PART VI

Windows, Pages, and Rich Controls

CHAPTER 23

Windows

Windows are the basic ingredients in traditional desktop applications—so basic that the operating system itself is named after them.

In the years since the window model was introduced, simplified page-based applications have often challenged the window model. The first and most successful challenger was the Web, with its range of app-like websites that run in the browser. But the page-based model of the Web has also crept into the territory of the desktop. Today, you'll find that Microsoft is busy creating a design framework called Metro, which favors simplified, touch-friendly interfaces that don't use windows at all. And even WPF has its own page-based development model, which you'll consider in Chapter 24. But for complex applications, content creation, and business tools, windows still remain the dominant user interface metaphor.

In this chapter, you'll explore the Window class. You'll learn the various ways to show and position windows, how window classes should interact, and what built-in dialog boxes WPF provides. You'll also look at more exotic window effects, such as nonrectangular windows and windows with transparency. Finally, you'll explore WPF"s support for programming the Windows taskbar.

The Window Class

As you learned in Chapter 6, the Window class derives from ContentControl. That means it can contain a single child (usually a layout container such as the Grid control), and you can paint the background with a brush by setting the Background property. You can also use the BorderBrush and BorderThickness properties to add a border around your window, but this border is added inside the window frame (around the edge of the client area). You can remove the window frame altogether by setting the WindowStyle property to None, which allows you to create a completely customized window, as you'll see later in the "Nonrectangular Windows" section.

Note The client area is the surface inside the window boundaries. This is where you place your content. The nonclient area includes the border and the title bar at the top of the window. The operating system manages this area.

In addition, the Window class adds a small set of members that will be familiar to any Windows programmer. The most obvious are the appearance-related properties that let you change the way the nonclient portion of the window appears. Table 23-1 lists these members.

Table 23-1. *Basic Properties of the Window Class*

Name	Description
AllowsTransparency	When set to true, the Window class allows other windows to show through if the background is set to a transparent color. If set to false (the default), the content behind the window never shows through, and a transparent background is rendered as a black background. This property allows you to create irregularly shaped windows when it's used in combination with a WindowStyle of None, as you'll see in the "Nonrectangular Windows" section.
Icon	An ImageSource object that identifies the icon you want to use for your window. Icons appear at the top left of a window (if it has one of the standard border styles), in the taskbar (if ShowInTaskBar is true), and in the selection window that's shown when the user presses Alt+Tab to navigate between running applications. Because these icons are different sizes, your .ico file should include at least a 16×16 pixel image and a 32×32 pixel image. In fact, the modern Windows icon standard includes both a 48×48 pixel image and a 256×256 image, which can be sized as needed for other purposes. If Icon is a null reference, the window is given the same icon as the application (which you can set in Visual Studio by double-clicking the Properties node in the Solution Explorer and then choosing the Application tab). If this is omitted, WPF will use a standard but unremarkable icon that shows a window.
Icon	An ImageSource object that identifies the icon you want to use for your window. Icons appear at the top left of a window (if it has one of the standard border styles), in the taskbar (if ShowInTaskBar is true), and in the selection window that's shown when the user presses Alt+Tab to navigate between running applications. Because these icons are different sizes, your .ico file should include at least a 16×16 pixel image and a 32×32 pixel image. In fact, the modern Windows icon standard includes both a 48×48 pixel image and a 256×256 image, which can be sized as needed for other purposes. If Icon is a null reference, the window is given the same icon as the application (which you can set in Visual Studio by double-clicking the Properties node in the Solution Explorer and then choosing the Application tab). If this is omitted, WPF will use a standard but unremarkable icon that shows a window.
Top and Left	Set the distance between the top-left corner of the window and the top and left edges of the screen, in device-independent pixels. The LocationChanged event fires when either of these details changes. If the WindowStartupPosition property is set to Manual, you can set these properties before the window appears to set its position. You can always use these properties to move the position of a window *after* it has appeared, no matter what value you use for WindowStartupPosition.
ResizeMode	Takes a value from the ResizeMode enumeration that determines whether the user can resize the window. This setting also affects the visibility of the maximize and minimize boxes. Use NoResize to lock a window's size completely, CanMinimize to allow minimizing only, CanResize to allow everything, or CanResizeWithGrip to add a visual detail at the bottom-right corner of the window to show that the window is resizable.
RestoreBounds	Gets the bounds of the window. However, if the window is currently maximized or minimized, this property provides the bounds that were last used before the window was maximized or minimized. This is extremely useful if you need to store the position and dimensions of a window, as described later in this chapter.
ShowInTaskbar	If set to true, the window appears in the taskbar and the Alt+Tab list. Usually, you will set this to true only for your application's main window.

SizeToContent	Allows you to create a window that enlarges itself automatically. This property takes a value from the SizeToContent enumeration. Use Manual to disable automatic sizing; or use Height, Width, or WidthAndHeight to allow the window to expand in different dimensions to accommodate dynamic content. When using SizeToContent, the window may be sized larger than the bounds of the screen.
Title	The caption that appears in the title bar for the window (and in the taskbar).
Topmost	When set to true, this window is always displayed on top of every other window in your application (unless these other windows also have TopMost set to true). This is a useful setting for palettes that need to "float" above other windows.
WindowStartupLocation	Takes a value from the WindowStartupLocation enumeration. Use Manual to position a window exactly with the Left and Top properties, CenterScreen to place the window in the center of the screen, or CenterOwner to center the window with respect to the window that launched it. When showing a modeless window with CenterOwner, make sure you set the Owner property of the new window before you show it.
WindowState	Takes a value from the WindowState enumeration. Informs you (and allows you to change) whether the window is currently maximized, minimized, or in its normal state. The StateChanged event fires when this property changes.
WindowStyle	Takes a value from the WindowStyle enumeration, which determines the border for the window. Your options include SingleBorderWindow (the default), None (a very thin raised border with no title bar region), and two other choices that are largely obsolete (ThreeDBorderWindow and ToolWindow), especially if you plan to run your application on Windows 8.

You've already learned about the lifetime events that fire when a window is created, activated, and unloaded (in Chapter 5). In addition, the Window class includes LocationChanged and WindowStateChanged events, which fire when the window's position and WindowState change, respectively.

Showing a Window

To display a window, you need to create an instance of the Window class and use the Show() or ShowDialog() method.

The ShowDialog() method shows a *modal* window. Modal windows stop the user from accessing the parent window by blocking any mouse or keyboard input to it, until the modal window is closed. In addition, the ShowDialog() method doesn't return until the modal window is closed, so any code that you've placed after the ShowDialog() call is put on hold. (However, that doesn't mean other code can't run—for example, if you have a timer running, its event handler will still run.) A common pattern in code is to show a modal window, wait until it's closed, and then act on its data.

Here's an example that uses the ShowDialog() method:

```
TaskWindow winTask = new TaskWindow();
winTask.ShowDialog();
// Execution reaches this point after winTask is closed.
```

The Show() method shows a *modeless* window, which doesn't block the user from accessing any other window. The Show() method also returns immediately after the window is shown, so subsequent code statements are executed immediately. You can create and show several modeless windows, and the user can interact with them all at once. When using modeless windows, synchronization code is sometimes

required to make sure that changes in one window update the information in another window to prevent a user from working with invalid information.

Here's an example that uses the Show() method:

```
MainWindow winMain = new MainWindow();
winMain.Show();
// Execution reaches this point immediately after winMain is shown.
```

Modal windows are ideal for presenting the user with a choice that needs to be made before an operation can continue. For example, consider Microsoft Word, which shows its Options and Print windows modally, forcing you to make a decision before continuing. On the other hand, the windows used to search for text or check the spelling in a document are shown modelessly, allowing the user to edit text in the main document window while performing the task.

Closing a window is equally easy, using the Close() method. Alternatively, you can hide a window from view using Hide() or by setting the Visibility property to Hidden. Either way, the window remains open and available to your code. Generally, it makes sense to hide only modeless windows. That's because if you hide a modal window, your code remains stalled until the window is closed, and the user can't close an invisible window.

Positioning a Window

Usually, you won't need to position a window exactly on the screen. You'll simply use CenterOwner for the WindowState and forget about the whole issue. In other, less common cases, you'll use Manual for the Windows state and set an exact position using the Left and Right properties.

Sometimes you need to take a little more care in choosing an appropriate location and size for your window. For example, you could accidentally create a window that is too large to be accommodated on a low-resolution display. If you are working with a single-window application, the best solution is to create a resizable window. If you are using an application with several floating windows, the answer is not as simple.

You could just restrict your window positions to locations that are supported on even the smallest monitors, but that's likely to frustrate higher-end users (who have purchased better monitors for the express purpose of fitting more information on their screen at a time). In this case, you usually want to make a runtime decision about the best window location. To do this, you need to retrieve some basic information about the available screen real estate using the System.Windows.SystemParameters class.

