If you want to extend the glass edge so that it covers the entire window, simply pass in margin settings of –1 for each side. Figure 23-10 shows the result.

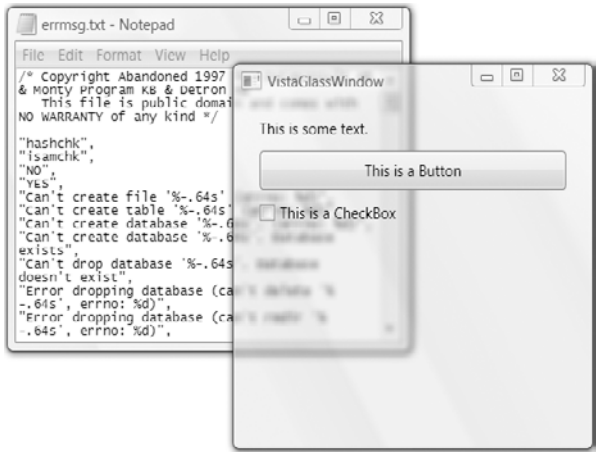

Figure 23-10*. A completely "glassified" window*

When using the Aero glass effect, you need to consider how the appearance of your content will vary when the window is placed over different backgrounds. For example, if you place black text in a glassified region of a window, it will be easier to read on a light background than on a dark background (although it will be legible on both). To improve the readability of text and make your content stand out against a variety of backgrounds, it's common to add some sort of glow effect. For example, black text with a white glow will be equally legible on light and dark backgrounds. Windows Vista includes its own unmanaged function for drawing glowing text, called DrawThemeTextEx(), but there are a variety of native WPF techniques that can give you a similar (or better) result. Two examples include using a fancy brush to paint your text and adding a bitmap effect to your text. (Both techniques are discussed in Chapter 12.)

More DWM Magic

In the Aero glass effect example, you learned how you can create a thicker glass edge (or a completely glassified window) using the DwmExtendFrameIntoClientArea() function. However, DwmExtendFrameIntoClientArea() isn't the only useful function from the Windows API. Several other API functions that start with *Dwm* allow you to interact with the Desktop Window Manager.

For example, you can call DwmIsCompositionEnabled() to check that the Aero glass effect is currently enabled, and you can use DwmEnableBlurBehindWindow() to apply the glass effect to a specific region in a window. There are also a few functions that allow you to get the live thumbnail representation of other applications. For the bare-bones information, check out the MSDN reference for the Desktop Window Manager at `http://tinyurl.com/333glv`.

Programming the Windows 7 Taskbar

Although WPF provides no direct support for Aero glass, it does a much better job with one the key innovations in Windows 7: the revamped taskbar. Not only does WPF provide basic support for jump lists (the list that appears when you right-click a taskbar button), but it also has deep integration for the taskbar features that let you to manage the taskbar icon and configure the thumbnail preview. In the following sections, you'll see how to use these features.

■ **Note** You can safely use the Windows 7 taskbar features in an application that will target earlier versions of Windows. Any markup or code you use to interact with the Windows 7 taskbar is harmlessly ignored on other operating systems.

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

Windows 7 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 23-11.

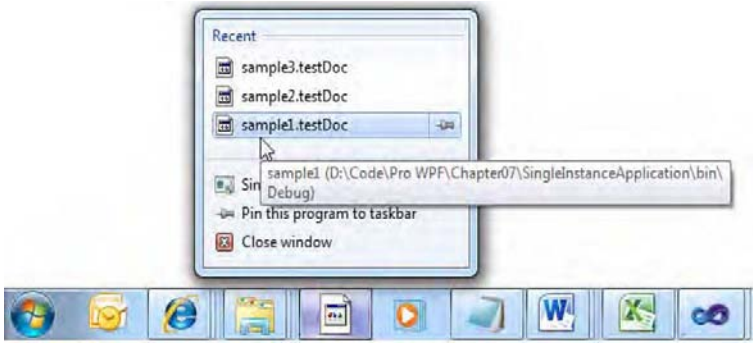

Figure 23-11. *An automatically generated jump list*

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, in order 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 into 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 23-12 shows a customized jump list that includes this newly added JumpTask.

