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, several equally important "main" windows may be 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 weblike 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 also allows you to programmatically explore their structure and content (using the HTML DOM). It even allows your application to interact with JavaScript code.

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 pop-up 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 weblike 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 weblike applications by 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 ä 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 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.

Creating 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.

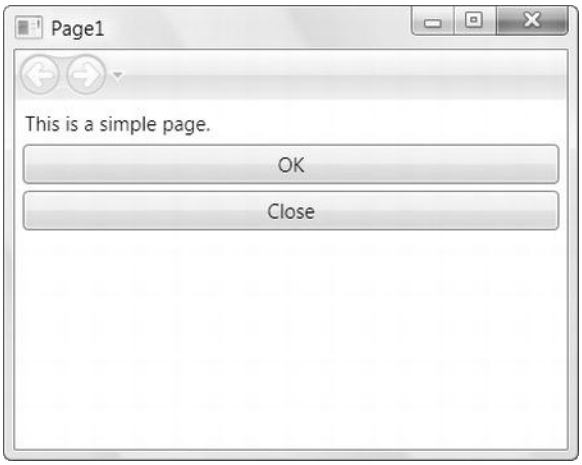

Figure 24-1. *A page in a NavigationWindow*

■ **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 by 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 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 by 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).

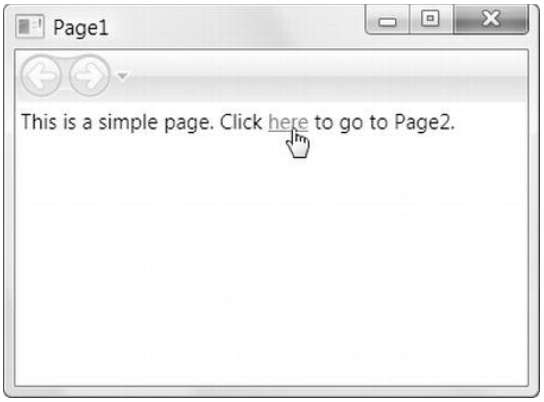

Figure 24-2. *Linking to another page*

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).

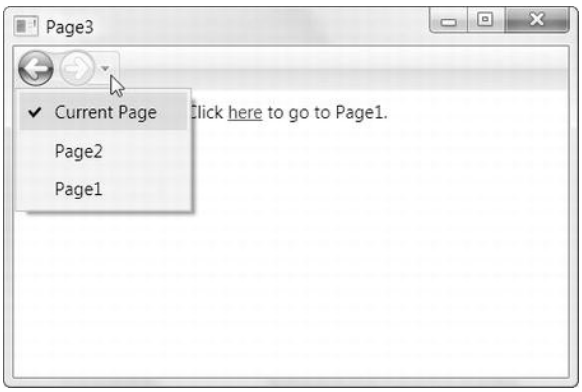

Figure 24-3. *The history of visited pages*

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 such as "404: File Not Found," which becomes a WebException. To handle this exception gracefully and prevent your application from shutting down unexpectedly, you need to neutralize it with an event handler such as this:

```
private void App_NavigationFailed(object sender, NavigationFailedEventArgs e)
{
    if (e.Exception is System.Net.WebException)
    {
        MessageBox.Show("Website " + e.Uri.ToString() + " cannot be reached.");

        // Neutralize the error so the application continues running.
        e.Handled = true;
    }
}
```

NavigationFailed is just one of several navigation events that are defined in the Application class. You'll get the full list later in this chapter, in Table 24-2.

■ **Note** After you lead users to a web page, they'll be able to click its links to travel to other web pages, leaving your content far behind. In fact, they'll return to your WPF page only if they use the navigation history to go back or if you're showing the page in a custom window (as discussed in the next section) and that window includes a control that navigates back to your content.

You can't do a number of things when displaying pages from external websites. You can't prevent the user from navigating to specific pages or sites. Also, you can't interact with the web page by using the HTML Document Object Model (DOM). That means you can't scan a page looking for links or dynamically change a page. All of these tasks are possible using the WebBrowser control, which is described at the end of this chapter.

Fragment Navigation

The last trick that you can use with the hyperlink is *fragment navigation*. By adding the number sign (#) at the end of the NavigateUri, followed by an element name, you can jump straight to a specific control on a page. However, this works only if the target page is scrollable. (The target page is scrollable if it uses the ScrollViewer control or if it's hosted in a web browser.) Here's an example:

```
<TextBlock Margin="3">
  Review the <Hyperlink NavigateUri="Page2.xaml#myTextBox">full text</Hyperlink>.
</TextBlock>
```

When the user clicks this link, the application moves to the page named Page2, and scrolls down the page to the element named myTextBox. The page is scrolled down until myTextBox appears at the top of the page (or as close as possible, depending on the size of the page content and the containing window). However, the target element doesn't receive focus.

Hosting Pages in a Frame

The NavigationWindow is a convenient container, but it's not your only option. You can also place pages directly inside other windows or even inside other pages. This makes for an extremely flexible system, because you can reuse the same page in different ways, depending on the type of application you need to create.

To embed a page inside a window, you simply need to use the Frame class. The Frame class is a content control that can hold any element, but it makes particular sense when used as a container for a page. It includes a property named Source, which points to a XAML page that you want to display.

Here's an ordinary window that wraps some content in a StackPanel and places a Frame in a separate column:

```
<Window x:Class="WindowPageHost.WindowWithFrame"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WindowWithFrame" Height="300" Width="300"
    >
  <Grid Margin="3">
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <StackPanel>
      <TextBlock Margin="3" TextWrapping="Wrap">
       This is ordinary window content.</TextBlock>
      <Button Margin="3" Padding="3">Close</Button>
    </StackPanel>
    <Frame Grid.Column="1" Source="Page1.xaml"
     BorderBrush="Blue" BorderThickness="1"></Frame>
  </Grid>
</Window>
```

Figure 24-4 shows the result. A border around the frame shows the page content. There's no reason you need to stop at one frame. You can easily create a window that wraps multiple frames, and you can point them to different pages.

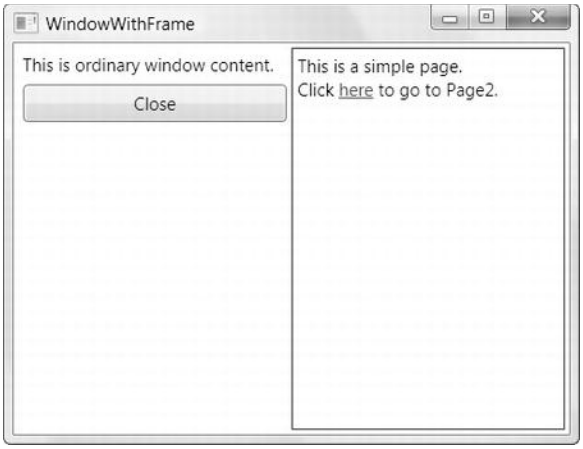

Figure 24-4. *A window with a page embedded in a frame*

As you can see in Figure 24-4, this example doesn't include the familiar navigation buttons. This is because the Frame.NavigationUIVisibility property is (by default) set to Automatic. As a result, the navigation controls appear only after there's something in the forward and back list. To try this, navigate to a new page. You'll see the buttons appear inside the frame, as shown in Figure 24-5.

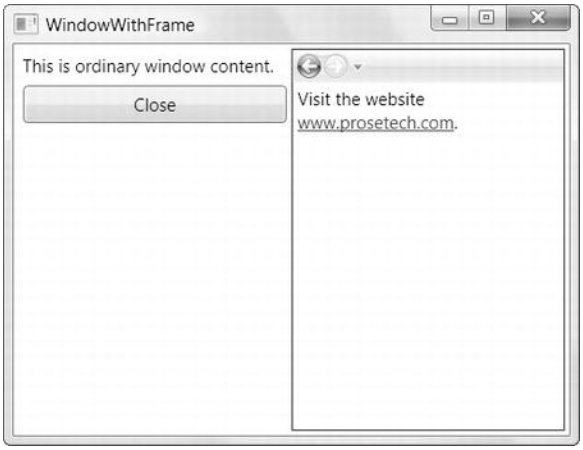

Figure 24-5. *A frame with navigation buttons*

You can change the NavigationUIVisibility property to Hidden if you never want to show the navigation buttons, or change it to Visible if you want them to appear right from the start.

Having the navigation buttons inside the frame is a great design if your frame contains content that's separate from the main flow of the application. (For example, maybe you're using it to display context-sensitive help or the content for a walk-through tutorial.) But in other cases, you may prefer to show the buttons at the top of the window. To do this, you need to change your top-level container from Window to NavigationWindow. That way, your window will include the navigation buttons. The frame inside the

window will automatically wire itself up to these buttons, so the user gets a similar experience to what's shown in Figure 24-3, except now the window also holds the extra content.

■ **Tip** You can add as many Frame objects as you need to a window. For example, you could easily create a window that allows the user to browse through an application task, help documentation, and an external website, using three separate frames.

Hosting Pages in Another Page

Frames give you the ability to create more-complex arrangements of windows. As you learned in the previous section, you can use several frames in a single window. You can also place a frame inside another page to create a *nested* page. In fact, the process is exactly the same—you simply add a Frame object inside your page markup.

