Programming

- A XAML document defines the appearance of a Windows 8 UI app.
- A `TextBlock` is similar to a `Label` in Windows `Forms`.

Section 25.2.1 Running the Welcome App

- You can test your Windows Store apps three ways by using the drop-down list to the right of **Start Debugging** on the Visual Studio toolbar: **Simulator**, **Local Machine** and **Remote Machine**.
- The **Simulator** option runs the app in a window that simulates a Windows 8 computer and allows you to test capabilities such as rotating a device, touch input, different screen sizes/resolutions and geolocation. You can also use the simulator to take screen captures of your app for use in the Windows Store.
- The **Local Machine** (the default) option runs the app directly on the Windows 8 device that's running Visual Studio.
- The **Remote Machine** option runs the app on a Windows 8 device connected to the Windows 8 computer that's running Visual Studio via a network or cable. Windows 8 tablets have various sensors (e.g, GPS, gyroscope, etc.) that are not available on most desktop and laptop computers. Running an app on a remote machine allows you to test these capabilities of your app.

Section 25.2.2 `MainPage.xaml` for the Welcome App

- A XAML document has a single root element that consists of many nested elements, delimited by start tags and end tags.
- Windows 8 UI controls are represented by elements in XAML markup.
- The root element of a Windows 8 UI XAML document is a `Page` control, which defines the content of the app's page.
- By default, Windows Store apps use the entire screen.
- In Windows 8, it's possible to have a maximum of two Windows Store apps on the screen. One occupies a column that's 320 pixels wide and as tall as the screen, and the other occupies the rest.
- With Windows 8.1, apps must still occupy the screen's height, but they can now be any width and there can be more that two apps on the screen at a time.
- A `Page` represents a screen of content and controls in a Windows Store app.
- The attribute `x:Class` of the `Page` start tag specifies the name of the class that provides the GUI's functionality.
- A XAML document must have an associated code-behind file to handle events.
- Together, the markup in `MainPage.xaml` and its code-behind file represent a derived class of `Page` that defines the `Page`'s GUI and functionality.
- `Page` (from namespace `Windows.UI.XAML.Controls`) is a content control that may contain only one layout container, which arranges other controls in the `Page`.

- A layout container such as a `Grid` can have many child elements, including nested layout containers, allowing it to arrange many controls in your GUI.
- A `Grid` (from namespace `Windows.UI.XAML.Controls`) is a flexible, all-purpose layout container that organizes controls into rows and columns (one row and one column by default).
- Layout containers help you arrange a GUI's controls.
- The `Background` attribute sets the `Grid`'s background color. By default, the IDE set this `Grid` element's `Background` to a predefined style named `ApplicationPageBackgroundThemeBrush`.
- In XAML elements, attributes like `Background` that are *not* qualified with an XML namespace (such as `x:`) correspond to properties of an object.
- A `TextBlock` (from namespace `Windows.UI.XAML.Controls`) displays text in a GUI.
- The `HorizontalAlignment` attribute specifies how a control is aligned horizontally within a layout container. The options are `Left`, `Center`, `Right` and `Stretch`.
- The `Margin` attribute specifies the number of pixels separating a control from other controls (or the edge of the screen). If you specify one value, it applies to the control's left, top, right and bottom edges. If you specify two values—as in `"10, 20"`—the first value applies to the left and right edges, and the second applies to the top and bottom edges. If you specify four values—as in `"10, 20, 30, 40"`—the values apply to the left, top, right and bottom, respectively.
- The `TextWrapping` attribute specifies whether a `TextBlock`'s text wraps to multiple lines if it cannot fit within the `TextBlock`'s width. The options are `Wrap` or `NoWrap` (the default).
- The `FontSize` attribute specifies the font size measured in pixels. The default size is 11.
- The `Text` attribute specifies the text displayed by the `TextBlock`.
- The `TextAlignment` attribute specifies how the text is aligned within the `TextBlock`. The options are `Left`, `Center`, `Right` and `Justify` (i.e., aligned both left and right).
- The `VerticalAlignment` attribute specifies how a control is aligned vertically within a layout container. The options are `Top`, `Center`, `Bottom` and `Stretch`.
- The `FontFamily` attribute specifies the text font.