The SystemParameters class consists of a huge list of static properties that return information about various system settings. For example, you can use the SystemParameters class to determine whether the user has enabled hot tracking and the "drag full windows" option, among many others. With windows, the SystemParameters class is particularly useful because it provides two properties that give the dimensions of the current screen: FullPrimaryScreenHeight and FullPrimaryScreenWidth. Both are quite straightforward, as this bit of code (which centers the window at runtime) demonstrates:

```
double screeHeight = SystemParameters.FullPrimaryScreenHeight;
double screeWidth = SystemParameters.FullPrimaryScreenWidth;
this.Top = (screenHeight - this.Height) / 2;
this.Left = (screenWidth - this.Width) / 2;
```

Although this code is equivalent to using CenterScreen for the WindowState property of the window, it gives you the flexibility to implement different positioning logic and to run this logic at the appropriate time.

An even better choice is to use the SystemParameters.WorkArea rectangle to center the window in the *available* screen area. The work area measurement doesn't include the area where the taskbar is docked (and any other "bands" that are docked to the desktop).

```
double workHeight = SystemParameters.WorkArea.Height;
double workWidth = SystemParameters.WorkArea.Width;
this.Top = (workHeight - this.Height) / 2;
this.Left = (workWidth - this.Width) / 2;
```

■ **Note** Both window-positioning examples have one minor drawback. When the Top property is set on a window that's already visible, the window is moved and refreshed immediately. The same process happens when the Left property is set in the following line of code. As a result, keen-eyed users may see the window move twice. Unfortunately, the Window class does not provide a method that allows you to set both position properties at once. The only solution is to position the window after you create it but before you make it visible by calling Show() or ShowDialog().

Saving and Restoring Window Location

A common requirement for a window is to remember its last location. This information can be stored in a user-specific configuration file or in the Windows registry.

If you wanted to store the position of an important window in a user-specific configuration file, you would begin by double-clicking the Properties node in the Solution Explorer and choosing the Settings section. Then, add a user-scoped setting with a data type of System.Windows.Rect, as shown in Figure 23-1.

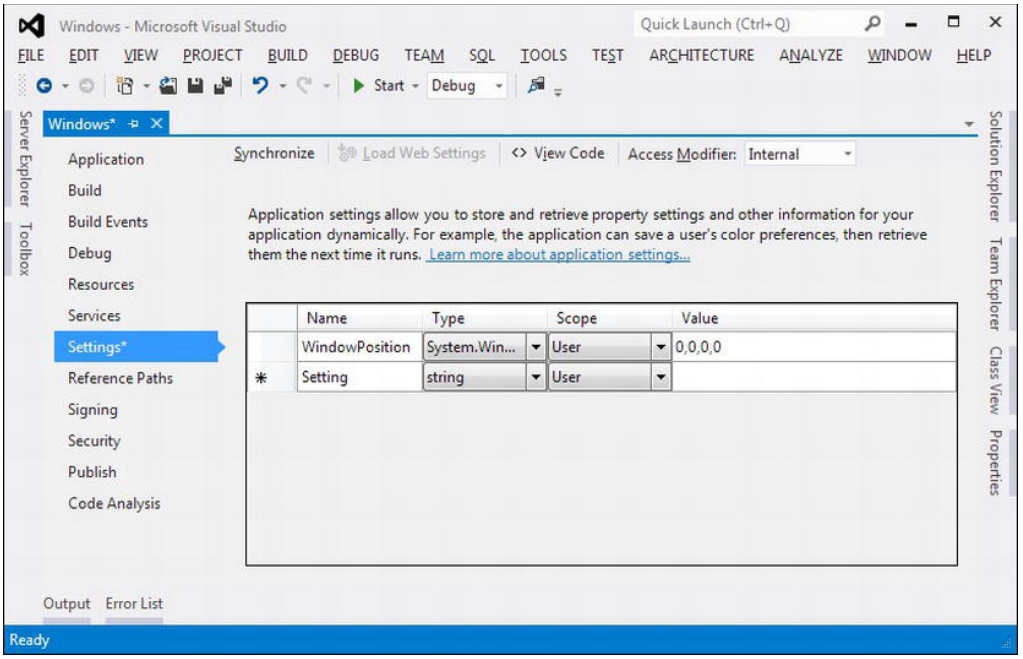

Figure 23-1. *A property for storing a window's position and size*

With this setting in place, it's easy to create code that automatically stores information about a window's size and position, as shown here:

```
Properties.Settings.Default.WindowPosition = win.RestoreBounds;
Properties.Settings.Default.Save();
```

Notice that this code uses the RestoreBounds property, which gives the correct dimensions (the last nonmaximized, nonminimized size), even if the window is currently maximized or minimized.

It's just as easy to retrieve this information when you need it:

```
try
{
    Rect bounds = Properties.Settings.Default.WindowPosition;
    win.Top = bounds.Top;
    win.Left = bounds.Left;

    // Restore the size only for a manually sized
    // window.
    if (win.SizeToContent == SizeToContent.Manual)
    {
        win.Width = bounds.Width;
        win.Height = bounds.Height;
    }
}
catch
{
    MessageBox.Show("No settings stored.");
}
```

The only limitation to this approach is that you need to create a separate property for each window that you want to store. If you need to store the position of many different windows, you might want to design a more flexible system. For example, the following helper class stores a position for any window you pass in, using a registry key that incorporates the name of that window. (You could use additional identifying information if you want to store the settings for several windows that will have the same name.)

```
public class WindowPositionHelper
{
    public static string RegPath = @"Software\MyApp\WindowBounds\";

    public static void SaveSize(Window win)
    {
        // Create or retrieve a reference to a key where the settings
        // will be stored.
        RegistryKey key;
        key = Registry.CurrentUser.CreateSubKey(RegPath + win.Name);

        key.SetValue("Bounds", win.RestoreBounds.ToString());
        key.SetValue("Bounds",
          win.RestoreBounds.ToString(CultureInfo.InvariantCulture));
    }

    public static void SetSize(Window win)
    {
```

```
        RegistryKey key;
        key = Registry.CurrentUser.OpenSubKey(RegPath + win.Name);

        if (key != null)
        {
            Rect bounds = Rect.Parse(key.GetValue("Bounds").ToString());
            win.Top = bounds.Top;
            win.Left = bounds.Left;

            // Restore the size only for a manually sized
            // window.
            if (win.SizeToContent == SizeToContent.Manual)
            {
                win.Width = bounds.Width;
                win.Height = bounds.Height;
            }
        }
    }
}
```

To use this class in a window, you call the SaveSize() method when the window is closing and call the SetSize() method when the window is first opened. In each case, you pass a reference to the window you want the helper class to inspect. Note that in this example, each window must have a different value for its Name property.

Window Interaction

In Chapter 7, you considered the WPF application model, and you took your first look at how windows interact. As you saw there, the Application class provides you with two tools for getting access to other windows: the MainWindow and Windows properties. If you want to track windows in a more customized way—for example, by keeping track of instances of a certain window class, which might represent documents—you can add your own static properties to the Application class.

Of course, getting a reference to another window is only half the battle. You also need to decide how to communicate. As a general rule, you should minimize the need for window interactions, because they complicate code unnecessarily. If you do need to modify a control in one window based on an action in another window, create a dedicated method in the target window. That makes sure the dependency is well identified, and it adds another layer of indirection, making it easier to accommodate changes to the window's interface.

■ **Tip** If the two windows have a complex interaction, are developed or deployed separately, or are likely to change, you can consider going one step further and formalize their interaction by creating an interface with the public methods and implementing that interface in your window class.

Figures 23-2 and 23-3 show two examples for implementing this pattern. Figure 23-2 shows a window that triggers a second window to refresh its data in response to a button click. This window does not directly attempt to modify the second window's user interface; instead, it relies on a custom intermediate method called DoUpdate().

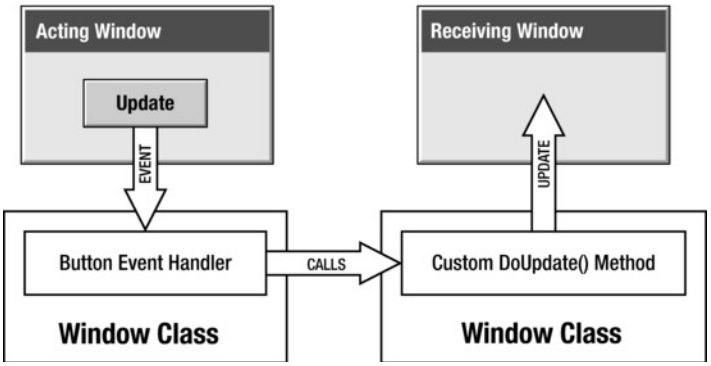

Figure 23-2. *A single window interaction*

The second example, Figure 23-3, shows a case where more than one window needs to be updated. In this case, the acting window relies on a higher-level application method, which calls the required window update methods (perhaps by iterating through a collection of windows). This approach is better because it works at a higher level. In the approach shown in Figure 23-2, the acting window doesn't need to know anything specific about the controls in the receiving window. The approach in Figure 23-3 goes one step further—the acting window doesn't need to know anything at all about the receiving window class.