Nested pages present a more complex navigation situation. For example, imagine you visit a page and then click a link in an embedded frame. What happens when you click the back button?

Essentially, all the pages in a frame are flattened into one list. So the first time you click the back button, you move to the previous page in the embedded frame. The next time you click the back button, you move to the previously visited parent page. Figure 24-6 shows the sequence you follow. Notice that the back navigation button is enabled in the second step.

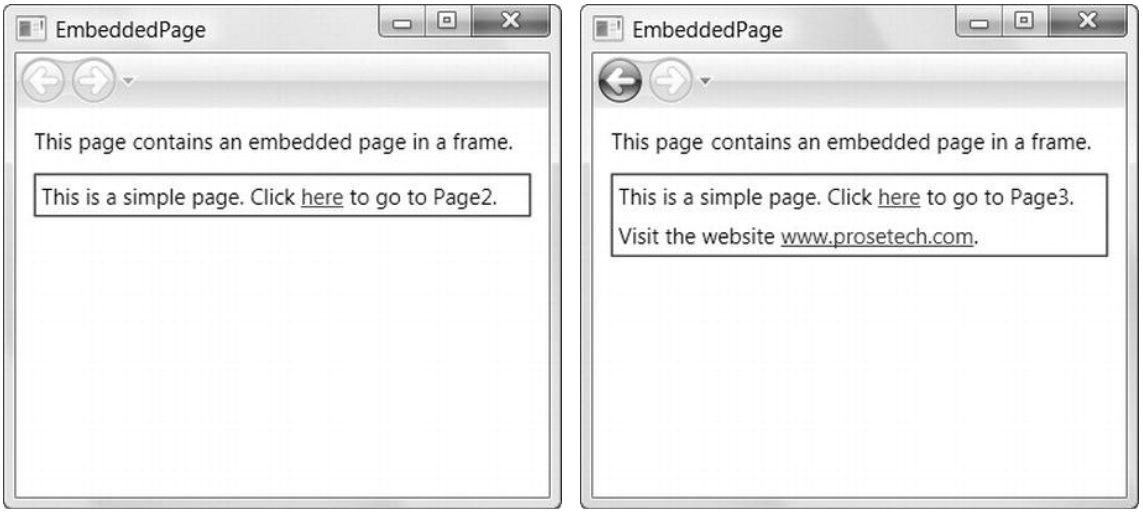

Figure 24-6. *Navigation with an embedded page*

Most of the time, this navigation model is fairly intuitive, because you'll have one item in the back list for each page you visit. However, in some cases the embedded frame plays a less important role. For example, maybe it shows different views of the same data or allows you to step through multiple pages of help content. In these cases, stepping through all the pages in the embedded frame may seem awkward or time-consuming. Instead, you may want to use the navigation controls to control the navigation of the parent frame only, so that when you click the back button, you move to the previous parent page right

away. To do this, you need to set the JournalOwnership property of the embedded frame to OwnsJournal. This tells the frame to maintain its own distinct page history. By default, the embedded frame will now acquire navigation buttons that allow you to move back and forth through its content (see Figure 24-7). If this isn't what you want, you can use the JournalOwnership property in conjunction with the NavigationUIVisibility property to hide the navigation controls altogether, as shown here:

```
<Frame Source="Page1.xaml"
  JournalOwnership="OwnsJournal" NavigationUIVisibility="Hidden"
  BorderThickness="1" BorderBrush="Blue"></Frame>
```

Now the embedded frame is treated as though it's just a piece of dynamic content inside your page. From the user's point of view, the embedded frame doesn't support navigation.

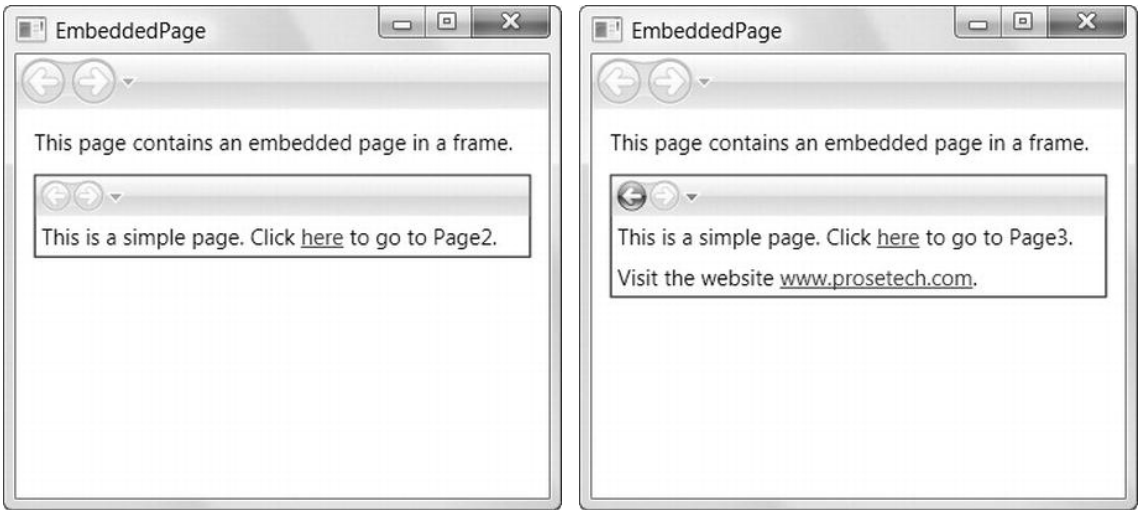

Figure 24-7. *An embedded page that owns its journal and supports navigation*

Hosting Pages in a Web Browser

The final way that you can use page-based navigation applications is in Internet Explorer or Firefox. However, in order to use this approach, you need to create a *XAML browser application* (which is known as an XBAP). In Visual Studio, the XBAP is a separate project template, and you must select it (rather than the standard WPF Windows application) when creating a project in order to use browser hosting. You'll examine the XBAP model later in this chapter.

GETTING THE RIGHT SIZE WINDOW

There are really two types of page-based applications:

- Stand-alone Windows applications that use pages for part or all of their user interfaces. You'll use this approach if you need to integrate a wizard into your application or you want a simple task-oriented application. This way, you can use WPF's navigation and journal features to simplify your coding.

- Browser applications (XBAPs) that are hosted by Internet Explorer or Firefox, and usually run with limited permissions. You'll use this approach if you want a lightweight, web-based deployment model.

If your application falls into the first category, you probably won't want to set the Application.StartupUri property to point to a page. Instead, you'll create the NavigationWindow manually, and then load your first page inside it (as shown earlier), or you'll embed your pages in a custom window by using the Frame control. Both of these approaches give you the flexibility to set the size of the application window, which is important for making sure your application looks respectable when it first starts up. On the other hand, if you're creating an XBAP, you have no control over the size of the containing web browser window, and you *must* set the StartupUri property to point to a page.

The Page History

Now that you've learned about pages and the different ways to host them, you're ready to delve deeper into the navigation model that WPF uses. In this section, you'll learn how WPF hyperlinks work and how pages are restored when you navigate back to them.

A Closer Look at URIs in WPF

You might wonder how properties such as Application.StartupUri, Frame.Source, and Hyperlink.NavigateUri actually work. In an application that's made up of loose XAML files and run in the browser, it's fairly straightforward: when you click a hyperlink, the browser treats the page reference as a relative URI and looks for the XAML page in the current folder. But in a compiled application, the pages are no longer available as separate resources; instead, they're compiled to Binary Application Markup Language (BAML) and embedded into the assembly. So, how can they be referenced using a URI?

This system works because of the way that WPF addresses application resources. When you click a hyperlink in a compiled XAML application, the URI is still treated as a relative path. However, it's relative to the *base URI* for the application. That's because a hyperlink that points to Page1.xaml is actually expanded to the pack URI shown here:

```
pack://application:,,,/Page1.xaml
```

Chapter 7 describes the pack URI syntax in detail. But the most important detail is the final portion, which includes the resource name.

At this point, you might be wondering why it's important to understand how hyperlink URIs work if the process is so seamless. The chief reason is that you might choose to create an application that navigates to XAML pages stored in another assembly. In fact, there are good reasons for this design. Because pages can be used in different containers, you might want to reuse the same set of pages in an XBAP and an ordinary Windows application. That way, you can deploy two versions of your application: a browser-based version and a desktop version. To avoid duplicating your code, you should place all the pages you plan to reuse in a separate class library assembly (DLL), which can then be referenced by both your application projects.

This necessitates a change in your URIs. If you have a page in one assembly that points to a page in another, you need to use the following syntax:

```
pack://application:,,,/PageLibrary;component/Page1.xaml
```

Here, the component is named PageLibrary and the path ,,,/PageLibrary;component/Page1.xaml points to a page named Page1.xaml that's compiled and embedded inside.

Of course, you probably won't use the absolute path. Instead, it makes more sense to use the following slightly shorter relative path in your URIs:

```
/PageLibrary;component/Page1.xaml
```

■ **Tip** Use the project template called Custom Control Library (WPF) when you create the SharedLibrary assembly to get the correct assembly references, namespace imports, and application settings.