Section 25.2.3 Creating the Welcome *App's Project*

- To create a new Windows Store app using Windows 8 UI, open the **New Project** dialog and select **Templates > Visual C# > Windows Store** from the list of templates.
- The template **Blank App** defines a single-page app for which you must define your own GUI.
- The template **Grid App** defines a three-page app with predefined layouts and GUI controls. The provided pages allow the user to view a collection of groups of items displayed in a grid of rows and columns, view particular group's details and view a particular item's details. This template also provides support for switching between pages (known as *navigation*) and more.
- The template **Split App** defines a two-page app with predefined layouts and GUI controls. The first page allows the user to view a collection of groups of items displayed in a grid. The second page allows the user to view a particular group *and* a selected item's details. This template also provides support for navigating between pages and more.
- There are two predefined *themes* for Windows Store apps—`Dark` (the default) and `Light`. Microsoft recommends using `Dark` for apps that present mostly images and video and `Light` for apps that display lots of text. To use the `Light` theme open `App.xaml`, then add the `RequestedTheme` attribute with its value set to `"Light"` to the `application` element's opening tag.
- The IDE's **Design** view is used for drag-and-drop GUI design and the **XAML** view shows the generated XAML markup. You can also edit the XAML directly in the XAML view. The IDE syncs these views and the **Properties** window.

Section 25.2.4 Building the App's GUI

- Because XAML is easy to understand and edit, it's often less difficult to manually edit your XAML markup. In some cases, you must manually write XAML markup.
- By default, the selected object's properties are arranged by category in the **Properties** window. Within each category, the most commonly used properties are shown. If a down arrow (⌄) is displayed you can click it to show more of that category's properties.

Section 25.2.5 Overview of Important Project Files and Folders

- When you create a Windows Store app, several files and folders are generated and can be viewed in the **Solution Explorer**.
- `App.xaml` defines the `Application` object and its settings. The most noteworthy element is `Application.Resources`, which specifies resources, such as common styles that define the GUI's look-and-feel, that should be available to all pages of your app.
- `App.xaml.cs` the code-behind file for `App.xaml` specifies app-level event handlers.
- `MainPage.xaml` defines the first `Page` that's loaded in the app. `MainPage.xaml.cs` is its code-behind file, which handles the `Page`'s events.
- `StandardStyles.xaml` (in the `Common` folder) defines styles that enable your app's GUI to have the look-and-feel of Windows Store apps.
- The `Assets` folder typically contains media files, such as images, audio and video files. By default this folder contains four images that are used as logos and a splash screen.
- The `Common` folder contains files typically used by all `Pages` in an app, such as the `StandardStyles.xaml` file.
- `Package.appxmanifest` defines various settings for your app, such as the app's display name, logo, splash screen and more.

Section 25.2.6 Splash Screen and Logos

- The app's splash screen provides a visual indication that the app is loading. The splash screen image is one of four PNG images that are provided by default in the project's `Assets` folder. `Logo.png` is shown on the app's tile in the **Start** screen. `SmallLogo.png` is displayed with your app's name when your app is shown in Windows 8 search results. `SplashScreen.png` is displayed briefly when your app first loads. `StoreLogo.png` is shown in the Windows Store with your app's description and in any Windows Store search results.

Section 25.2.7 `Package.appmanifest`

- The app's manifest file `Package.appmanifest` enables you to quickly configure various app settings, such as the app's display name, its logo images and splash screen, the hardware and software capabilities it uses, information for the Windows Store and more.
- Double clicking the `Package.appmanifest` file in the **Solution Explorer** displays the manifest editor.