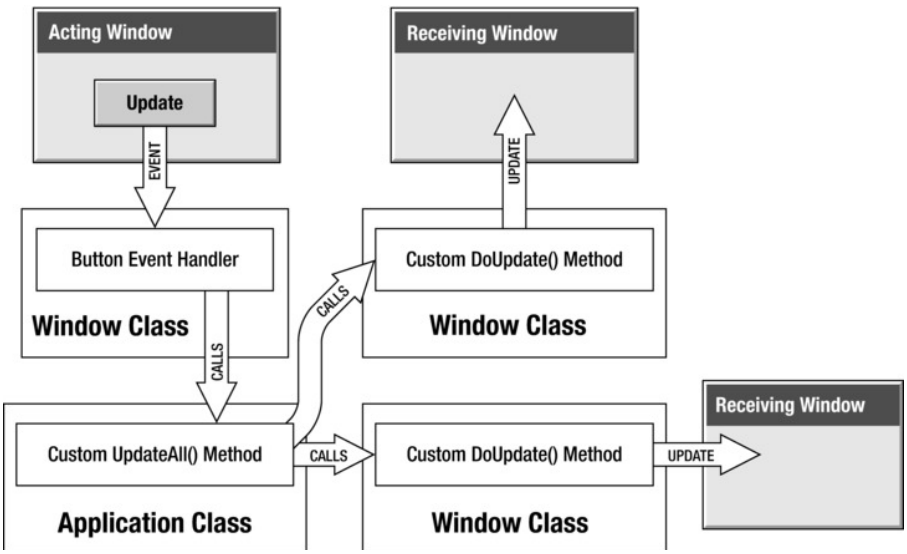

Figure 23-3. *A one-to-many window interaction*

■ **Tip** When interacting between windows, the Window.Activate() method often comes in handy. It transfers the activation to the window you want. You can also use the Window.IsActive property to test whether a window is currently the one and only active window.

You can go one step further in decoupling this example. Rather than having the Application class trigger a method in the various windows, it could simply fire an event and allow the windows to choose how to respond to that event.

■ **Note** WPF can help you abstract your application logic through its support for commands, which are application-specific tasks that can be triggered any way you like. Chapter 9 has the full story.

The examples in Figure 23-2 and Figure 23-3 show how separate windows (usually modeless) can trigger actions in one another. But certain other patterns for window interaction are simpler (such as the dialog model) and supplement this model (such as window ownership). You'll consider these features in the following sections.

Window Ownership

.NET allows a window to "own" other windows. Owned windows are useful for floating toolbox and command windows. One example of an owned window is the Find and Replace window in Microsoft Word. When an owner window is minimized, the owned windows are also minimized automatically. When an owned window overlaps its owner, it is always displayed on top.

To support window ownership, the Window class adds two properties. Owner is a reference that points to the window that owns the current window (if there is one). OwnedWindows is a collection of all the windows that the current window owns (if any).

Setting up ownership is simply a matter of setting the Owner property, as shown here:

```
// Create a new window.
ToolWindow winTool = new ToolWindow();

// Designate the current window as the owner.
winTool.Owner = this;

// Show the owned window.
winTool.Show();
```

Owned windows are always shown modelessly. To remove an owned window, set the Owner property to null.

■ **Note** WPF does not include a system for building multiple document interface (MDI) applications. If you want more sophisticated window management, it's up to you to build it (or buy a third-party component).

An owned window can own another window, which can own another window, and so forth (although it's questionable whether this design has any practical use). The only limitations are that a window cannot own itself and two windows cannot own each other.

The Dialog Model

Often, when you show a window modally, you are offering the user some sort of choice. The code that displays the window waits for the result of that choice and then acts on it. This design is known as the *dialog model*. The window you show modally is the dialog box.

You can easily accommodate this design pattern by creating some sort of public property in your dialog window. When the user makes a selection in the dialog window, you would set this property and then close the window. The code that shows the dialog box can then check for this property and determine what to do next based on its value. (Remember that even when a window is closed, the window object, and all its control information, still exists until the variable referencing it goes out of scope.)

Fortunately, some of this infrastructure is already hardwired into the Window class. Every window includes a ready-made DialogResult property, which can take a true, false, or null value. Usually, true indicates the user chose to go forward (for example, clicked OK), while false indicates that the user canceled the operation.

Best of all, once you set the dialog result, it's returned to the calling code as the return value of the ShowDialog() method. That means you can create, show, and consider the result of a dialog box window with this lean code:

```
DialogWindow dialog = new DialogWindow();
if (dialog.ShowDialog() == true)
{
    // The user accepted the action. Full speed ahead.
}
else
{
    // The user canceled the action.
}
```

■ **Note** Using the DialogResult property doesn't prevent you from adding custom properties to your window. For example, it's perfectly reasonable to use the DialogResult property to inform the calling code whether an action was accepted or canceled and to provide other important details through custom properties. If the calling code finds a DialogResult of true, it can then check these other properties to get the information it needs.

You can take advantage of another shortcut. Rather than setting the DialogResult by hand after the user clicks a button, you can designate a button as the accept button (by setting IsDefault to true). Clicking that button automatically sets the DialogResult of the window to true. Similarly, you can designate a button as the cancel button (by setting IsCancel to true), in which case clicking it will set the DialogResult to Cancel. (You learned about IsDefault and IsCancel when you considered buttons in Chapter 6.)

Common Dialog Boxes

The Windows operating system includes many built-in dialog boxes that you can access through the Windows API. WPF provides wrappers for just a few of these.

■ **Note** There are good reasons that WPF doesn't include wrappers for all the Windows APIs. One of the goals of WPF is to decouple it from the Windows API so it's usable in other environments (like a browser) or portable to other platforms. Also, many of the built-in dialog boxes are showing their age and shouldn't be the first choice for modern applications.

The most obvious of these is the System.Windows.MessageBox class, which exposes a static Show() method. You can use this code to display a standard Windows message box. Here's the most common overload:

```
MessageBox.Show("You must enter a name.", "Name Entry Error",
  MessageBoxButton.OK, MessageBoxImage.Exclamation) ;
```

The MessageBoxButton enumeration allows you to choose the buttons that are shown in the message box. Your options include OK, OKCancel, YesNo, and YesNoCancel. (The less user-friendly AbortRetryIgnore isn't supported.) The MessageBoxImage enumeration allows you to choose the message box icon (Information, Exclamation, Error, Hand, Question, Stop, and so on).

Along with the MessageBox class, WPF includes specialized printing support that uses the PrintDialog (which is described in Chapter 29) and, in the Microsoft.Win32 namespace, the OpenFileDialog and SaveFileDialog classes.

The OpenFileDialog and SaveFileDialog classes acquire some additional features (some which are inherited from the FileDialog class). Both support a filter string, which sets the allowed file extensions. The OpenFileDialog class also provides properties that let you validate the user's selection (CheckFileExists) and allow multiple files to be selected (Multiselect). Here's an example that shows an OpenFileDialog and displays the selected files in a list box after the dialog box is closed:

```
OpenFileDialog myDialog = new OpenFileDialog();

myDialog.Filter = "Image Files(*.BMP;*.JPG;*.GIF)|*.BMP;*.JPG;*.GIF" +
  "|All files (*.*)|*.*";
myDialog.CheckFileExists = true;
myDialog.Multiselect = true;

if (myDialog.ShowDialog() == true)
{
    lstFiles.Items.Clear();
    foreach (string file in myDialog.FileNames)
    {
        lstFiles.Items.Add(file);
    }
}
```

You won't find any color pickers, font pickers, or folder browsers (although you can get these ingredients using the System.Windows.Forms classes from .NET 2.0).

Nonrectangular Windows

Irregularly shaped windows are often used in consumer applications such as photo editors, movie makers, and MP3 players. Creating a basic shaped window in WPF is easy. However, creating a slick, professional-looking shaped window takes more work—and, most likely, a talented graphic designer to create the outlines and design the background art.

A Simple Shaped Window

The basic technique for creating a shaped window is to follow these steps:

1. Set the Window.AllowsTransparency property to true.

2. Set the Window.WindowStyle property to None to hide the nonclient region of the window (the blue border). If you don't, you'll get an InvalidOperationException when you attempt to show the window.
3. Set the Background to be transparent (using the color Transparent, which has an alpha value of 0). Or set the Background to use an image that has transparent areas (regions that are painted with an alpha value of 0).