Navigation History

The WPF page history works just like the history in a browser. Every time you navigate to a new page, the previous page is added to the back list. If you click the back button, the page is added to the forward list. If you back out from one page and then navigate to a new page, the forward list is cleared.

The behavior of the back and forward lists is fairly straightforward, but the plumbing that supports them is more complex. For example, imagine you visit a page with two text boxes, type something in, and move ahead. If you head back to this page, you'll find that WPF restores the state of your text boxes—meaning whatever content you placed in them is still there.

■ **Note** There's an important difference between returning to a page through the navigation history and clicking a link that takes you to the same page. For example, if you click links that take you from Page1 to Page2 to Page1, WPF creates three separate page objects. The second time you see Page1, WPF creates it as a separate instance, with its own state. However, if you click the back button twice to return to the first Page1 instance, you'll see that your original Page1 state remains.

You might assume that WPF maintains the state of previously visited pages by keeping the page object in memory. The problem with that approach is that the memory overhead may not be trivial in a complex application with many pages. For that reason, WPF can't assume that maintaining the page object is a safe strategy. Instead, when you navigate away from a page, WPF stores the state of all your controls and then destroys the page. When you return to a page, WPF re-creates the page (from the original XAML) and then restores the state of your controls. This strategy has lower overhead because the memory required to save just a few details of control state is far less than the memory required to store the page and its entire visual tree of objects.

This system raises an interesting question: how does WPF decide which details to store? WPF examines the complete element tree of your page, and it looks at the dependency properties of all your elements. Properties that should be stored have a tiny bit of extra metadata—a *journal* flag that indicates they should be kept in the navigation log known as the *journal*. (The journal flag is set by using the FrameworkPropertyMetadata object when registering the dependency property, as described in Chapter 4.)

If you take a closer look at the navigation system, you'll find that many properties don't have the journal flag. For example, if you set the Content property of a content control or the Text property of a TextBlock element by using code, neither of these details will be retained when you return to the page. The same is true if you set the Foreground or Background properties dynamically. However, if you set the Text property of a TextBox, the IsSelected property of a CheckBox, or the SelectedIndex property of a ListBox, all these details will remain.

So what can you do if this isn't the behavior you want? What if you set many properties dynamically, and you want your pages to retain all of their information? You have several options. The most powerful is to use the Page.KeepAlive property, which is false by default. When set to true, WPF doesn't use the serialization mechanism described previously. Instead, it keeps all your page objects alive. Thus, when you navigate back to a page, it's exactly the way you left it. Of course, this option has the drawback of increased memory overhead, so you should enable it only on the few pages that really need it.

■ **Tip** When you use the KeepAlive property to keep a page alive, it won't fire the Initialized event the next time you navigate to it. (Pages that aren't kept alive but are "rehydrated" by using WPFs journaling system will fire the Initialized event each time the user visits them.) If this behavior isn't what you want, you should instead handle the Unloaded and Loaded events of the Page, which always fire.

Another solution is to choose a different design that passes information around. For example, you can create page functions (described later in this chapter) that return information. Using page functions, along with extra initialization logic, you can design your own system for retrieving the important information from a page and restoring it when needed.

There's one more wrinkle with the WPF navigation history. As you'll discover later in this chapter, you can write code that dynamically creates a page object and then navigates to it. In this situation, the ordinary mechanism of maintaining the page state won't work. WPF doesn't have a reference to the XAML document for the page, so it doesn't know how to reconstruct the page. (And if the page is created dynamically, there may not even *be* a corresponding XAML document.) In this situation, WPF always keeps the page object alive in memory, no matter what the KeepAlive property says.

Maintaining Custom Properties

Ordinarily, any fields in your page class lose their values when the page is destroyed. If you want to add custom properties to your page class and make sure *they* retain their values, you can set the journal flag accordingly. However, you can't take this step with an ordinary property or a field. Instead, you need to create a dependency property in your page class.

You've already taken a look at dependency properties in Chapter 4. To create a dependency property, you need to follow two steps. First, you need to create the dependency property definition. Second, you need an ordinary property procedure that sets or gets the value of the dependency property.

To define the dependency property, you need to create a static field like this:

```
private static DependencyProperty MyPageDataProperty;
```

By convention, the field that defines your dependency property has the name of your ordinary property, plus the word *Property* at the end.

■ **Note** This example uses a private dependency property. That's because the only code that needs to access this property is in the page class where it's defined.

To complete your definition, you need a static constructor that registers your dependency property definition. This is the place where you set the services that you want to use with your dependency property (such as support for data binding, animation, and journaling).

```
static PageWithPersistentData()
{
    FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
    metadata.Journal = true;

    MyPageDataProperty = DependencyProperty.Register(
      "MyPageDataProperty", typeof(string),
      typeof(PageWithPersistentData), metadata, null);
}
```

Now you can create the ordinary property that wraps this dependency property. However, when you write the getter and setter, you'll use the GetValue() and SetValue() methods that are defined in the base DependencyObject class:

```
private string MyPageData
{
    set { SetValue(MyPageDataProperty, value); }
    get { return (string)GetValue(MyPageDataProperty); }
}
```

Add all these details to a single page (in this example, one named PageWithPersistentData), and the MyPageData property value will be automatically serialized when users navigate away and restored when they return.

The Navigation Service

So far, the navigation you've seen relies heavily on hyperlinks. When this approach works, it's simple and elegant. However, in some cases, you'll want to take more control of the navigation process. For example, hyperlinks work well if you're using pages to model a fixed, linear series of steps that the user traverses from start to finish (such as a wizard). However, if you want the user to complete small sequences of steps and return to a common page, or if you want to configure the sequence of steps based on other details (such as the user's previous actions), you need something more.

Programmatic Navigation

You can set the Hyperlink.NavigateUri and Frame.Source properties dynamically. However, the most flexible and powerful approach is to use the WPF navigation service. You can access the navigation service through the container that hosts the page (such as Frame or NavigationWindow), but this approach limits your pages so they can be used only in that type of container. The best approach is to access the navigation service through the static NavigationService.GetNavigationService() method. You pass a reference to your page to the GetNavigationService() method, and it returns a live NavigationService object that lets you perform programmatic navigation:

```
NavigationService nav;
nav = NavigationService.GetNavigationService(this);
```

This code works no matter which container you're using to host your pages.

■ **Note** The NavigationService isn't available in a page constructor or when the Page.Initialized event fires. Use the Page.Loaded event instead.

The NavigationService class gives you a number of methods you can use to trigger navigation. The most commonly used is the Navigate() method, which allows you to navigate to a page based on its URI:

```
nav.Navigate(new System.Uri("Page1.xaml", UriKind.RelativeOrAbsolute));
```

or by creating the appropriate page object:

```
Page1 nextPage = new Page1();
nav.Navigate(nextPage);
```

If possible, you'll want to navigate by URI, because that allows WPF's journaling system to preserve the page data without needing to keep the tree of page objects alive in memory. When you pass a page object to the Navigate() method, the entire object is always retained in memory.

However, you may decide to create the page object manually if you need to pass information into the page. You can pass in information by using a custom page class constructor (which is the most common approach), or you can call another custom method in the page class after you've created it. If you add a new constructor to the page, make sure your constructor calls InitializeComponent() to process your markup and create the control objects.

■ **Note** If you decide you need to use programmatic navigation, it's up to you whether you use button controls, hyperlinks, or something else. Typically, you'll use conditional code in your event handler to indicate the page to which you navigate.

WPF navigation is asynchronous. As a result, you can cancel the navigation request before it's complete by calling the NavigationService.StopLoading() method. You can also use the Refresh() method to reload a page.

Finally, the NavigationService also provides GoBack() and GoForward() methods, which allow you to move through the back and forward lists. This is useful if you're creating your own navigation controls. Both of these methods raise an InvalidOperationException if you try to navigate to a page that doesn't exist (for example, you attempt to go back when you're on the first page). To avoid these errors, check the Boolean CanGoBack and CanGoForward properties before using the matching methods.

Navigation Events

The NavigationService class also provides a useful set of events that you can use to react to navigation. The most common reason you'll react to navigation is to perform some sort of task when navigation is complete. For example, if your page is hosted inside a frame in a normal window, you might update status bar text in the window when navigation is complete.

Because navigation is asynchronous, the Navigate() method returns before the target page has appeared. In some cases, the time difference could be significant, such as when you're navigating to a loose XAML page on a website (or a XAML page in another assembly that triggers a web download) or when the page includes time-consuming code in its Initialized or Loaded event handler.

The WPF navigation process unfolds like this:

1. The page is located.
2. The page information is retrieved. (If the page is on a remote site, it's downloaded at this point.)
3. Any related resources that the page needs (such as images) are also located and downloaded.

4. The page is parsed, and the tree of objects is generated. At this point, the page fires its Initialized event (unless it's being restored from the journal) and its Loaded event.
5. The page is rendered.
6. If the URI includes a fragment, WPF navigates to that element.

Table 24-2 lists the events that are raised by the NavigationService class during the process. These navigation events are also provided by the Application class and by the navigation containers (NavigationWindow and Frame). If you have more than one navigation container, this gives you the flexibility to handle the navigation in different containers separately. However, there's no built-in way to handle the navigation events for a single *page*. After you attach an event handler to the navigation service in a navigation container, it continues to fire events as you move from page to page (or until you remove the event handler). Generally, this means that the easiest way to handle navigation is at the application level.