Figure 23-12. *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 Windows 7 taskbar adds several more refinements, including an optional progress display and thumbnail preview windows. Happily, WPF makes it easy to work with all of 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 has the ability to 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 7 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 23-13 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 23-13 shows the preview after clicking the second button.

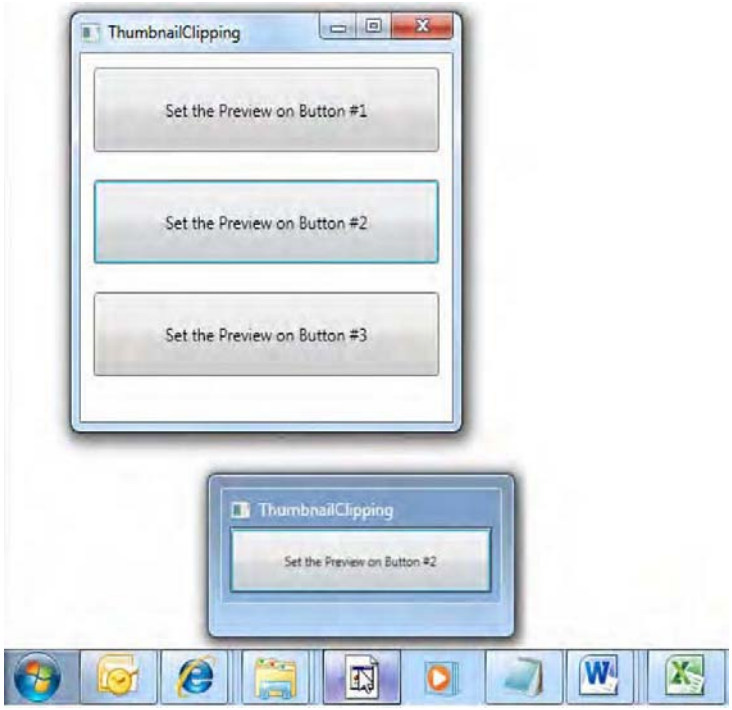

Figure 23-13. *A window that clips its thumbnail preview*

■ **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 of 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 23-14 shows these buttons under the preview window.

Figure 23-14. *Adding buttons to the thumbnail preview*

■ **Note** Remember that the taskbar buttons are shown only in Windows 7. 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.

Name	Description
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 Windows 7 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 that Windows 7 provides 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 23-15 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 23-15. *Showing an icon overlay*

■ **Tip** The overlay icon gives you a great way to provide additional information about your application's current state. In previous versions of Windows, this type of behavior was more common with the icons in the system tray area (the section at the bottom-right corner of the taskbar). In Windows 7, system tray icons are hidden by default, so it makes more sense for long-running applications to stay on the taskbar and use icon overlays.

Getting More Windows 7 Support

Although WPF provides quite a bit of support for Windows 7, there are still a few features that have slipped through the cracks. Fortunately, the Windows API Code Pack, which is available at `http://code.msdn.microsoft.com/WindowsAPICodePack`, provides .NET libraries that wrap *all* the new Windows 7 features.

You can use these libraries with .NET 3.5 SP1 to get basic support for jump lists, icon overlays, progress notification, and thumbnail buttons. More interestingly, you can use them with WPF 4 to get support for a few more features, including the following:

- **Tabbed thumbnails.** These are multiple preview windows, similar to the ones Internet Explorer displays for its individual tabs.
- **The Explorer Browser control.** This allows you to place Windows Explorer-style controls directly inside one of your WPF windows.
- **Task dialogs.** These specialized windows were first introduced with Windows Vista. They give you a simple, polished way to present information and gather basic user input with minimal coding.

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, custom window templates, and Aero glass.

In the last part of this chapter, you considered the impressive support that WPF 4 adds for the Windows 7 taskbar. 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.

CHAPTER 24

Pages and Navigation

Most traditional Windows applications are arranged around a window that contains toolbars and menus. The toolbars and menus *drive* the application—as the user clicks them, actions happen, and other windows appear. In document-based applications, there may be several equally important "main" windows that are open at once, but the overall model is the same. The users spend most of their time in one place, and jump to separate windows when necessary.