These three steps effectively remove the standard window appearance (known to WPF experts as the window *chrome*). To get the shaped window effect, you now need to supply some nontransparent content that has the shape you want. You have a number of options:

- Supply background art, using a file format that supports transparency. For example, you can use a PNG file to supply the background of a window. This is a simple, straightforward approach, and it's suitable if you're working with designers who have no knowledge of XAML. However, because the window will be rendered with more pixels at higher system DPIs, the background graphic may become blurry. This is also a problem if you choose to allow the user to resize the window.
- Use the shape-drawing features in WPF to create your background with vector content. This approach ensures that you won't lose quality regardless of the window size and system DPI setting. However, you'll probably want to use a XAML-capable design tool like Expression Blend.
- Use a simpler WPF element that has the shape you want. For example, you can create a nicely rounded window edge with the Border element. This gives you a modern Office-style window appearance with no design work.

Here's a bare-bones transparent window that uses the first approach and supplies a PNG file with transparent regions:

```
<Window x:Class="Windows.TransparentBackground" ...
    WindowStyle="None" AllowsTransparency="True"
    >
  <Window.Background>
    <ImageBrush ImageSource="squares.png"></ImageBrush>
  </Window.Background>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
      </Grid.RowDefinitions>
      <Button Margin="20">A Sample Button</Button>
      <Button Margin="20" Grid.Row="2">Another Button</Button>
    </Grid>
</Window>
```

Figure 23-4 shows this window with a Notepad window underneath. Not only does the shaped window (which consists of a circle and square) leave gaps through which you can see the content underneath, but also some buttons drift off the image and into the transparent region, which means they appear to be floating without a window.

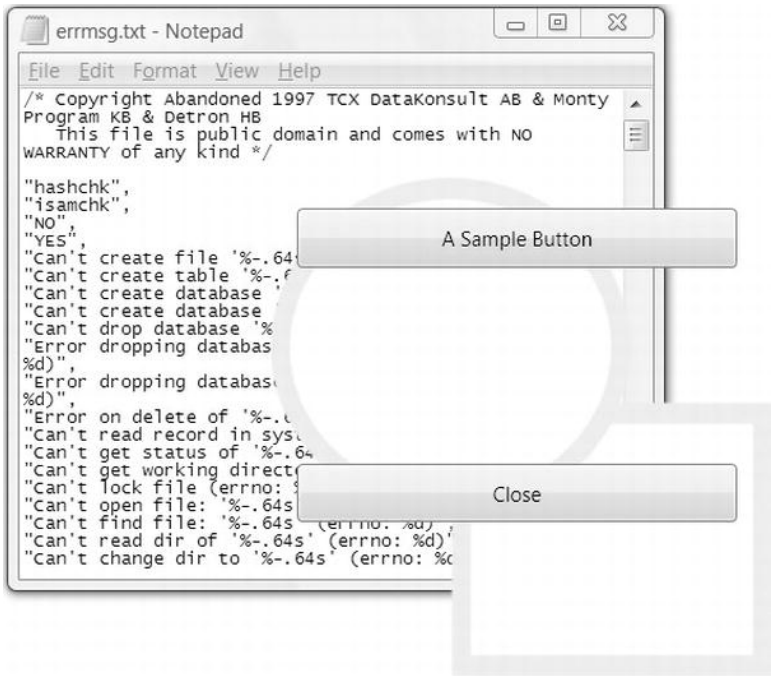

Figure 23-4. *A shaped window that uses a background image*

■ **Note** WPF can perform anti-aliasing between the background of your window and the content underneath, ensuring that your window gets a clean, smoothed edge.

Figure 23-5 shows another, subtler shaped window. This window uses a rounded Border element to give a distinctive look. The layout is also simplified, because there's no way your content could accidentally leak outside the border, and the border can be easily resized with no Viewbox required.

Figure 23-5. *A shaped window that uses a Border*

This window holds a Grid with three rows, which are used for the title bar, the footer bar, and all the content in between. The content row holds a second Grid, which sets a different background and holds any other elements you want (currently, it holds just a single TextBlock).

Here's the markup that creates the window:

```
<Window x:Class="Windows.ModernWindow" ...
    AllowsTransparency="True" WindowStyle="None"
    Background="Transparent"
    >
  <Border Width="Auto" Height="Auto" Name="windowFrame"
    BorderBrush="#395984" BorderThickness="1"
    CornerRadius="0,20,30,40" >
    <Border.Background>
      <LinearGradientBrush>
        <GradientBrush.GradientStops>
          <GradientStopCollection>
            <GradientStop Color="#E7EBF7" Offset="0.0"/>
            <GradientStop Color="#CEE3FF" Offset="0.5"/>
            </GradientStopCollection>
        </GradientBrush.GradientStops>
      </LinearGradientBrush>
    </Border.Background>

    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
      </Grid.RowDefinitions>
```

```
    <TextBlock Text="Title Bar" Margin="1" Padding="5"></TextBlock>

     <Grid Grid.Row="1" Background="#B5CBEF">
       <TextBlock VerticalAlignment="Center" HorizontalAlignment="Center"
        Foreground="White" FontSize="20">Content Goes Here</TextBlock>
     </Grid>

     <TextBlock Grid.Row="2" Text="Footer" Margin="1,10,1,1" Padding="5"
      HorizontalAlignment="Center"></TextBlock>
   </Grid>
  </Border>
</Window>
```

To complete this window, you would want to create buttons that mimic the standard maximize, minimize, and close buttons in the top-right corner.

A Transparent Window with Shaped Content

In most cases, WPF windows won't use fixed graphics to create shaped windows. Instead, they'll use a completely transparent background and then place shaped content on this background. (You can see how this works by looking at the button in Figure 23-4, which is hovering over a completely transparent region.)

The advantage of this approach is that it's more modular. You can assemble a window out of many separate components, all of which are first-class WPF elements. But more important, this allows you to take advantage of other WPF features to build truly dynamic user interfaces. For example, you might assemble shaped content that can be resized, or use animation to produce perpetually running effects right in your window. This isn't as easy if your graphics are provided in a single static file.

Figure 23-6 shows an example. Here, the window contains a Grid with one cell. Two elements share that cell. The first element is a Path that draws the shaped window border and gives it a gradient fill. The other element is a layout container that holds the content for the window, which overlays the Path. In this case, the layout container is a StackPanel, but you could also use something else (such as another Grid or a Canvas for coordinate-based absolute positioning). This StackPanel holds the close button (with the familiar X icon) and the text.

Figure 23-6. *A shaped window that uses a Path*

■ **Note** Even though Figure 23-4 and Figure 23-6 show different examples, they are interchangeable. In other words, you could create either one using the background-based approach or the shape-drawing approach. However, the shape-drawing approach gives you more abilities if you want to dynamically change the shape later and gives you the best quality if you need to resize the window.

The key piece of this example is the Path element that creates the backgrounds. It's a simple vector-based shape that's composed of a series of lines and arcs. Here's the complete markup for the Path:

```
<Path Stroke="DarkGray" StrokeThickness="2">
  <Path.Fill>
    <LinearGradientBrush StartPoint="0.2,0" EndPoint="0.8,1" >
      <LinearGradientBrush.GradientStops>
        <GradientStop Color="White" Offset="0"></GradientStop>
        <GradientStop Color="White" Offset="0.45"></GradientStop>
        <GradientStop Color="LightBlue" Offset="0.9"></GradientStop>
        <GradientStop Color="Gray" Offset="1"></GradientStop>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Path.Fill>

  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="20,0" IsClosed="True">
          <LineSegment Point="140,0"/>
          <ArcSegment Point="160,20" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="160,60"/>
          <ArcSegment Point="140,80" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="70,80"/>
          <LineSegment Point="70,130"/>
          <LineSegment Point="40,80"/>
          <LineSegment Point="20,80"/>
          <ArcSegment Point="0,60" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="0,20"/>
          <ArcSegment Point="20,0" Size="20,20" SweepDirection="Clockwise"/>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

Currently, the Path is fixed in size (as is the window), although you could make it resizable by hosting it in the Viewbox container that you learned about in Chapter 12. You could also improve this example by giving the close button a more authentic appearance—probably a vector X icon that's drawn on a red surface. Although you could use a separate Path element to represent a button and handle the button's mouse events, it's better to change the standard Button control using a control template (as described in Chapter 17). You can then make the Path that draws the X icon part of your customized button.

Moving Shaped Windows

One limitation of shaped forms is that they omit the nonclient title bar portion, which allows the user to easily drag the window around the desktop. Fortunately, WPF allows you to initiate window-dragging mode at any time by calling the Window.DragMove() method.

So, to allow the user to drag the shaped form you saw in the previous examples, you simply need to handle the MouseLeftButtonDown event for the window (or an element on the window, which will then play the same role as the title bar):