Navigation events can't be suppressed by using the RoutedEventArgs.Handled property. That's because navigation events are ordinary .NET events, not routed events.

■ **Tip** You can pass data from the Navigate() method to the navigation events. Just look for one of the Navigate() method overloads that take an extra object parameter. This object is made available in the Navigated, NavigationStopped, and LoadCompleted events through the NavigationEventArgs.ExtraData property. For example, you could use this property to keep track of the time a navigation request was made.

Table 24-2. *Events of the NavigationService Class*

Name	Description
Navigating	Navigation is just about to start. You can cancel this event to prevent the navigation from taking place.
Navigated	Navigation has started, but the target page has not yet been retrieved.
NavigationProgress	Navigation is underway, and a chunk of page data has been downloaded. This event is raised periodically to provide information about the progress of navigation. It provides the amount of information that has been downloaded (NavigationProgressEventArgs.BytesRead) and the total amount of information that's required (NavigationProgressEventArgs.MaxBytes). This event fires every time 1 KB of data is retrieved.
LoadCompleted	The page has been parsed. However, the Initialized and Loaded events have not yet been fired.
FragmentNavigation	The page is about to be scrolled to the target element. This event fires only if you use a URI with fragment information.
NavigationStopped	Navigation was canceled with the StopLoading() method.
NavigationFailed	Navigation has failed because the target page could not be located or downloaded. You can use this event to neutralize the exception before it bubbles up to become an unhandled application exception. Just set NavigationFailedEventArgs.Handled to true.

Managing the Journal

Using the techniques you've learned so far, you'll be able to build a linear navigation-based application. You can make the navigation process adaptable (for example, using conditional logic so that users are directed to different steps along the way), but you're still limited to the basic start-to-finish approach. Figure 24-8 shows this navigation topology, which is common when building simple task-based wizards. The dashed lines indicate the steps we're interested in—when the user exits a group of pages that represent a logical task.

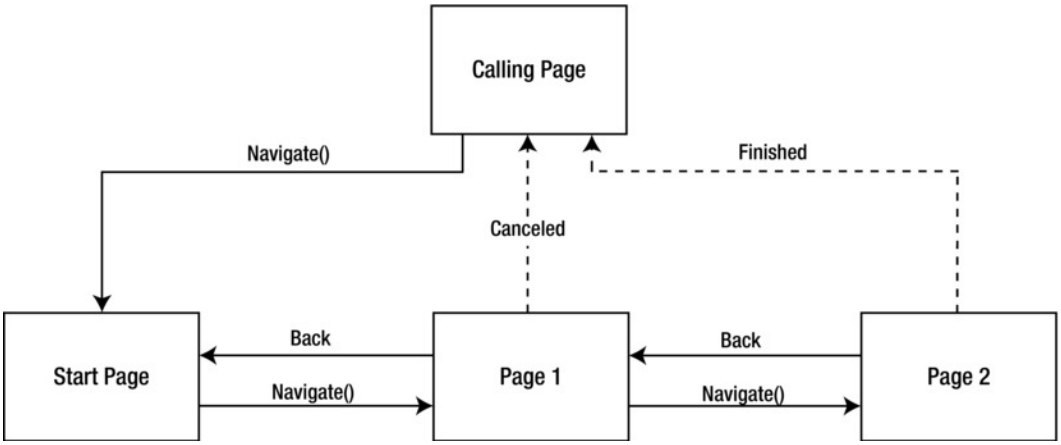

Figure 24-8. *Linear navigation*

If you try to implement this design by using WPF navigation, you'll find that there's a missing detail. Namely, when the user is finished with the navigation process (either because the user canceled the operation during one of the steps or because the user completed the task at hand), you need to wipe out the back history. If your application revolves around a main window that isn't navigation-based, this isn't a problem. When the user launches the page-based task, your application simply creates a new NavigationWindow to take the user through it. When the task ends, you can destroy that window. However, if your entire application is navigation-based, this isn't as easy. You need a way to drop the history list when the task is canceled or complete, so the user can't step back to one of the intermediary steps.

Unfortunately, WPF doesn't allow you to have much control over the navigation stack. It just gives you two methods in the NavigationService class: AddBackEntry() and RemoveBackEntry(). RemoveBackEntry() is the one you need in this example. It takes the most recent item from the back list and deletes it. RemoveBackEntry() also returns a JournalEntry object that describes that item. It tells you the URI (through the Source property) and the name that it uses in the navigation history (through the Name property). Remember that the name is set based on the Page.Title property.

If you want to clear several entries after a task is complete, you'll need to call RemoveBackEntry() multiple times. You can use two approaches. If you've decided to remove the entire back list, you can use the CanGoBack property to determine when you've reached the end:

```
while (nav.CanGoBack)
{
    nav.RemoveBackEntry();
}
```

Alternatively, you can continue removing items until you remove the task starting point. For example, if a page launches a task starting with a page named ConfigureAppWizard.xaml, you could use this code when the task is complete:

```
string pageName;
while (pageName != "ConfigureAppWizard.xaml")
{
    JournalEntry entry = nav.RemoveBackEntry();
    pageName = System.IO.Path.GetFileName(entry.Source.ToString());
}
```

This code takes the full URI that's stored in the JournalEntry.Source property and trims it down to just the page name by using the static GetFileName() method of the Path class (which works equally well with URIs). Using the Title property would make for more convenient coding, but it isn't as robust. Because the page title is displayed in the navigation history and is visible to the user, it's a piece of information you would need to translate into other languages when localizing your application. This would break code that expects a hard-coded page title. And even if you don't plan to localize your application, it's not difficult to imagine a scenario where the page title is changed to be clearer or more descriptive.

Incidentally, it is possible to examine all the items in the back and forward lists by using the BackStack and ForwardStack properties of the navigation container (such as NavigationWindow or Frame). However, it's not possible to get this information generically through the NavigationService class. In any case, these properties expose simple read-only collections of JournalEntry objects. They don't allow you to modify the lists, and they're rarely needed.

Adding Custom Items to the Journal

Along with the RemoveBackEntry() method, the NavigationService also gives you an AddBackEntry() method. The purpose of this method is to allow you to save "virtual" entries in the back list. For example, imagine you have a single page that allows the user to perform a fairly sophisticated configuration task. If you want the user to be able to step back to a previous state of that window, you can save it by using the AddBackEntry() method. Even though it's only a single page, it may have several corresponding entries in the list.

Contrary to what you might expect, when you call AddBackEntry(), you don't pass in a JournalEntry object. (In fact, the JournalEntry class has a protected constructor and so it can't be instantiated by your code.) Instead, you need to create a custom class that derives from the abstract System.Windows.Navigation.CustomContentState class and stores all the information you need. For example, consider the application shown in Figure 24-9, which allows you to move items from one list to another.

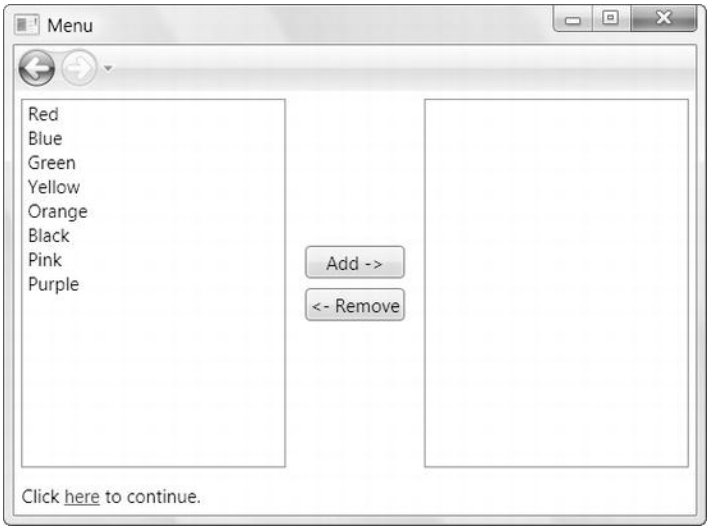

Figure 24-9. *A dynamic list*

Now imagine that you want to save the state of this window every time an item is moved from one list to the other. The first thing you need is a class that derives from CustomContentState and keeps track of this information you need. In this case, you simply need to record the contents of both lists. Because this class will be stored in the journal (so your page can be "rehydrated" when needed), it needs to be serializable:

```
[Serializable()]
public class ListSelectionJournalEntry : CustomContentState
{
    private List<String> sourceItems;
    private List<String> targetItems;
    public List<String> SourceItems
    {
        get { return sourceItems; }
    }
    public List<String> TargetItems
    {
        get { return targetItems; }
    }
    ...
```

This gets you off to a good start, but there's still a fair bit more to do. For example, you probably don't want the page to appear with the same title in the navigation history multiple times. Instead, you'll probably want to use a more descriptive name. To make this possible, you need to override the JournalEntryName property.