Section 25.3 Painter App: Layouts; Event Handling

- Depending on your app's purpose, you may choose to support the orientations **Landscape**, **Portrait**, **Landscape-flipped** and **Portrait-flipped**.

Section 25.3.1 General Layout Principles

- In Windows 8, apps can execute on a variety of desktop, laptop and tablet computers. Tablets in particular can be held in portrait (longer side vertical) or landscape (longer side horizontal) orientations, so a Windows Store app's user interface should be able to dynamically adjust based on

the device orientation. For this reason, a control's size is often specified as a range of values, and its location specified relative to those of other controls.

- Unless necessary, a control's size should not be defined explicitly.
- In addition to the `Width` and `Height` properties associated with every control, all Windows 8 UI controls have the `MinWidth`, `MinHeight`, `MaxHeight` and `MaxWidth` properties. If the `Width` and `Height` properties are both `Auto` (the default), you can use these minimum and maximum properties to specify a range of acceptable sizes for a control.
- A control's position 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 doing this—`Margin`, `HorizontalAlignment` and `VerticalAlignment`. `Margin` specifies how much space to put around a control's edges.
- `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`.
- Windows 8 UI provides many layout containers derived from class `Panel` in the namespace `Windows.UI.Xaml.Control`.
- A `Grid` layout provides a grid of rows and columns, depending on the `RowDefinitions` and `ColumnDefinitions` properties.
- A `Canvas` layout is coordinate based. Element positions are defined explicitly by their distance from the `Canvas`'s top and left edges.
- A `StackPanel`'s elements are arranged in a single row or column, depending on the `Orientation` property.

Section 25.3.2 `MainPage.xaml`: Layouts and Controls in the Painter App

- Each `Page` can have `AppBar`s at the top and bottom of the screen for quick access to commands.
- By default, the `AppBar`s are hidden. You can show them by swiping from the screen's bottom edge on a touch screen or by right clicking with the mouse on a desktop computer. They're dismissed by interacting with the app's content.
- Setting the `AppBar`'s `IsOpen` attribute to `True` indicates that the `AppBar` should be displayed when the app first executes.
- You create an `AppBar` by clicking the `Page` element in the XAML view, then clicking the **New** button to the right of the `BottomAppBar` or `TopAppBar` property in the **Properties** window. This creates a `Page.BottomAppBar` or `Page.TopAppBar` element containing an `AppBar` control. You can then drag layouts and controls into the `AppBar`'s area in **Design** view or edit the `AppBar` element's XAML directly in **XAML** view.
- Windows 8 UI `RadioButton`s function as mutually exclusive options, just like their Windows `Forms` counterparts. A Windows 8 UI `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—it can be labeled by an image or other item specified with control's `Content` property.
- Windows 8 UI does not provide a `GroupBox` control for maintaining the mutually exclusive relationship between `RadioButton`s. Instead, this relationship is based on the `GroupName` property—`RadioButton`s with the same `GroupName` are mutually exclusive.
- A control's `x:Name` attribute in XAML (set by the `Name` property at the top of the **Properties** window) specifies the variable name used in C# code to interact with the control programmatically.

- A `RadioButton`'s `Checked` property specifies the event handler that's called when the user interacts with the control.
- Set a `RadioButton`'s `IsChecked` property to `True` to indicate that the `RadioButton` should be selected when the app begins executing.
- A Windows 8 UI `Button` behaves like a Windows `Forms` `Button` but is a `ContentControl`.
- A `RadioButton`'s `Click` property specifies the event handler that's called when the user interacts with the control.
- A control's `Style` property specifies the look-and-feel of the control.
- `StandardStyles.xaml` provides many pre-configured `AppBar` button styles (and other styles).
- To apply a style to a control, you create a resource binding between a control's `Style` property and a `Style` resource by using a markup extension—an expression enclosed in curly braces (`{}`) of the form `{`*ResourceType ResourceKey*`}`.
- To set a style in the **Properties** window, click the control's `Style` property, select **Custom Expression...** and enter the markup extension code.
- 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. 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.
- By default, the `AppBar` button styles are located in XML comments within the `StandardStyles.xaml` file. To use these styles you must first remove them from the comments.
- A `Canvas` is a layout container that allows you to position controls by defining explicit coordinates from the `Canvas`'s upper-left corner.