```
<TextBlock Text="Title Bar" Margin="1" Padding="5"
 MouseLeftButtonDown="titleBar_MouseLeftButtonDown"></TextBlock>
```

In your event handler, you need only a single line of code:

```
private void titleBar_MouseLeftButtonDown(object sender,
  MouseButtonEventArgs e)
{
    this.DragMove();
}
```

Now the window follows the mouse around the screen, until the user releases the mouse button.

Resizing Shaped Windows

Resizing a shaped window isn't as easy. If your window is roughly rectangular in shape, the easiest approach is to add a sizing grip to the bottom-right corner by setting the Window.ResizeMode property to CanResizeWithGrip. However, the sizing grip placement assumes that your window is rectangular. For example, if you're creating a rounded window effect using a Border object, as shown earlier in Figure 23-5, this technique may work. The sizing grip will appear in the bottom-right corner, and depending how much you've rounded off that corner, it may appear over the window surface where it belongs. But if you've created a more exotic shape, such as the Path shown earlier in Figure 23-6, this technique definitely won't work; instead, it will create a sizing grip that floats in empty space next to the window.

If the sizing grip placement isn't right for your window, or you want to allow the user to size the window by dragging its edges, you'll need to go to a bit more work. You can use two basic approaches. You can use .NET's platform invoke feature (p/invoke) to send a Win32 message that resizes the window. Or you can simply track the mouse position as the user drags to one side, and resize the window manually, by setting its Width property. The following example uses the latter approach.

Before you can use either approach, you need a way to detect when the user moves the mouse over the edges of the window. At this point, the mouse pointer should change to a resize cursor. The easiest way to do this in WPF is to place an element along the edge of each window. This element doesn't need to have any visual appearance—in fact, it can be completely transparent and let the window show through. Its sole purpose is to intercept mouse events.

A 5-unit wide Rectangle is perfect for the task. Here's how you might place a Rectangle that allows right-side resizing in the rounded-edge window shown in Figure 23-5:

```
<Grid>
    ...
    <Rectangle Grid.RowSpan="3" Width="5"
     VerticalAlignment="Stretch" HorizontalAlignment="Right"
     Cursor="SizeWE" Fill="Transparent"
     MouseLeftButtonDown="window_initiateWiden"
     MouseLeftButtonUp="window_endWiden"
```

```
     MouseMove="window_Widen"></Rectangle>
</Grid>
```

The Rectangle is placed in the top row but is given a RowSpan value of 3. That way, it stretches along all three rows and occupies the entire right side of the window. The Cursor property is set to the mouse cursor you want to show when the mouse is over this element. In this case, the "west-east" resize cursor does the trick—it shows the familiar two-way arrow that points left and right.

The Rectangle event handlers toggle the window into resize mode when the user clicks the edge. The only trick is that you need to capture the mouse to ensure you continue receiving mouse events even if the mouse is dragged off the rectangle. The mouse capture is released when the user releases the left mouse button.

```
bool isWiden = false;

private void window_initiateWiden(object sender, MouseEventArgs e)
{
    isWiden = true;
}

private void window_Widen(object sender, MouseEventArgs e)
{
    Rectangle rect = (Rectangle)sender;
    if (isWiden)
    {
        rect.CaptureMouse();
        double newWidth = e.GetPosition(this).X + 5;
        if (newWidth > 0) this.Width = newWidth;
    }
}

private void window_endWiden(object sender, MouseEventArgs e)
{
    isWiden = false;

    // Make sure capture is released.
    Rectangle rect = (Rectangle)sender;
    rect.ReleaseMouseCapture();
}
```

Figure 23-7 shows the code in action.

Figure 23-7. *Resizing a shaped window*

Putting It All Together: A Custom Control Template for Windows

Using the code you've seen so far, you can build a custom-shaped window quite easily. However, if you want to use a new window standard for your entire application, you'll be forced to manually restyle every window with the same shaped border, header region, close buttons, and so on. In this situation, a better approach is to adapt your markup into a control template that you can use on any window. (If you're a bit sketchy on the inner workings of control templates, you might want to review the information in Chapter 17 before you continue with the rest of this section.)

The first step is to consult the default control template for the Window class. For the most part, this template is pretty straightforward, but it includes one detail you might not expect: an AdornerDecorator element. This element creates a special drawing area called the *adorner layer* over the rest of the window's client content. WPF controls can use the adorner layer to draw content that should appear superimposed over your elements. This includes small graphical indicators that show focus, flag validation errors, and guide drag-and-drop operations. When you build a custom window, you need to ensure that the adorner layer is present, so that controls that use it continue to function.

With that in mind, it's possible to identify the basic structure that the control template for a window should take. Here's a standardized example with markup that creates window like the one shown in Figure 23-7:

```
<ControlTemplate x:Key="CustomWindowTemplate" TargetType="{x:Type Window}">
  <Border Name="windowFrame" ... >
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
      </Grid.RowDefinitions>

      <!-- The title bar. -->
      <TextBlock Text="{TemplateBinding Title}"
       FontWeight="Bold"></TextBlock>
```

```
        <Button Style="{StaticResource CloseButton}"
         HorizontalAlignment="Right"></Button>

        <!-- The window content. -->
        <Border Grid.Row="1">
          <AdornerDecorator>
            <ContentPresenter></ContentPresenter>
          </AdornerDecorator>
        </Border>

        <!-- The footer. -->
        <ContentPresenter Grid.Row="2" Margin="10"
         HorizontalAlignment="Center"
         Content="{TemplateBinding Tag}"></ContentPresenter>

        <!-- The resize grip. -->
        <ResizeGrip Name="WindowResizeGrip" Grid.Row="2"
         HorizontalAlignment="Right" VerticalAlignment="Bottom"
         Visibility="Collapsed" IsTabStop="False" />

        <!-- The invisible rectangles that allow dragging to resize. -->
        <Rectangle Grid.Row="1" Grid.RowSpan="3" Cursor="SizeWE"
         VerticalAlignment="Stretch" HorizontalAlignment="Right"
         Fill="Transparent" Width="5"></Rectangle>
        <Rectangle Grid.Row="2" Cursor="SizeNS"
         HorizontalAlignment="Stretch" VerticalAlignment="Bottom"
         Fill="Transparent" Height="5"></Rectangle>
      </Grid>
    </Border>

    <ControlTemplate.Triggers>
      <Trigger Property="ResizeMode" Value="CanResizeWithGrip">
        <Setter TargetName="WindowResizeGrip"
         Property="Visibility" Value="Visible"></Setter>
      </Trigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
```

The top-level element in this template is a Border object for the window frame. Inside that is a Grid with three rows. The contents of the Grid break down as follows:

- The top row holds the title bar, which consists of an ordinary TextBlock that displays the window title and a close button. A template binding pulls the window title from the Window.Title property.
- The middle row holds a nested Border with the rest of the window content. The content is inserted using a ContentPresenter. The ContentPresenter is wrapped in the AdornerDecorator, which ensures that the adorner layer is placed over your element content.
- The third row holds another ContentPresenter. However, this content presenter doesn't use the standard binding to get its content from the Window.Content

property. Instead, it explicitly pulls its content from the Window.Tag property. Usually, this content is just ordinary text, but it could include any element content you want to use.

■ **Note** The Tag property is used because the Window class doesn't include any property that's designed to hold footer text. Another option is to create a custom class that derives from Window and adds a Footer property.

- Also in the third row is a resize grip. A trigger shows the resize grip when the Window.ResizeMode property is set to CanResizeWithGrip.
- Finally, two invisible rectangles run along the right and bottom edge of the Grid (and thus the window). They allow the user to click and drag to resize the window.

Two details that aren't shown here are the relatively uninteresting style for the resize grip (which simply creates a small pattern of dots to use as the resize grip) and the close button (which draws a small *X* on a red square). This markup also doesn't include the formatting details, such as the gradient brush that paints the background and the properties that create a nicely rounded border edge. To see the full markup, refer to the sample code provided for this chapter.

The window template is applied using a simple style. This style also sets three key properties of the Window class that make it transparent. This allows you to create the window border and background using WPF elements.