In this example, there's no obvious, concise way to describe the state of both lists. So it makes sense to let the page choose the name when it saves the entry in the journal. This way, the page can add a descriptive name based on the most recent action (such as Added Blue or Removed Yellow). To create this design, you simply need to make the JournalEntryName depend on a variable, which can be set in the constructor:

```
...
private string _journalName;
public override string JournalEntryName
{
    get { return _journalName; }
}
...
```

The WPF navigation system calls your JournalEntryName property to get the name it should show in the list.

The next step is to override the Replay() method. WPF calls this method when the user navigates to an entry in the back or forward list so that you can apply the previously saved state.

There are two approaches you can take in the Replay() method. You can retrieve a reference to the current page by using the NavigationService.Content property. You can then cast that into the appropriate page class and call whatever method is required to implement your change. The other approach, which is used here, is to rely on a callback:

```
...
private ReplayListChange replayListChange;

public override void Replay(NavigationService navigationService,
  NavigationMode mode)
{
    this.replayListChange(this);
}
...
```

The ReplayListChange delegate isn't shown here, but it's quite simple. It represents a method with one parameter: the ListSelectionJournalEntry object. The page can then retrieve the list information from the SourceItems and TargetItems properties and restore the page.

With this in place, the last step is to create a constructor that accepts all the information you need: the two lists of items, the title to use in the journal, and the delegate that should be triggered when the state needs to be reapplied to the page:

```
...
public ListSelectionJournalEntry(
  List<String> sourceItems, List<String> targetItems,
  string journalName, ReplayListChange replayListChange)
{
    this.sourceItems = sourceItems;
    this.targetItems = targetItems;
    this.journalName = journalName;
    this.replayListChange = replayListChange;
}
}
```

To hook up this functionality into the page, you need to take three steps:

1. Call AddBackReference() at the appropriate time to store an extra entry in the navigation history.
2. Handle the ListSelectionJournalEntry callback to restore your window when the user navigates through the history.

3. Implement the IProvideCustomContentState interface and its single GetContentState() method in your page class. When the user navigates to another page through the history, the GetContentState() method is called by the navigation service. This allows you to return an instance of your custom class that will be stored as the state of the current page.

■ **Note** The IProvideCustomContentState interface is an easily overlooked but essential detail. When the user navigates using the forward or back list, two things need to happen. Your page needs to add the current view to the journal (using IProvideCustomContentState), and then it needs to restore the selected view (using the ListSelectionJournalEntry callback).

First, whenever the Add button is clicked, you need to create a new ListSelectionJournalEntry object and call AddBackReference()journal:AddBackReference() method" so the previous state is stored in the history. This process is factored out into a separate method so that you can use it in several places in the page (for example, when either the Add button or the Remove button is clicked):

```
private void cmdAdd_Click(object sender, RoutedEventArgs e)
{
    if (lstSource.SelectedIndex != -1)
    {
        // Determine the best name to use in the navigation history.
        NavigationService nav = NavigationService.GetNavigationService(this);
        string itemText = lstSource.SelectedItem.ToString();
        string journalName = "Added " + itemText;

        // Update the journal (using the method shown below.)
        nav.AddBackEntry(GetJournalEntry(journalName));

        // Now perform the change.
        lstTarget.Items.Add(itemText);
        lstSource.Items.Remove(itemText);
    }
}

private ListSelectionJournalEntry GetJournalEntry(string journalName)
{
    // Get the state of both lists (using a helper method).
    List<String> source = GetListState(lstSource);
    List<String> target = GetListState(lstTarget);

    // Create the custom state object with this information.
    // Point the callback to the Replay method in this class.
    return new ListSelectionJournalEntry(
      source, target, journalName, Replay);
}
```

You can use a similar process when the Remove button is clicked.

The next step is to handle the callback in the Replay() method and update the lists, as shown here:

```
private void Replay(ListSelectionJournalEntry state)
{
    lstSource.Items.Clear();
    foreach (string item in state.SourceItems)
      { lstSource.Items.Add(item); }

    lstTarget.Items.Clear();
    foreach (string item in state.TargetItems)
      { lstTarget.Items.Add(item); }
}
```

And the final step is to implement IProvideCustomContentState in the page:

```
public partial class PageWithMultipleJournalEntries : Page,
 IProvideCustomContentState
```

IProvideCustomContentState defines a single method named GetContentState(). In GetContentState(), you need to store the state for the page in the same way you do when the Add or Remove button is clicked. The only difference is that you don't add it by using the AddBackReference() method. Instead, you provide it to WPF through a return value:

```
public CustomContentState GetContentState()
{
    // We haven't stored the most recent action,
    // so just use the page name for a title.
    return GetJournalEntry("PageWithMultipleJournalEntries");
}
```

Remember that the WPF navigation service calls GetContentState() when the user travels to another page by using the back or forward buttons. WPF takes the CustomContentState object you return and stores that in the journal for the current page. There's a potential quirk here. If the user performs several actions and then travels back through the navigation history reversing them, the "undone" actions in the history will have the hard-coded page name (PageWithMultipleJournalEntries), rather than the more descriptive original name (such as Added Orange). To improve the way this is handled, you can store the journal name for the page by using a member variable in your page class. The downloadable code for this example takes that extra step.

This completes the example. Now when you run the application and begin manipulating the lists, you'll see several entries appear in the history (Figure 24-10).

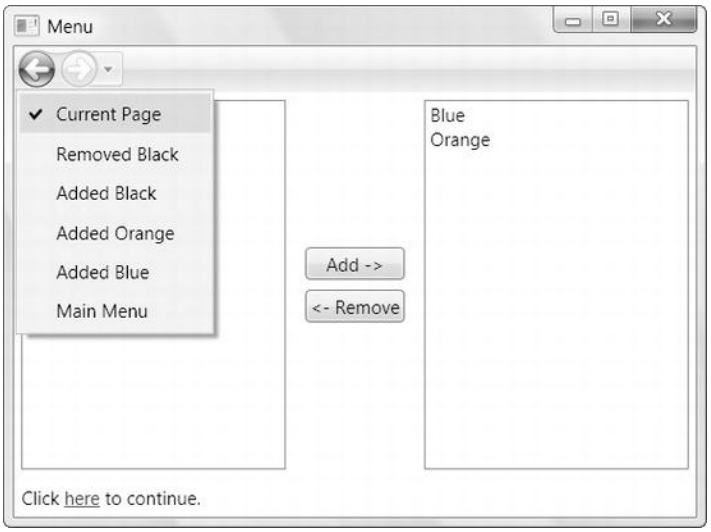

Figure 24-10. *Custom entries in the journal*

Using Page Functions

So far, you've learned how to pass information to a page (by instantiating the page programmatically, configuring it, and then passing it to the NavigationService.Navigate() method), but you haven't seen how to return information *from* a page. The easiest (and least structured) approach is to store information in some sort of static application variable so that it's accessible to any other class in your program. However, that design isn't the best if you just need a way to transmit simple bits of information from one page to another, and you don't want to keep this information in memory for a long time. If you clutter your application with global variables, you'll have a difficult time figuring out the dependencies (which variables are used by which pages), and it will become much more difficult to reuse your pages and maintain your application.

The other approach that WPF provides is the PageFunction class. A PageFunction is a derived version of the Page class that adds the ability to return a result. In a way, a PageFunction is analogous to a dialog box, while a page is analogous to a window.

To create a PageFunction in Visual Studio, right-click your project in the Solution Explorer, and choose Add ä New Item. Next, select the WPF category, choose the Page Function (WPF) template, enter a file name, and click Add. The markup for a PageFunction is nearly identical to the markup you use for a Page. The difference is the root element, which is <PageFunction> instead of <Page>.

Technically, the PageFunction is a generic class. It accepts a single type parameter, which indicates the data type that's used for the PageFunction's return value. By default, every new page function is parameterized by string (which means it returns a single string as its return value). However, you can easily modify that detail by changing the TypeArguments attribute in the <PageFunction> element.

In the following example, the PageFunction returns an instance of a custom class named Product. In order to support this design, the <PageFunction> element maps the appropriate namespace (NavigationAplication) to a suitable XML prefix (local), which is then used when setting the TypeArguments attribute.

```
<PageFunction
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:local="clr-namespace:NavigationApplication"
    x:Class="NavigationApplication.SelectProductPageFunction"
    x:TypeArguments="local:Product"
    Title="SelectProductPageFunction"
    >
```

Incidentally, as long as you set the TypeArguments attribute in your markup, you don't need to specify the same information in your class declaration. Instead, the XAML parser will generate the correct class automatically. That means this code is enough to declare the page function shown earlier:

```
public partial class SelectProductPageFunction
{ ... }
```

Although this more explicit code works just as well:

```
public partial class SelectProductPageFunction:
  PageFunction<Product>
{ ... }
```

Visual Studio uses this more explicit syntax when you create a PageFunction. By default, all new PageFunction classes that Visual Studio creates derive from PageFunction<string>.

The PageFunction needs to handle all its navigation programmatically. When you click a button or a link that finishes the task, your code must call the PageFunction.OnReturn() method. At this point, you supply the object you want to return, which must be an instance of the class you specified in the declaration. Or you can supply a null value, which indicates that the task was not completed.