Windows applications are so common that it's sometimes hard to imagine different ways to design an application. However, the Web uses a dramatically different page-based navigation model, and desktop developers have realized that it's a surprisingly good choice for designing certain types of applications. In a bid to give desktop developers the ability to build web-like desktop applications, WPF includes its own page-based navigation system. As you'll see in this chapter, it's a remarkably flexible model.

Currently, the page-based model is most commonly used for simple, lightweight applications (or small feature subsets in a more complex window-based application). However, page-based applications are a good choice if you want to streamline application *deployment*. That's because WPF allows you to create a page-based application that runs directly inside Internet Explorer or Firefox. This means that users can run your application without an explicit installation step—they simply point their browsers to the correct location. You'll learn about this model, called XBAP, in this chapter.

Finally, this chapter wraps up with a look at WPF's WebBrowser control, which lets you host HTML web pages in a WPF window. As you'll see, the WebBrowser not only shows web pages, but it also allows you to programmatically explore their structure and content (using the HTML DOM). It even allows your application to interact with JavaScript code.

What's New WPF 3.5 SP1 added the WebBrowser control that's described at the end of this chapter. In earlier versions of WPF, developers needed to use the WebBrowser from the Windows Forms toolkit to get similar functionality.

Understanding Page-Based Navigation

The average web application looks quite a bit different from traditional rich client software. The users of a website spend their time navigating from one page to another. Unless they're unlucky enough to face popup advertising, there's never more than one page visible at a time. When completing a task (such as placing an order or performing a complicated search), the user traverses these pages in a linear sequence from start to finish.

HTML doesn't support the sophisticated windowing capabilities of desktop operating systems, so the best web developers rely on good design and clear, straightforward interfaces. As web design has become increasingly more sophisticated, Windows developers have also begun to see the advantages of this approach. Most important, the web model is simple and streamlined. For that reason, novice users often find websites easier to use than Windows applications, even though Windows applications are obviously much more capable.

In recent years, developers have begun mimicking some of the conventions of the Web in desktop applications. Financial software such as Microsoft Money is a prime example of a web-like interface that leads users through set tasks. However, creating these applications is often more complicated than designing a traditional window-based application, because developers need to re-create basic browser features such as navigation.

■ **Note** In some cases, developers have built web-like applications using the Internet Explorer browser engine. This is the approach that Microsoft Money takes, but it's one that would be more difficult for non-Microsoft developers. Although Microsoft provides hooks into Internet Explorer, such as the WebBrowser control, building a complete application around these features is far from easy. It also risks sacrificing the best capabilities of ordinary Windows applications.

In WPF, there's no longer any reason to compromise, because WPF includes a built-in page model that incorporates navigation. Best of all, this model can be used to create a variety of page-based applications, applications that use some page-based features (for example, in a wizard or help system), or applications that are hosted directly in the browser.

Page-Based Interfaces

To create a page-based application in WPF, you need to stop using the Window class as your top-level container for user interfaces. Instead, it's time to switch to the System.Windows.Controls.Page class.

The model for creating pages in WPF is much the same as the model for creating windows. Although you could create page objects with just code, you'll usually create a XAML file and a code-behind file for each page. When you compile that application, the compiler creates a derived page class that combines your code with a bit of automatically generated glue (such as the fields that refer to each named element on your page). This is the same process that you learned about when you considered compilation with a window-based application in Chapter 2.

■ **Note** You can add a page to any WPF project. Just choose Project ➤ Add Page in Visual Studio.

Although pages are the top-level user interface ingredient when you're designing your application, they aren't the top-level container when you *run* your application. Instead, your pages are hosted in another container. This is the secret to WPF's flexibility with page-based applications, because you can use one of several different containers:

- The NavigationWindow, which is a slightly tweaked version of the Window class
- A Frame that's inside another window
- A Frame that's inside another page
- A Frame that's hosted directly in Internet Explorer or Firefox

You'll consider all of these hosts in this chapter.