Section 25.3.3 Event Handling

- As in Windows `Forms`, when you double click a control in **Design** view, the IDE 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 method name of the event handler that responds to the event.
- The `Canvas.SetTop` and `Canvas.SetLeft` methods change the `Canvas.Left` and `Canvas.Top` properties of an object that's attached to a `Canvas`. These two properties are defined by the `Canvas` but applied to the objects contained in the `Canvas`. Such properties are known as attached properties, because they're attached to the child objects.
- The `Canvas`'s `Children` property stores a list (of type `UIElementCollection`) of the container's child elements. The `Add` method adds an object to the `Children` list. The `RemoveAt` and `Clear` methods remove one child or all the children from the `Children` list, respectively.
- Windows 8 UI has built-in support for keyboard and mouse events that's similar to the support in Windows `Forms`. Because Windows 8 runs on both tablets and desktop computers, some users will interact with your app by touching the screen, some will use a stylus (a pen like device) and some will use a mouse. In Windows 8 a mouse, a stylus or a finger are known generically as *pointers*, and pointer events enable your apps to respond to these types of input in a uniform manner.
- Windows 8 also supports keyboard events. For devices that do not have hardware keyboards, "soft" (on-screen) keyboards are displayed when the user must supply keyboard input.
- The `PointerEntered` event indicates that the pointer entered an element's bounds.
- The `PointerExited` event indicates that the pointer exited an element's bounds.
- The `PointerMoved` event indicates that the pointer moved within an element's bounds.

- The `PointerPressed` event indicates that a mouse button was pressed with the mouse pointer inside an element's bounds or that a finger/stylus touched an element.
- The `PointerReleased` event indicates that a mouse button was released with the mouse pointer inside an element's bounds or that a finger/stylus was removed from an element.
- The `PointerWheelChanged` event indicates that the mouse wheel scrolled.
- A pointer can be captured in a `PointerPressed` event handler by calling the control's `CapturePointer` method, at which point only that control can generate pointer events until capture is lost. The `PointerCaptureLost` event can occur when the pointer moves to another app or when the pointer is released.
- The `PointerCanceled` event occurs when a pointer unintentionally makes contact with an element, then the user removes the pointer from that element to cancel the interaction.
- The `KeyDown` event indicates that a key was pressed.
- The `KeyUp` event indicates that a key was released.
- Information for a pointer event is passed to the event handler using a `PointerRoutedEventArgs` object, which contains pointer-specific information. `PointerRoutedEventArgs` method `GetCurrentPoint`, for example, returns the current position of the pointer relative to the upper-left corner of the control that triggered the event.

Section 25.4.1 `MainPage.xaml` *for the* CoverViewer *App*

- `Styles` and `DataTemplates` for a `Page` are defined in the `Page`'s `Page.Resources` element.
- You define the rows and columns of a `Grid` by setting its `RowDefinitions` and `ColumnDefinitions` properties (located in the **Layout** section of the **Properties** window). These contain collections of `RowDefinition` and `ColumnDefinition` objects, respectively.
- A class's property can be defined in XAML as a nested element with the name *ClassName.PropertyName*.
- You can specify the `Height` of a `RowDefinition` and the `Width` of a `ColumnDefinition` 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.
- 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.
- If you click the ellipsis button next to the `RowDefinitions` or `ColumnDefinitions` property in the **Properties** window, the **Collection Editor** window will appear. This tool can be used to add, remove, reorder, and edit the properties of rows and columns in a `Grid`.
- Any property that takes a collection as a value can be edited in a version of the **Collection Editor** specific to that collection.
- To indicate a control's location in the `Grid`, you use the `Grid.Row` and `Grid.Column` attached properties. For the selected control in **Design** or **XAML** view, these appear with the names **Row** and **Column** in the **Layout** section of the **Properties** window.
- Many of the preconfigured `Page` templates provided by Visual Studio show a page header that describes the page as a `TextBlock` in a 140 pixel row at the top of the `Page`. `StandardStyles.xaml` provides the `PageHeaderTextStyle` for such text.
- When defining a control in a `Grid`, the control is placed in row 0 and column 0 unless you specify otherwise.