Here's an example with two event handlers:

```
private void lnkOK_Click(object sender, RoutedEventArgs e)
{
    // Return the selection information.
    OnReturn(new ReturnEventArgs<Product>(lstProducts.SelectedValue));
}

private  void lnkCancel_Click(object sender, RoutedEventArgs e)
{
    // Indicate that nothing was selected.
    OnReturn(null);
}
```

Using the PageFunction is just as easy. The calling page needs to instantiate the PageFunction programmatically because it needs to hook up an event handler to the PageFunction.Returned event. (This extra step is required because the NavigationService.Navigate() method is asynchronous and returns immediately.)

```
SelectProductPageFunction pageFunction = new SelectProductPageFunction();
pageFunction.Return += new ReturnEventHandler<Product>(
  SelectProductPageFunction_Returned);
this.NavigationService.Navigate(pageFunction);
```

When the user finishes using the PageFunction and clicks a link that calls OnReturn(), the PageFunction.Returned event fires. The returned object is available through the ReturnEventArgs.Result property:

```
private void SelectProductPageFunction_Returned(object sender,
  ReturnEventArgs<Product> e)
```

```
{
    Product product = (Product)e.Result;
    if (e != null) lblStatus.Text = "You chose: " + product.Name;
}
```

Usually, the OnReturn() method marks the end of a task, and you don't want the user to be able to navigate back to the PageFunction. You could use the NavigationService.RemoveBackEntry() method to implement this, but there's an easier approach. Every PageFunction also provides a property named RemoveFromJournal. If you set this to true, the page is automatically removed from the history when it calls OnReturn().

By adding the PageFunction to your application, you now have the ability to use a different sort of navigation topology. You can designate one page as a central hub and allow users to perform various tasks through page functions, as shown in Figure 24-11.

Often, a PageFunction will call another page function. In this case, the recommended way to handle the navigation process after it's complete is to use a chained series of OnReturn() calls. In other words, if PageFunction1 calls PageFunction2, which then calls PageFunction3, when PageFunction3 calls OnReturn(), it triggers the Returned event handler in PageFunction2, which then calls OnReturn(), which then fires the Returned event in PageFunction1, which finally calls OnReturn() to end the whole process. Depending on what you're trying to accomplish, it may be necessary to pass your return object up through the whole sequence until it reaches a root page.

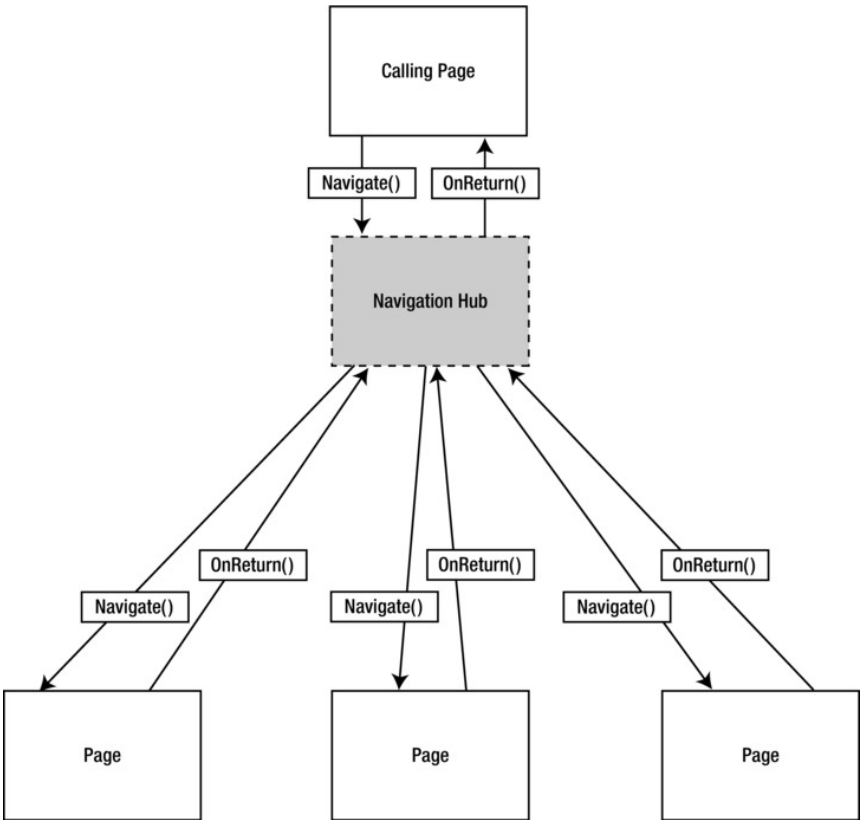

Figure 24-11. *Linear navigation*

XAML Browser Applications

XBAPs are page-based applications that run inside the browser. XBAPs are full-blown WPF applications, with a few key differences:

- *They run inside the browser window.* They can take the entire display area for the web page, or you can place them somewhere inside an ordinary HTML document by using the <iframe> tag (as you'll see shortly).

■ **Note** The technical reality is that any type of WPF application, including an XBAP, runs as a separate process managed by the Common Language Runtime (CLR). An XBAP appears to run "inside" the browser simply because it displays all its content in the browser window. This is different from the model used by plug-ins and Silverlight applications, which are loaded inside the browser process.

- *They usually have limited permissions.* Although it's possible to configure an XBAP so that it requests full trust permissions, the goal is to use XBAP as a lighter-weight deployment model that allows users to run WPF applications without allowing potentially risky code to execute. The permissions given to an XBAP are the same as the permissions given to a .NET application that's run from the Web or local intranet, and the mechanism that enforces these restrictions (code access security) is the same. That means that by default an XBAP cannot write files, interact with other computer resources (such as the Registry), connect to databases, or pop up full-fledged windows.
- *They aren't installed.* When you run an XBAP, the application is downloaded and cached in the browser. However, it doesn't remain installed on the computer. This gives you the instant-update model of the Web. In other words, every time a user returns to use an application, the newest version is downloaded if it doesn't exist in the cache.

The advantage of XBAPs is that they offer a *prompt-free* experience. If .NET is installed, a client can surf to an XBAP in the browser and start using it just like a Java applet, a Flash movie, or a JavaScript-enhanced web page. There's no installation prompt or security warning. The obvious trade-off is that you need to abide by a stringently limited security model. If your application needs greater capabilities (for example, it needs to read or write arbitrary files, interact with a database, use the Windows Registry, and so on), you're far better off creating a stand-alone Windows application. You can then offer a streamlined (but not completely seamless) deployment experience for your application by using ClickOnce deployment, which is described in Chapter 33.

Currently, two browsers are able to launch XBAP applications: Internet Explorer and Firefox. Chrome doesn't support XBAP applications (although you can google your way to some unsupported hacks that some developers have used to make them work on a specific machine). As with any .NET application, the client computer also needs the version of .NET that you targeted (when you compiled your application) in order to run it.

Creating an XBAP

Any page-based application can become an XBAP, although Visual Studio forces you to create a new project with the WPF Browser Application template in order to create one. The difference is four key elements in the .csproj project file, as shown here:

```
<HostInBrowser>True</HostInBrowser>
<Install>False</Install>
<ApplicationExtension>.xbap</ApplicationExtension>
<TargetZone>Internet</TargetZone>
```

These tags tell WPF to host the application in the browser (HostInBrowser), to cache it along with other temporary Internet files rather than install it permanently (Install), to use the extension .xbap (ApplicationExtension), and to request the permissions for only the Internet zone (TargetZone). The fourth part is optional. As you'll see shortly, it's technically possible to create an XBAP that has greater permissions. However, XBAPs almost always run with the limited permissions available in the Internet zone, which is the key challenge to programming one successfully.

■ **Tip** The .csproj file also includes other XBAP-related tags that ensure the right debugging experience. The easiest way to change an application from an XBAP into a page-based application with a stand-alone window (or vice versa) is to create a new project of the desired type, and then import all the pages from the old project.

After you've created your XBAP, you can design your pages and code them in exactly the same way as if you were using the NavigationWindow. For example, you set the StartupUri in the App.xaml file to one of your pages. When you compile your application, an .xbap file is generated. You can then request that .xbap file in Internet Explorer or Firefox, and (provided the .NET Framework is installed) the application runs in limited trust mode automatically. Figure 24-12 shows an XBAP in Internet Explorer.