```
<Style x:Key="CustomWindowChrome" TargetType="{x:Type Window}">
  <Setter Property="AllowsTransparency" Value="True"></Setter>
  <Setter Property="WindowStyle" Value="None"></Setter>
  <Setter Property="Background" Value="Transparent"></Setter>
  <Setter Property="Template"
   Value="{StaticResource CustomWindowTemplate}"></Setter>
</Style>
```

At this point, you're ready to use your custom window. For example, you could create a window like this that sets the style and fills in some basic content:

```
<Window x:Class="ControlTemplates.CustomWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CustomWindowTest" Height="300" Width="300"
  Tag="This is a custom footer"
  Style="{StaticResource CustomWindowChrome}">

  <StackPanel Margin="10">
    <TextBlock Margin="3">This is a test.</TextBlock>
    <Button Margin="3" Padding="3">OK</Button>
  </StackPanel>
</Window>
```

There's just one problem. Currently, the window lacks most of the basic behavior windows require. For example, you can't drag the window around the desktop, resize it, or use the close button. To perform these actions, you need code.

There are two possible ways to add the code you need: you could expand your example into a custom Window-derived class, or you could create a code-behind class for your resource dictionary. The custom

control approach provides better encapsulation and allows you to extend the public interface of your window (for example, adding useful methods and properties that you can use in your application). The code-behind approach is a relatively lightweight alternative that allows you to extend the capabilities of a control template while letting your application continue to use the base control classes. It's the approach that you'll see in this example.

You've already learned how to create a code-behind class for your resource dictionary (see the "User-Selected Skins" section in Chapter 17). Once you've created the code file, it's easy to add the event handling code you need. The only challenge is that your code runs in the resource dictionary object, not inside your window object. That means you can't use the this keyword to access the current window. Fortunately, there's an easy alternative: the FrameworkElement.TemplatedParent property.

For example, to make the window draggable, you need to intercept a mouse event on the title bar and initiate dragging. Here's the revised TextBlock that wires up an event handler when the user clicks with the mouse:

```
<TextBlock Margin="1" Padding="5" Text="{TemplateBinding Title}"
 FontWeight="Bold" MouseLeftButtonDown="titleBar_MouseLeftButtonDown"></TextBlock>
```

Now you can add the following event handler to the code-behind class for the resource dictionary:

```
private void titleBar_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Window win = (Window)
      ((FrameworkElement)sender).TemplatedParent;
    win.DragMove();
}
```

You can add the event handling code for the close button and resize rectangles in the same way. To see the finished resource dictionary markup and code, along with a template that you apply to any window, refer to the download examples for this chapter.

Of course, there's still a lot of polish needed before this window is attractive enough to suit a modern application. But it demonstrates the sequence of steps you need to follow to build a complex control template, with code, and it achieves a result that would have required custom control development in previous user interface frameworks.

Programming the Windows Taskbar

Both Windows 7 and Windows 8 (in desktop mode) use a revamped taskbar that adds several enhanced features—features that aren't available in Windows Vista. WPF does a good job of supporting these features without forcing developers to add unmanaged API calls or rely on separate assemblies.

First, WPF provides basic support for jump lists (the lists that appear when you right-click a taskbar button). Second, WPF lets you to change the taskbar preview image and the taskbar icon that's used for your application. In the following sections, you'll see how to use these features.

■ **Note** You can safely use the Windows taskbar features even in an application that targets Windows Vista. Any markup or code you use to interact with the enhanced Windows 7 and Windows 8 taskbar is harmlessly ignored on Windows Vista. (The same is true if you're targeting .NET 4, and you build an application that can run on Windows XP computers. WPF simply ignores the taskbar features that don't apply.)

Using Jump Lists

Jump lists are the handy mini-menus that pop up when you right-click a taskbar button. They appear for both currently running applications and applications that aren't currently running but have their buttons pinned to the taskbar. Typically, a jump list provides a quick way to open a document that belongs to the appropriate application—for example, to open a recent document in Word or a frequently played song in Windows Media Player. However, some programs use them more creatively to perform application-specific tasks.

Recent Document Support

The taskbar in Windows 7 and Windows 8 adds a jump list to every document-based application, provided that application is registered to handle a specific file type. For example, in Chapter 7, you learned how to build a single-instance application that was registered to handle .testDoc files (called SingleInstanceApplication). When you run this program and right-click its taskbar button, you'll see a list of recently opened documents, as shown in Figure 23-8.

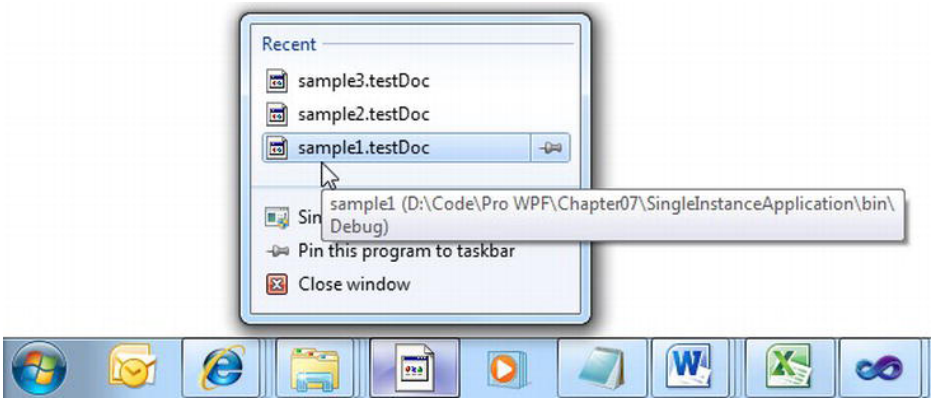

Figure 23-8. *An automatically generated jump list*

If you click one of the recent documents in the jump list, Windows launches another instance of your application and passes the full document path to it as a command-line argument. Of course, you can code around this behavior if it isn't what want. For example, consider the single-instance application from Chapter 7. If you open a document from its jump list, the new instance quietly passes the file path to the currently running application, and then shuts itself down. The end result is that the same application gets to handle all the documents, whether they're opened from inside the application or from the jump list.

As noted, to get the recent document support, your application must be registered to handle the corresponding file type. There are two easy ways to accomplish this. First, you can add the relevant details to the Windows registry using code, as detailed in Chapter 7. Second, you can do it by hand with Windows Explorer. Here's a quick review of what you need to do:

1. Right-click the appropriate file (for example, one with the extension .testDoc).
2. Choose Open With ➤ Choose Default Program to show the Open With dialog box.
3. Click the Browse button, find your WPF application's .exe file, and select it.

4. Optionally, clear the "Always use selected program to open this kind of file" option. Your application doesn't need to be the default application to get jump list support.
5. Click OK.

When registering a file type, you need to keep a few guidelines in mind:

- When creating a file-type registration, you give Windows the exact path to your executable. So do this *after* you place your application somewhere sensible, or you'll need to reregister it every time you move your application file.
- Don't worry about taking over common file types. As long as you don't make your application into the default handler for that file type, you won't change the way Windows works. For example, it's perfectly acceptable to register your application to handle .txt files. That way, when the user opens a .txt file with your application, it appears in your application's recent document list. Similarly, if the user chooses a document from your application's jump list, Windows launches your application. However, if the user double-clicks a .txt file in Windows Explorer, Windows still launches the default application for .txt files (typically Notepad).
- To test jump lists in Visual Studio, you must switch off running the Visual Studio hosting process. If you're running it, Windows will check the file-type registrations for the hosting process (say, YourApp.vshost.exe) instead of your application (YourApp.exe). To avoid this problem, run your compiled application directly from Windows Explorer, or choose Debug ▯ Start Without Debugging. Either way, you won't get debugging support while you're testing the jump list.

■ **Tip** If you want to stop using the Visual Studio hosting process for a longer period of time, you can change your project configuration. Double-click the Properties node in the Solution Explorer. Then choose the Debug tab and clear the check box next to the "Enable the Visual Studio hosting process" setting.

Not only does Windows give your applications the recent document list for free, but it also supports *pinning*, which allows users to attach their most important documents to the jump list and keep them there permanently. As with the jump list for any other application, the user can pin a document by clicking the tiny thumbnail icon. Windows will then move the chosen file to a separate list category, which is called Pinned. Similarly, the user can remove an item from the recent documents list by right-clicking it and choosing Remove.

Custom Jump Lists

The jump list support you've seen so far is built in to Windows, and doesn't require any WPF logic. However, WPF adds on to this support by allowing you to take control of the jump list and fill it with custom items. To do so, you simply add some markup that defines a <JumpList.List> section in your App.xaml file, as shown here:

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

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

  <JumpList.JumpList>
    <JumpList>
    </JumpList>
  </JumpList.JumpList>
</Application>
```

When you define a custom jump list in this way, Windows stops showing the recent document list. To get it back, you need to explicitly opt in with the JumpList.ShowRecentCategory property:

```
<JumpList ShowRecentCategory="True">
```

You can also add the ShowFrequentCategory property to show a list of the most frequently opened documents that your application is registered to handle.