- A `ListView`'s `ItemTemplate` property specifies how each item in the `ListView` is displayed. By default, items are displayed as text, but you can customize this with a `DataTemplate`.
- The attached property `Grid.RowSpan` indicates how many rows an element occupies in a `Grid`. Similarly, there is a `Grid.ColumnSpan` property to indicate that a control spans more than one column.
- A data binding is a pointer to data that's represented by a `Binding` object. You can create bindings to a broad range of data types, including objects, collections, data in XML documents, data in databases and LINQ query results.
- Data bindings can be created declaratively in XAML markup with markup extensions. To declare a data binding, you must specify the data's source. If it's another element in the XAML markup, you use property `ElementName`; otherwise, you use `Source`. Then, if you're binding to a specific property of a control or other object, you must specify the `Path` to that piece of information.
- Many controls have built-in support for data binding, and do not require a separate `Binding` object. A `ListView`, for example, has an `ItemsSource` property that specifies the data source of the `ListView`'s items. When you set `ItemsSource` to a collection of data, the objects in the collection automatically become the items in the `ListView`.

Section 25.4.2 Defining Styles and Data Templates

- One advantage of Windows 8 UI over Windows `Forms` is that you can customize the look-and-feel of controls to meet your app's needs. In addition, many controls that allowed only text content in Windows `Forms` are `ContentControl`s in Windows 8 UI. Such controls can host any type of content—including other controls.
- Styles and data templates are Windows 8 UI resources. A resource is an object that is defined for one or more `Page`s of your app and can be reused multiple times. Every Windows 8 UI control can hold a collection of resources that can be accessed by any of the control's nested elements.
- If you define a style as a `Page` resource, then any element in the `Page` 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 can also define app-level resources for an `Application` object in the `App.xaml` file—the contents of `StandardStyles.xaml` are specified in the `App.xaml` file as app-level resource.
- A `Page`'s resources are defined in the `Page.Resources` element.
- A `Style` element's required `x:Key` attribute allows it to be referenced for styling controls. The required attribute `x:TargetType` indicates the type of control that the style can be applied to.
- The children of a `Style` element can set properties and define event handlers. A `Setter` sets a property to a specific value. An `EventSetter` specifies the method that responds to an event.
- A `DataTemplate` element can be used to define how to display bound data in a `ListView`'s items. You apply a data template by using a resource binding. To apply a data template to items in a `ListView`, use the `ItemTemplate` property.
- A data template uses data bindings to specify which data to display.

Section 25.5 App Lifecycle

- There is a well-defined app lifecycle in which an app is at any given time not running, running, suspended, terminated or closed by the user. An app that's not running has not yet been launched. An app that's running is on the screen so that the user can interact with it. An app that's suspended is not currently on the screen, so Windows 8 suspends its execution to conserve power and reduce the load on the CPU. An app that's terminated was shut down by Windows 8—typically to recover resources like memory that are needed by other apps. An app that's closed

by the user was shut down explicitly by the user by swiping from the top of the screen to the bottom.

- Various events occur during an app's lifecycle: The *Suspending* event occurs when an app transitions to the suspended state—usually because the app is no longer visible on the screen. This typically occurs when the user runs another app. The *Resuming* event occurs when the app transitions from suspended to running.