A Simple Page-Based Application with NavigationWindow

To try an extremely simple page-based application, create a page like this:

```
<Page x:Class="NavigationApplication.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="Page1"
    >
  <StackPanel Margin="3">
    <TextBlock Margin="3">
      This is a simple page.
    </TextBlock>
    <Button Margin="2" Padding="2">OK</Button>
    <Button Margin="2" Padding="2">Close</Button>
  </StackPanel>
</Page>
```

Now modify the App.xaml file so that the startup page is your page file:

```
<Application x:Class="NavigationApplication.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Page1.xaml"
    >
</Application>
```

When you run this application, WPF is intelligent enough to realize that you're pointing it to a page rather than a window. It automatically creates a new NavigationWindow object to serve as a container and shows your page inside of it (Figure 24-1). It also reads the page's WindowTitle property and uses that for the window caption.

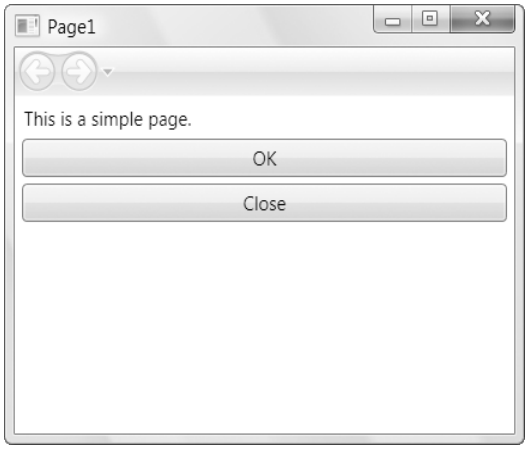

Figure 24-1. *A page in a NavigationWindow*

■ **Note** One difference between a page and a window is that you don't typically set the size of a page, because the page size is determined by the host. If you do set the Width and Height properties of the page, the page is made exactly that size, but some content is clipped if the host window is smaller, or it's centered inside the available space if the host window is larger.

The NavigationWindow looks more or less like an ordinary window, aside from the back and forward navigation buttons that appear in the bar at the top. As you might expect, the NavigationWindow class derives from Window, and it adds a small set of navigation-related properties. You can get a reference to the containing NavigationWindow object using code like this:

```
// Get a reference to the window that contains the current page.
NavigationWindow win = (NavigationWindow)Window.GetWindow(this);
```

This code won't work in the page constructor, because the page hasn't been placed inside its container yet—instead, wait at least until the Page.Loaded event fires.

■ **Tip** It's best to avoid the NavigationWindow approach if at all possible, and use properties of the Page class (and the navigation service described later in this chapter). Otherwise, your page will be tightly coupled to the NavigationWindow, and you won't be able to reuse it in different hosts.

If you want to create a code-only application, you'd need to create both the navigation window and the page to get the effect shown in Figure 24-1. Here's the code that would do it:

```
NavigationWindow win = new NavigationWindow()
win.Content = new Page1();
win.Show();
```

The Page Class

Like the Window class, the Page class allows a single nested element. However, the Page class isn't a content control; it actually derives directly from FrameworkElement. The Page class is also simpler and more streamlined than the Window class. It adds a small set of properties that allow you to customize its appearance, interact with the container in a limited way, and use navigation. Table 24-1 lists these properties.

Table 24-1. *Properties of the Page Class*

Name	Description
Background	Takes a brush that allows you to set the background fill.
Content	Takes the single element that's shown in the page. Usually, this is a layout container, such as a Grid or a StackPanel.
Foreground, FontFamily, and FontSize	Determine the default appearance of text inside the page. The values of these properties are inherited by the elements inside the page. For example, if you set the foreground fill and font size, by default, the content inside the page gets these details.
WindowWidth, WindowHeight, and WindowTitle	Determine the appearance of the window that wraps your page. These properties allow you to take control of the host by setting its width, height, and caption. However, they have an effect only if your page is being hosted in a window (rather than a frame).
NavigationService	Returns a reference to a NavigationService object, which you can use to programmatically send the user to another page.
KeepAlive	Determines whether the page object should be kept alive after the user navigates to another page. You'll take a closer look at this property later in this chapter (in the "Navigation History" section) when you consider how WPF restores the pages in your navigation history.
ShowsNavigationUI	Determines whether the host for this page shows its navigation controls (the forward and back buttons). By default, it's true.
Title	Sets the name that's used for the page in the navigation history. The host does not use the title to set the caption in the title bar; instead, the WindowTitle property serves that purpose.