In addition, you can create your own jump list items and place them in a custom category of your choosing. To do so, you must add JumpPath or JumpTask objects to your JumpList. Here's an example of the JumpPath, which represents a document:

```
<JumpList ShowRecentCategory="True">
  <JumpPath CustomCategory="Sample Documents"
   Path="c:\Samples\samples.testDoc"></JumpPath>
</JumpList>
```

When creating a JumpPath, you can supply two details. The CustomCategory property sets the heading that's shown before the item in the jump list. (If you add several items with the same category name, they will be grouped together.) If you don't supply a category, Windows uses the category name Tasks. The Path property is a file path that points to a document. Your path must use a fully qualified file name, the file must exist, and it must be a file type that your application is registered to handle. If you break any of these rules, the item won't appear in the jump list.

Clicking a JumpPath item is exactly the same as clicking one of the files in the recent documents section. When you do, Windows launches a new instance of the application and passes the document path as a command-line argument.

The JumpTask object serves a slightly different purpose. While each JumpPath maps to a document, each JumpTask maps to an *application*. Here's an example that creates a JumpTask for Windows Notepad:

```
<JumpList>
  <JumpTask CustomCategory="Other Programs" Title="Notepad"
   Description="Open a sample document in Notepad"
   ApplicationPath="c:\windows\notepad.exe"
   IconResourcePath="c:\windows\notepad.exe"
   Arguments=" c:\Samples\samples.testDoc "></JumpTask>
  ...
</JumpList>
```

Although a JumpPath requires just two details, a JumpTask uses several more properties. Table 23-2 lists them all.

Table 23-2. *Properties of the JumpTask Class*

Name	Description
Title	The text that appears in the jump list.
Description	The tooltip text that appears when you hover over the item.
ApplicationPath	The executable file for the application. As with the document path in a JumpList, the ApplicationPath property requires a fully qualified path.
IconResourcePath	Points to the file that has the thumbnail icon that Windows will show next to that item in the jump list. Oddly enough, Windows won't choose a default icon or pull it out of the application executable. If you want to see a valid icon, you must set the IconResourcePath.
IconResourceIndex	If the application or icon resource identified by IconResourcePath has multiple icons, you also need to use IconResourceIndex to pick the one you want.
WorkingDirectory	The working directory where the application will be started. Usually, this will be a folder that contains documents for the application.
ApplicationPath	A command-line parameter that you want to pass to the application, such as a file to open.

Creating Jump List Items in Code

Although it's easy to fill a jump list using markup in the App.xaml file, there's a serious disadvantage to this approach. As you've seen, both the JumpPath and the JumpTask items require a fully qualified file path. However, this information often depends on the way the application is deployed, and so it shouldn't be hard-coded. For that reason, it's common to create or modify the application jump list programmatically.

To configure the jump list in code, you use the JumpList, JumpPath, and JumpPath classes from the System.Windows.Shell namespace. The following example demonstrates this technique by creating a new JumpPath object. This item allows the user to open Notepad to view a readme.txt file that's stored in the current application folder, no matter where it's installed.

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    // Retrieve the current jump list.
    JumpList jumpList = new JumpList();
    JumpList.SetJumpList(Application.Current, jumpList);

    // Add a new JumpPath for a file in the application folder.
    string path = Path.GetDirectoryName(
      System.Reflection.Assembly.GetExecutingAssembly().Location);
    path = Path.Combine(path, "readme.txt");
    if (File.Exists(path))
    {
        JumpTask jumpTask = new JumpTask();
        jumpTask.CustomCategory = "Documentation";
        jumpTask.Title = "Read the readme.txt";
        jumpTask.ApplicationPath = @"c:\windows\notepad.exe";
        jumpTask.IconResourcePath = @"c:\windows\notepad.exe";
        jumpTask.Arguments = path;
        jumpList.JumpItems.Add(jumpTask);
```

```
    }

    // Update the jump list.
    jumpList.Apply();
}
```

Figure 23-9 shows a customized jump list that includes this newly added JumpTask.

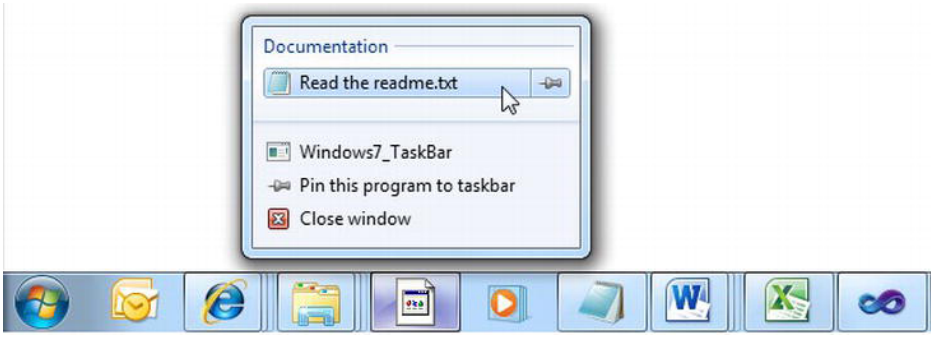

Figure 23-9. *A jump list with a custom JumpTask*

Launching Application Tasks from the Jump List

So far, all the examples you've seen have used the jump list to open documents or launch applications. You haven't seen an example that uses it to trigger a task inside a running application.

This isn't an oversight in the WPF jump list classes—it's just the way jump lists are designed. To work around it, you need to use a variation of the single-instance technique you used in Chapter 7. Here's the basic strategy:

- When the Application.Startup event fires, create a JumpTask that points to your application. Instead of using a file name, set the Arguments property to a special code that your application recognizes. For example, you could set it to @#StartOrder if you wanted this task to pass a "start order" instruction to your application.
- Use the single-instance code from Chapter 7. When a second instance starts, pass the command-line parameter to the first instance and shut down the new application.
- When the first instance receives the command-line parameter (in the OnStartupNextInstance() method), perform the appropriate task.
- Don't forget to remove the tasks from the jump list when the Application.Exit event fires, unless the tasks' commands will work equally well when your application is launched for the first time.

To see a basic implementation of this technique, refer to the JumpListApplicationTask project with the sample code for this chapter.

Changing the Taskbar Icon and Preview

The taskbar in Windows 7 and Windows 8 adds several more refinements, including an optional progress display and thumbnail preview windows. Happily, WPF makes it easy to work with all these features.

To access any of these features, you use TaskbarItemInfo class, which is found in the same System.Windows.Shell namespace as the jump list classes you considered earlier. Every window can have one associated TaskbarItemInfo object, and you can create it in XAML by adding this markup to your window:

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

  <Grid>
    ...
  </Grid>

  <Window.TaskbarItemInfo>
    <TaskbarItemInfo x:Name="taskBarItem"></TaskbarItemInfo>
  </Window.TaskbarItemInfo>
</Window>
```

This step, on its own, doesn't change anything in your window or application. But now you're ready to use the features that are demonstrated in the following sections.

Thumbnail Clipping

Much as Windows gives every application automatic jump list support, it also provides a thumbnail preview window that appears when the user hovers over the taskbar button. Ordinarily, the thumbnail preview window shows a scaled-down version of the client region of the window (everything but the window border). However, in some cases, it might show just a portion of the window. The advantage in this case is that it focuses attention on the relevant part of the window. In a particularly large window, it may make content legible that would otherwise be too small to read.

You can tap into this feature in WPF using the TaskbarItemInfo.ThumbnailClipMargin property. This specifies a Thickness object that sets the margin space between the content you want to show in the thumbnail and the edges of the window. The example shown in Figure 23-10 demonstrates how this works. Every time the user clicks a button in this application, the clipping region is shifted to include just the clicked button. Figure 23-10 shows the preview after clicking the second button.

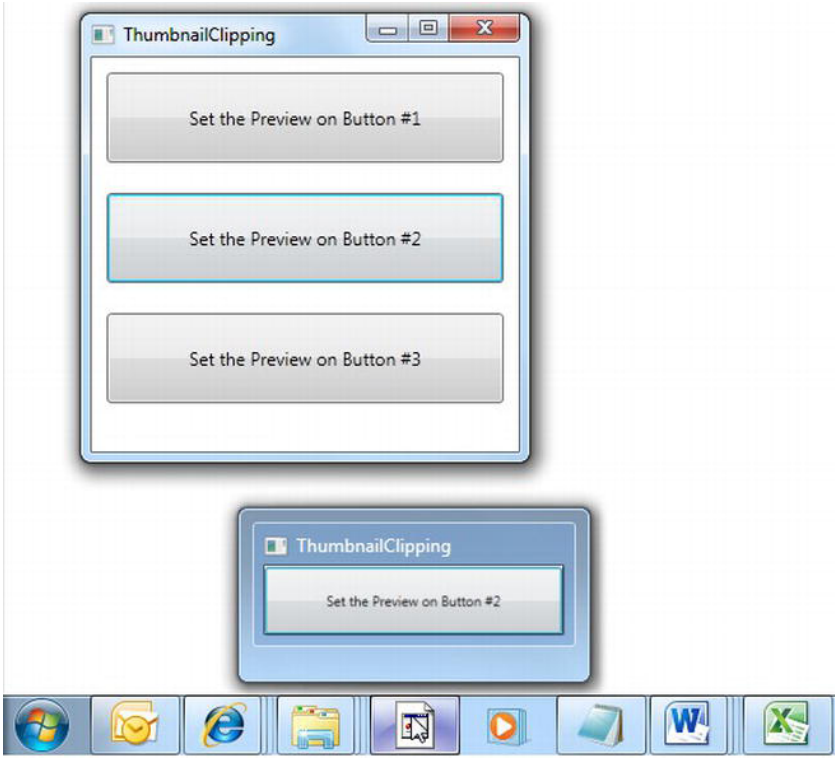

Figure 23-10. *A window that clips its thumbnail preview*

■ **Note** You can't change the thumbnail preview to show a graphic of your choosing. Your only option is to direct it to show a portion of the full window.