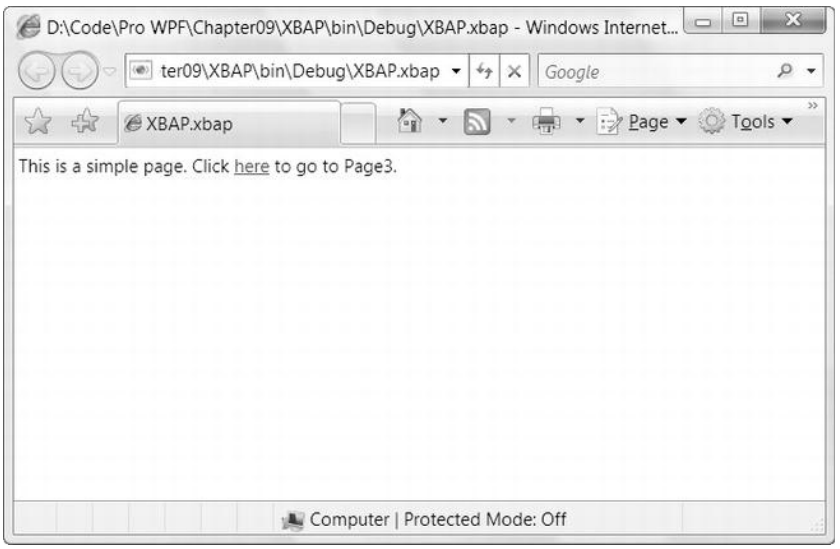

Figure 24-12. *An XBAP in the browser*

The XBAP application runs just the same as an ordinary WPF application, provided you don't attempt to perform any restricted actions (such as showing a stand-alone window). If you're running your application in Internet Explorer, the browser buttons take the place of the buttons on the NavigationWindow, and they show the back and forward page lists. On previous versions of Internet Explorer and in Firefox, you get a new set of navigation buttons at the top of your page, which isn't quite as nice.

Deploying an XBAP

Although you could create a setup program for an XBAP (and you can run an XBAP from the local hard drive), there's rarely a reason to take this step. Instead, you can simply copy your compiled application to a network share or a virtual directory.

■ **Note** You can get a similar effect by using loose XAML files. If your application consists entirely of XAML pages with no code-behind files, you don't need to compile it at all. Instead, just place the appropriate .xaml files on your web server and let users browse to them directly. Of course, loose XAML files obviously can't do as much as their compiled counterparts, but they're suitable if you simply need to display a document, a graphic, or an animation, or if you wire up all the functionality you need through declarative binding expressions.

Unfortunately, deploying an XBAP isn't as simple as just copying the .xbap file. You need to copy the following three files to the same folder:

- *ApplicationName.exe*: This file has the compiled IL code, just as it does in any .NET application.
- *ApplicationName.exe.manifest*: This file is an XML document that indicates requirements of your application (for example, the version of the .NET assemblies you used to compile your code). If your application uses other DLLs, you can make these available in the same virtual directory as your application, and they'll be downloaded automatically.
- *ApplicationName.xbap*: The .xbap file is another XML document. It represents the entry point to your application. In other words, this is the file that the user needs to request in the browser to install your XBAP. The markup in the .xbap file points to the application file and includes a digital signature that uses the key you've chosen for your project.

After you've transferred these files to the appropriate location, you can run the application by requesting the .xbap file in Internet Explorer or Firefox. It makes no difference whether the files are on the local hard drive or a remote web server—you can request them in the same way.

■ **Tip** It's tempting, but don't run the .exe file. If you do, nothing will happen. Instead, double-click the .xbap file in Windows Explorer (or type its path in the address box in your web browser). Either way, all three files must be present, and the browser must be able to recognize the .xbap file extension.

The browser will show a progress page as it begins downloading the .xbap file (Figure 24-13). This downloading process is essentially an installation process that copies the .xbap application to the local

Internet cache. When the user returns to the same remote location on subsequent visits, the cached version will be used. (The only exception occurs if there's a newer version of the XBAP on the server, as described in the next section.)

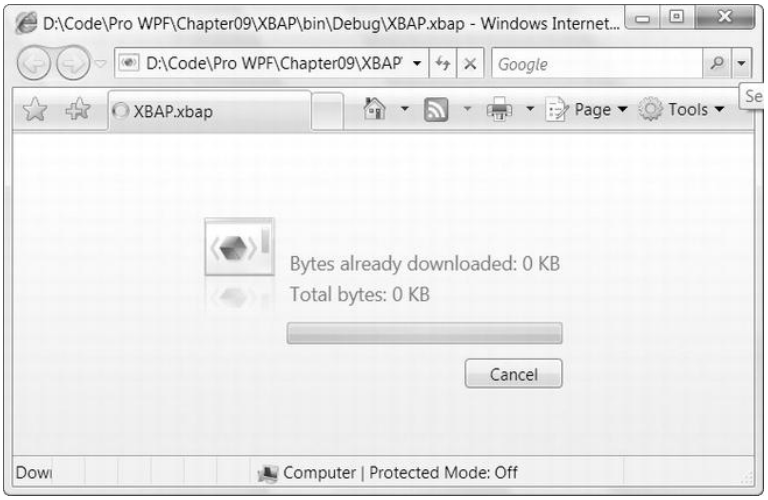

Figure 24-13. *Running an .xbap application for the first time*

When you create a new XBAP application, Visual Studio also includes an automatically generated certificate file with a name such as ApplicationName_TemporaryKey.pfx. This certificate contains a public/private key pair that's used to add a signature to your .xbap file. If you publish an update to your application, you'll need to sign it with the same key to ensure that the digital signature remains consistent.

Rather than using the temporary key, you may want to create a key of your own (which you can then share between projects and protect with a password). To do so, double-click the Properties node under your project in the Solution Explorer and use the options in the Signing tab.

Updating an XBAP

When you debug an XBAP application, Visual Studio always rebuilds your XBAP and loads the latest version in the browser. You don't need to take any extra steps. This isn't the case if you request an XBAP directly in your browser. When running XBAPs in this fashion, there's a potential problem. If you rebuild the application, deploy it to the same location, and then rerequest it in the browser, you won't necessarily get the updated version. Instead, you'll continue running the older cached copy of the application. This is true even if you close and reopen the browser window, click the browser's Refresh button, and increment the assembly version of your XBAP.

You can manually clear the ClickOnce cache, but this obviously isn't a convenient solution. Instead, you need to update the publication information that's stored in your .xbap file so that the browser recognizes that your newly deployed XBAP represents a new version of your application. Updating the assembly version isn't enough to trigger an update; instead, you need to update the *publish version*.

■ **Note** The extra step of updating the publication information is required because the download-and-cache functionality of an .xbap is built by using the plumbing from ClickOnce, the deployment technology that you'll learn

about in Chapter 33. ClickOnce uses the publication version to determine when an update should be applied. This allows you to build an application multiple times for testing (each time with a different assembly version number) but increment the publish version only when you want to deploy a new release.

The easiest way to rebuild your application *and* apply a new publication version is to choose Build ä Publish [ProjectName] from the Visual Studio menu (and then click Finish). You don't need to use the publication files (which are placed in the Publish folder under your project directory). That's because the newly generated .xbap file in the Debug or Release folder will indicate the new publish version. All you need to do is deploy this .xbap file (along with the .exe and .manifest files) to the appropriate location. The next time you request the .xbap file, the browser will download the new application files and cache them.

You can see the current publish version by double-clicking the Properties item in the Solution Explorer, choosing the Publish tab, and looking at the settings in the Publish Version section at the bottom of the tab. Make sure you keep the Automatically Increment Revision with Each Publish setting switched on so that the publish version is incremented when you publish your application, which clearly marks it as a new release.

XBAP Security

The most challenging aspect to creating an XBAP is staying within the confines of the limited security model. Ordinarily, an XBAP runs with the permissions of the Internet zone. This is true even if you run your XBAP from the local hard drive.

The .NET Framework uses *code access security* (a core feature that it has had since version 1.0) to limit what your XBAP is allowed to do. In general, the limitations are designed to correspond with what comparable Java or JavaScript code could do in an HTML page. For example, you'll be allowed to render graphics, perform animations, use controls, show documents, and play sounds. You can't access computer resources such as files, the Windows Registry, databases, and so on.

One simple way to find out whether an action is allowed is to write some test code and try it. The WPF documentation also has full details. Table 24-3 provides a quick list of significant supported and disallowed features.

Table 24-3. *Key WPF Features and the Internet Zone*

Allowed	Not Allowed
All core controls, including the RichTextBox	Windows Forms controls (through interop)
Pages, the MessageBox, and the OpenFileDialog	Stand-alone windows and other dialog boxes (such as the SaveFileDialog)
Isolated storage	Access to the file system and access to the Registry
2-D and 3-D drawing, audio and video, flow and XPS documents, and animation	Bitmap effects and pixel shaders (presumably because they rely on unmanaged code)
"Simulated" drag-and-drop (code that responds to mouse-move events)	Windows drag-and-drop
ASP.NET (.asmx) web services and Windows Communication Foundation (WCF) services	Most advanced WCF features (non-HTTP transport, server-initiated connections, and WS-* protocols) and communicating with any server other than the one where the XBAP is hosted

So what's the effect if you attempt to use a feature that's not allowed in the Internet zone? Ordinarily, your application fails as soon as it runs the problematic code with a SecurityException. Figure 24-14 shows the result of running an ordinary XBAP that attempts to perform a disallowed action and not handling the resulting SecurityException.