It's also important to notice what's not there—namely, there's no equivalent of the Hide() and Show() methods of the Window class. If you want to show a different page, you'll need to use navigation.

Hyperlinks

The easiest way to allow the user to move from one page to another is using hyperlinks. In WPF, hyperlinks aren't separate elements. Instead, they're *inline flow elements*, which must be placed inside another element that supports them. (The reason for this design is that hyperlinks and text are often intermixed. You'll learn more about flow content and text layout in Chapter 28.)

For example, here's a combination of text and links in a TextBlock element, which is the most practical container for hyperlinks:

```
<TextBlock Margin="3" TextWrapping="Wrap">
  This is a simple page.
  Click <Hyperlink NavigateUri="Page2.xaml">here</Hyperlink> to go to Page2.
</TextBlock>
```

When rendered, hyperlinks appear as the familiar blue, underlined text (see Figure 24-2).

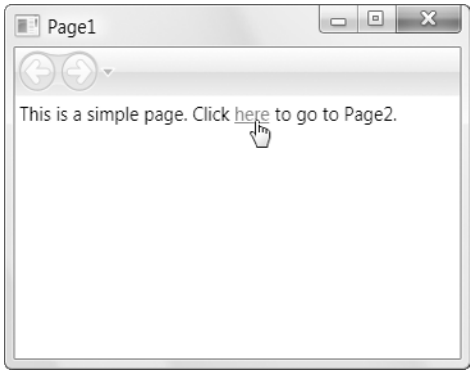

Figure 24-2. *Linking to another page*

You can handle clicks on a link in two ways. You can respond to the Click event and use code to perform some task, or direct the user to another page. However, there's an easier approach. The Hyperlink class also includes a NavigateUri property, which you set to point to any other page in your application. Then, when users click this hyperlink, they travel to the destination page automatically.

■ **Note** The NavigateUri property works only if you place the hyperlink in a page. If you want to use a hyperlink in a window-based application to let users perform a task, launch a web page, or open a new window, you need to handle the RequestNavigate event and write the code yourself.

Hyperlinks aren't the only way to move from one page to another. The NavigationWindow includes prominent forward and back buttons (unless you set the Page.ShowsNavigationUI property to false to hide them). Clicking these buttons moves you through the navigation sequence one page at a time. And similar to a browser, you can click the drop-down arrow at the edge of the forward button to examine the complete sequence and jump forward or backward several pages at a time (Figure 24-3).

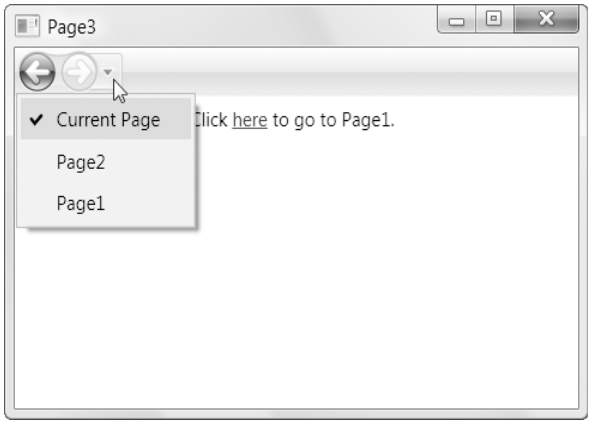

Figure 24-3. *The history of visited pages*

You'll learn more about how the page history works—and its limitations—later in the "Navigation History" section.

■ **Note** If you navigate to a new page, and that page doesn't set the WindowTitle property, the window keeps the title it had on the previous page. If you don't set the WindowTitle on any page, the window caption is left blank.

Navigating to Websites

Interestingly, you can also create a hyperlink that points to web content. When the user clicks the link, the target web page loads in the page area:

```
<TextBlock Margin="3" TextWrapping="Wrap">
  Visit the website
  <Hyperlink NavigateUri="http://www.prosetech.com">www.prosetech.com</Hyperlink>.
</TextBlock>
```

However, if you use this technique, make sure you attach an event handler to the Application.DispatcherUnhandledException or Application.NavigationFailed event. The attempt to navigate to a website could fail if the computer isn't online, the site isn't available, or the web content can't be reached. In this case, the network stack returns an error like "404: File Not Found," which