To create this effect, the code must take several details into account: the coordinates of the button, its size, and the size of the content region of the window (which, helpfully, is the size of the top-level Grid named LayoutRoot, which sits just inside the window and contains all its markup). Once you have these numbers, a few simple calculations are all you need to constrain the preview to the correct region:

```
private void cmdShrinkPreview_Click(object sender, RoutedEventArgs e)
{
    // Find the position of the clicked button, in window coordinates.
    Button cmd = (Button)sender;
    Point locationFromWindow = cmd.TranslatePoint(new Point(0, 0), this);

    // Determine the width that should be added to every side.
    double left = locationFromWindow.X;
    double top = locationFromWindow.Y;
    double right = LayoutRoot.ActualWidth - cmd.ActualWidth - left;
    double bottom = LayoutRoot.ActualHeight - cmd.ActualHeight - top;
```

```
    // Apply the clipping.
    taskBarItem.ThumbnailClipMargin = new Thickness(left, top, right, bottom);
}
```

Thumbnail Buttons

Some applications use the preview window for an entirely different purpose. They place buttons into a small toolbar area under the preview. Windows Media Player is one example. If you hover over its taskbar icon, you'll get a preview that includes play, pause, forward, and back buttons, which give you a convenient way to control playback without switching to the application itself.

WPF supports thumbnail buttons—in fact, it makes them easy. You simply need to add one or more ThumbButtonInfo objects to the TaskbarItemInfo.ThumbButtonInfos collection. Each ThumbButtonInfo needs an image, which you supply with the ImageSource property. You can also use a Description that adds tooltip text. You can then hook up the button to a method in your application by handling its Click event.

Here's an example that adds Media Player play and pause buttons:

```
<TaskbarItemInfo x:Name="taskBarItem">
  <TaskbarItemInfo.ThumbButtonInfos>
    <ThumbButtonInfo ImageSource="play.png" Description="Play"
     Click="cmdPlay_Click"></ThumbButtonInfo>
    <ThumbButtonInfo ImageSource="pause.png" Description="Pause"
     Click="cmdPause_Click"></ThumbButtonInfo>
  </TaskbarItemInfo.ThumbButtonInfos>
</TaskbarItemInfo>
```

Figure 23-11 shows these buttons under the preview window.

Figure 23-11. *Adding buttons to the thumbnail preview*

■ **Note** Remember that the taskbar buttons are not shown in Windows Vista. For that reason, they should duplicate the functionality that's already in your window, rather than provide new features.

As your application performs different tasks and enters different states, some taskbar buttons may not be appropriate. Fortunately, you can manage them using a small set of useful properties, which are listed in Table 23-3.

Table 23-3. *Properties of the ThumbButtonInfo Class*

Name	Description
ImageSource	Sets the image you want to show for the button, which must be embedded in your application as a resource. Ideally, you'll use a .png file that has a transparent background.
Description	Sets the tooltip text that appears when the user hovers over the button.
Command, CommandParameter, and CommandTarget	Designate a command that the button should trigger. You can use these properties instead of the Click event.
Visibility	Allows you to hide or show a button.
IsEnabled	Allows you to disable a button, so it's visible but can't be clicked.
IsInteractive	Allows you to disable a button without dimming its appearance. This is useful if you want the button to act as a sort of status indicator.
IsBackgroundVisible	Allows you to disable the mouseover feedback for the button. If true (the default), Windows highlights the button and displays a border around it when the mouse moves overtop. If false, it doesn't.
DismissWhenClicked	Allows you to create a single-use button. As soon as it is clicked, Windows removes it from the taskbar. (For more control, you can use code to add or remove buttons on the fly, although it's usually just easier to show and hide them with the Visibility property.)

Progress Notification

If you've ever copied a large file in Windows Explorer, you've seen how it uses progress notification to shade the background of the taskbar button in green. As the copy progresses, the green background fills the button area from left to right, like a progress bar, until the operation completes.

What you might not realize is that this feature isn't specific to Windows Explorer. Instead, it's built into the taskbar and available to all your WPF applications. All you need to do is use two properties of the TaskbarItemInfo class: ProgressValue and ProgressState.

ProgressState starts out at None, which means no progress indicator is shown. However, if you set it to TaskbarItemProgressState.Normal, you get the green-colored progress background that Windows Explorer uses. The ProgressValue property determines its size, from 0 (none) to 1 (full, or complete). For example, setting ProgressValue to 0.5 fills half of the taskbar button's background with a green fill.

The TaskbarItemProgressState enumeration provides a few possibilities other than just None and Normal. You can use Pause to show a yellow background instead of green, Error to show a red background, and Indeterminate to show a continuous, pulsing progress background that ignores the ProgressValue

property. This latter option is suitable when you don't know how long the current task will take to complete (for example, when calling a web service).

Overlay Icons

The final taskbar feature is the taskbar overlay—the ability to add a small image overtop of the taskbar icon. For example, the Messenger chat application uses different overlays to signal different statuses.

To use an overlay, you simply need a very tiny .png or .ico file with a transparent background. You aren't forced to use a specific pixel size, but you'll obviously want an image that's a fair bit smaller than the taskbar button picture. Assuming you've added this image to your project, you can show it by simply setting the TaskBarItemInfo.Overlay property. Most commonly, you'll set it using an image resource that's already defined in your markup, as shown here:

```
taskBarItem.Overlay = (ImageSource)this.FindResource("WorkingImage");
```

Alternatively, you can use the familiar pack URI syntax to point to an embedded file, as shown here:

```
taskBarItem.Overlay = new BitmapImage(
  new Uri("pack://application:,,,/working.png"));
```

Set the Overlay property to a null reference to remove the overlay altogether.

Figure 23-12 shows the pause.png image being used as an overlay over the generic WPF application icon. This indicates that the application's work is currently paused.

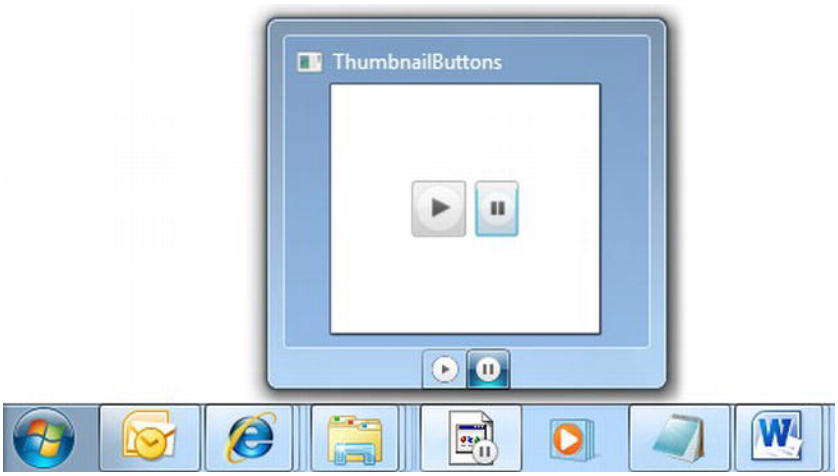

Figure 23-12. *Showing an icon overlay*

The Last Word

In this chapter, you explored the WPF window model. You began with the basics: positioning and sizing a window, creating owned windows, and using the common dialog boxes. Then you considered more sophisticated effects, like irregularly shaped windows, and custom window templates.

In the last part of this chapter, you considered the impressive support that WPF includes for the taskbar in Windows 7 and Windows 8 (in desktop mode). You saw how your WPF application can get basic

jump list support and add custom items. You also learned how you can focus the thumbnail preview window on a section of your window and give it convenient command buttons. Finally, you learned to use progress notification and overlays with your taskbar icon.