Self-Review Exercises

25.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Windows 8 UI allows designers to define the appearance and content of a GUI in XAML without any C# code.

b) With Windows 8.1, apps must still occupy the screen's height, but they can now be any width (the default minimum width is 500 pixels) and there can be a maximum of two apps on the screen at a time.

c) A `Grid` template is a two-page app with predefined layouts and GUI controls. The first page allows the user to view a collection of groups of items displayed in a grid. The second page allows the user to view a particular group *and* a selected item's details. This template also provides support for navigating between pages and more.

d) Microsoft recommends using the `Dark` theme for apps that display lots of text.

e) When you edit content in the **Design** view, the **XAML** view or the **Properties** window, you must be careful to update the others.

f) In the **Application UI** tab's **Tile** section, which specifies information for the app's **Start** screen tile, changing the **Background color** to `#FFFFFF` causes Windows 8 to display the app's logo and app name on tile with a black background.

g) Tablets can be held in portrait orientation (longer side horizontal).

h) `HorizontalAlignment` and `VerticalAlignment` specify how to align a control within its root container.

i) By default, `AppBar`s are visible.

j) Unlike Windows `Forms`, Windows 8 UI does *not* provide a `GroupBox` control for maintaining the mutually exclusive relationship between `RadioButton`s. Instead, this relationship is based on the `GroupName` property of the `RadioButton`s.

k) When defining a control in a `Grid`, the control is placed in row 1 and column 1 unless you specify otherwise.

l) The caption of a Windows 8 UI `Button` could be an image or even a video.

m) Data templates enable you to define how data is stored in a data-bound control.

n) If you define a style as a resource of a layout container, then only the elements of the layout container can use that style.

o) The `Resuming` event occurs when the app transitions from running to suspended.

25.2 Fill in the blanks in each of the following statements:.

a) Windows `Forms` is considered to be a _____ technology.

b) Apps that use the Windows 8 UI are known as _____ apps.

c) _____ is an XML vocabulary that can be used to define and arrange GUI controls without any C# code.

d) Building a GUI with Windows 8 UI is similar to building a GUI with Windows `Forms`—you drag-and-drop predefined controls from the **Toolbox** onto the _____.

e) As with any other XML document, each XAML document must contain a single _____.

f) A XAML document must have an associated _____ file to handle events.

g) A _____ (from namespace `Windows.UI.XAML.Controls`) is a flexible, all-purpose layout container that organizes controls into rows and columns (one row and one column by default).
h) _____ help you arrange a GUI's controls.
i) In XAML elements, attributes like `Background` that are *not* qualified with an XML namespace (such as `x:`) correspond to _____ of an object.
j) A Windows 8 UI _____ template is a single-page (i.e., one screen) app for which you must define your own GUI.
k) Use the _____ box to quickly locate a given property.
l) The app's _____ provides a visual indication that the app is loading.
m) The app's _____ file enables you to quickly configure various app settings, such as the app's display name, its logo images and splash screen, the hardware and software capabilities it uses, information for the Windows Store and more.
n) Apps in which the user reads lots of text are often better in portrait orientation because _____.
o) The _____ property of each control specifies how much space to put around the control's edges.
p) A Windows 8 UI `RadioButton` does *not* have a `Text` property. Instead, it's a _____, meaning it can have exactly one child or text content.
q) Because styles don't normally change during runtime, they are usually used as _____ resources.
r) A Windows 8 UI `RadioButton` has a `Checked` event. The event-handler method looks almost identical to how it would look in a Windows `Forms` app, except that the event-arguments object is a _____ object.
s) In Windows 8 a mouse, a stylus or a finger are known generically as _____, and the associated events enable your apps to respond to these types of input in a uniform manner.
t) Using a collection of objects as the source of the data that's displayed in a `ListView` control is known as _____.
u) A property of a class can be defined in XAML as a nested element with the name _____ .
v) To indicate a control's location in the `Grid`, you use the `Grid.Row` and `Grid.Column` _____ properties.
w) To apply a data template to items in a `ListView`, use the _____ property.

Answers to Self-Review Exercises

25.1 a) True. b) False. There can be more that two apps on the screen at a time. c) False. It's a **Split App**. d) False. Microsoft recommends the `Light` theme for apps that display lots of text. e) False. The IDE automatically updates the others. f) False. It will display a white background. g) False. Should be "longer side vertical." h) False. Withing its layout container, not root container. i) False. By default `AppBars` are hidden. j) True. k) False.When defining a control in a `Grid`, the control is placed in row 0 and column 0 unless you specify otherwise. l) True. m) False. Data templates enable you to define how data is presented in a data-bound control. n) True. o) False. The `Resuming` event occurs when the app transitions from suspended to running.

25.3 a) legacy. b) Windows Store apps. c) XAML. d) design area. e) root element. f) code-behind. g) `Grid`. h) Layout containers. i) properties. j) **Blank App**. k) **Search Properties** box. l) splash screen. m) `Package.appmanifest`. n) it's easier to read short lines of text. o) `Margin`. p) `ContentControl`. q) static. r) `RoutedEventArgs`. s) pointers. t) data binding. u) *ClassName.PropertyName.* v) attached. w) `ItemTemplate`.

Exercises

25.4 *(Scrapbooking App)* Create an app that displays the heading "My Favorite Pictures" at the top of the app, and shows four pictures along with text that identifies each image. Use the layouts you learned in this chapter.

25.5 *(Enhanced Painter App)* Modify the **Painter** app in section Section 25.3 to include a top `AppBar` and provide the following additional features:

a) Allow the user to select the `Canvas`'s background color. Clearing the entire image should return the background to the default white.

b) Investigate the `FileOpenPicker` class (`msdn.microsoft.com/library/windows/apps/BR207847`). Provide a `Button` that displays a `FileOpenPicker` from which the user can choose a background image on which to draw. Clearing the entire image should return the `Canvas`'s background to the default white. To draw on an image, place an `Image` element before the `Canvas` element in the same `Grid` cell and set the `Canvas`'s `Background` to `Transparent` so that the `Image` shows through the `Canvas`.

c) Add the ability to draw rectangles and ellipses. Like class `Ellipse`, class `Rectangle` provides `Width`, `Height` and `Fill` options.

25.6 *(PhotoViewer App)* Using the techniques you used in Section 25.4, create a **PhotoViewer** App. Rather than a `ListView`, investigate the `GridView` layout (`msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.controls.gridview.aspx`) and display photo thumbail images on the left side of the app in a a `GridView`. Use a `DataTemplate` for specify how each thumbnail image is displayed in a `GridView` item. Like a `ListView`, a `GridView` provides a `SelectedItem` property that represents the currently selected item.

25.7 *(Cash Register App)* Create a cash-register app with `Buttons` for the digits `0`–`9`, an **Enter** `Button` to add the current amount to the total, a **Delete** `Button` to reset the current input value and a **Clear** `Button` to reset both the current input and the total. The app should use `TextBlocks` to display the current input, tax and total amounts, each with two digits after the decimal point. Use 6.25% as the sales tax rate. The user can enter amounts only as integer values via the digit `Buttons`, so you should divide the value entered so far by 100 before displaying it (e.g., `7` it would be displayed as `0.07`, `73` would be displayed as `0.73` and `730` would be displayed as `7.30`). Use nested `Grid` and `StackPanel` layouts as necessary to present a nicely formatted GUI.

25.8 *(Binding to LINQ Query Results)* You can create data bindings to LINQ queries. Modify the example given in Fig. 9.4 to display a `ListView` of all `Employees` with their names and earnings. Provide a separate `ListView`, two `TextBoxes` and a `Button` that allow the user to search the collection of `Employees` for salaries in a specific range. Bind the LINQ query's results to the second `ListView`. Use a `DataTemplate` to format the `ListViews`' items with the `Employee`'s name in the format *LastName, FirstName* on one line and the `Employee`'s salary below the name.