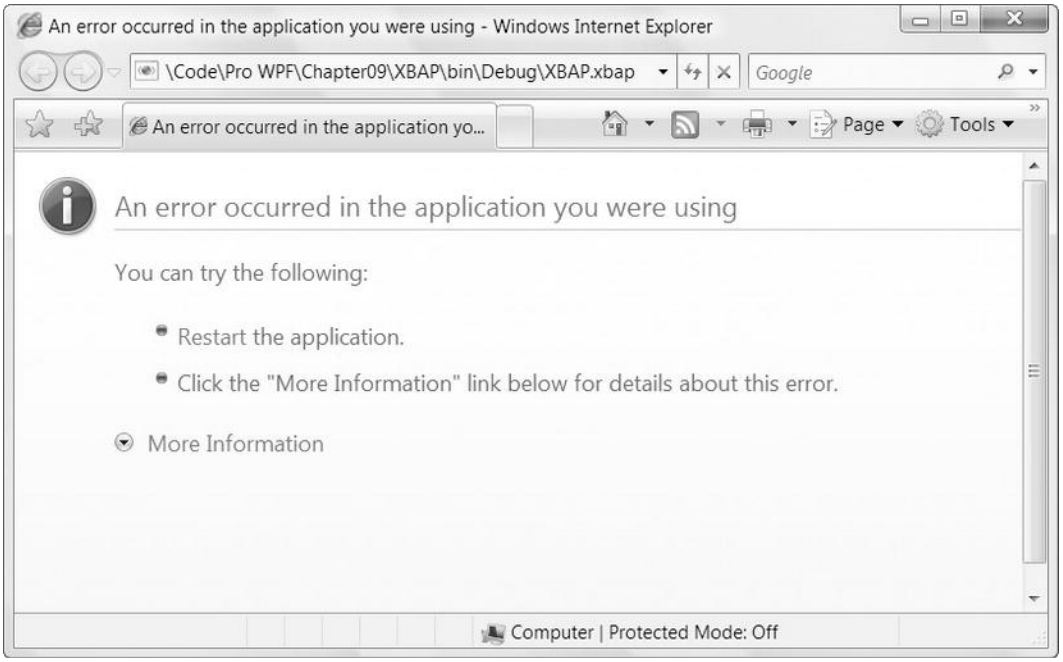

Figure 24-14. *An unhandled exception in an XBAP*

Full-Trust XBAPs

It's possible to create an XBAP that runs with full trust, although this technique isn't recommended. To do so, double-click the Properties node in the Solution Explorer, choose the Security tab, and select This Is a Full Trust Application. However, users won't be able to run your application from a web server or virtual directory anymore. Instead, you'll need to take one of the following steps to ensure that your application is allowed to execute in full trust:

- Run the application from the local hard drive. (You can launch the .xbap file like an executable file by double-clicking it or using a shortcut.) You may want to use a setup program to automate the installation process.
- Add the certificate you're using to sign the assembly (by default, it's a .pfx file) to the Trusted Publishers store on the target computer. You can do this by using the certmgr.exe tool.
- Assign full trust to the website URL or network computer where the .xbap file is deployed. To do this, you need to use the Microsoft .NET 2.0 Framework Configuration tool (which you can find in the Administrative Tools section of the Control Panel, accessed from the Start menu).

The first option is the most straightforward. However, all of these steps require an awkward configuration or deployment step that must be performed on everyone else's computer. As a result, they aren't ideal approaches.

■ **Note** If your application requires full trust, you should consider building a stand-alone WPF application and deploying it by using ClickOnce (as described in Chapter 33). The real goal of the XBAP model is to create a WPF equivalent to the traditional HTML-and-JavaScript website (or Flash applet).

Combination XBAP/Stand-Alone Applications

So far, you've considered how to deal with XBAPs that may run under different levels of trust. However, there's another possibility. You might take the same application and deploy it as both an XBAP *and* a stand-alone application that uses the NavigationWindow (as described in the beginning of this chapter).

In this situation, you don't necessarily need to test your permissions. It may be enough to write conditional logic that tests the static BrowserInteropHelper.IsBrowserHosted property and assumes that a browser-hosted application is automatically running with Internet zone permissions. The IsBrowserHosted property is true if your application is running inside the browser.

Unfortunately, changing between a stand-alone application and an XBAP is not an easy feat, because Visual Studio doesn't provide direct support. However, other developers have created tools to simplify the process. One example is the flexible Visual Studio project template found at `http://scorbs.com/2006/06/04/vs-template-flexible-application`. It allows you to create a single project file and choose between an XBAP and a stand-alone application by using the build configuration list. In addition, it provides a compilation constant you can use to conditionally compile code in either scenario, as well as an application property you can use to create binding expressions that conditionally show or hide certain elements based on the build configuration.

Another option is to place your pages in a reusable class library assembly. Then you can create two top-level projects: one that creates a NavigationWindow and loads the first page inside and another that launches the page directly as an XBAP. This makes it easier to maintain your solution, but will probably still need some conditional code that tests the IsBrowserHosted property and checks specific CodeAccessPermission objects.

Coding for Different Security Levels

In some situations, you might choose to create an application that can function in different security contexts. For example, you may create an XBAP that can run locally (with full trust) or be launched from a website. In this case, it's key to write flexible code that can avoid an unexpected SecurityException.

Every separate permission in the code-access security model is represented by a class that derives from CodeAccessPermission. You can use this class to check whether your code is running with the required permission. The trick is to call the CodeAccessPermission.Demand() method, which requests a permission. This demand fails (throwing a SecurityException) if the permission isn't granted to your application.

Here's a simple function that allows you to check for a given permission:

```
private bool CheckPermission(CodeAccessPermission requestedPermission)
{
    try
    {
        // Try to get this permission.
        requestedPermission.Demand();
        return true;
    }
    catch
    {
        return false;
    }
}
```

You can use this function to write code like this, which checks to see whether the calling code has permission to write to a file before attempting the operation:

```
// Create a permission that represents writing to a file.
FileIOPermission permission = new FileIOPermission(
  FileIOPermissionAccess.Write, @"c:\highscores.txt");

// Check for this permission.
if (CheckPermission(permission))
{
    // (It's safe to write to the file.)
}
else
{
    // (It's not allowed. Do nothing or show a message.)
}
```

The obvious disadvantage with this code is that it relies on exception handling to control normal program flow, which is discouraged (both because it leads to unclear code and because it adds overhead). An alternative would be to simply attempt to perform the operation (such as writing to a file) and then catch any resulting SecurityException. However, this approach makes it more likely that you'll run into a problem halfway through a task, when recovery or cleanup may be more difficult.

Using Isolated Storage

In many cases, you may be able to fall back on less-powerful functionality if a given permission isn't available. For example, although code running in the Internet zone isn't allowed to write to arbitrary locations on the hard drive, it is able to use isolated storage. Isolated storage provides a virtual file system that lets you write data to a small, user-specific and application-specific slot of space. The actual location on the hard drive is obfuscated (so there's no way to know exactly where the data will be written beforehand), and the total space available is typically 1 MB. A typical location is a path in the form c:\Users\[UserName]\AppData\Local\IsolatedStorage\[GuidIdentifier]. Data in one user's isolated store is restricted from all other nonadministrative users.

■ **Note** Isolated storage is the .NET equivalent of persistent cookies in an ordinary web page. It allows small bits of information to be stored in a dedicated location that has specific controls in place to prevent malicious attacks (such as code that attempts to fill the hard drive or replace a system file).

Isolated storage is covered in detail in the .NET reference. However, it's quite easy to use because it exposes the same stream-based model as ordinary file access. You simply use the types in the System.IO.IsolatedStorage namespace. Typically, you'll begin by calling the IsolatedStorageFile.GetUserStoreForApplication() method to get a reference to the isolated store for the current user and application. (Each application gets a separate store.) You can then create a virtual file in that location by using the IsolatedStorageFileStream. Here's an example:

```
// Create a permission that represents writing to a file.
string filePath = System.IO.Path.Combine(appPath, "highscores.txt");
FileIOPermission permission = new FileIOPermission(
  FileIOPermissionAccess.Write, filePath);

// Check for this permission.
if (CheckPermission(permission))
{
    // Write to local hard drive.
    try
    {
        using (FileStream fs = File.Create(filePath))
        {
            WriteHighScores(fs);
        }
    }
    catch { ... }
}
else
{
    // Write to isolated storage.
    try
    {
        IsolatedStorageFile store =
          IsolatedStorageFile.GetUserStoreForApplication();
        using (IsolatedStorageFileStream fs = new IsolatedStorageFileStream(
          "highscores.txt", FileMode.Create, store))
        {
            WriteHighScores(fs);
        }
    }
    catch { ... }
}
```

You can also use methods such as IsolatedStorageFile.GetFileNames() and IsolatedStorageFile.GetDirectoryNames() to enumerate the contents of the isolated store for the current user and application.

Remember that if you've made the decision to create an ordinary XBAP that will be deployed on the Web, you already know that you won't have FileIOPermission for the local hard drive (or anywhere else). If

this is the type of application you're designing, there's no reason to use the conditional code shown here. Instead, your code can jump straight to the isolated storage classes.

■ **Tip** To determine the amount of available isolated storage space, check IsolatedStorageFile.AvailableFreeSpace. You should use code that checks this detail and refrains from writing data if the available space is insufficient. To increase the amount of data you can pack into isolated storage, you may want to wrap your file-writing operations with the DeflateStream or GZipStream. Both types are defined in the System.IO.Compression namespace and use compression to reduce the number of bytes required to store data.

Simulating Dialog Boxes with the Pop-up Control

Another limited feature in XBAPs is the ability to open a secondary window. In many cases, you'll use navigation and multiple pages instead of separate windows, and you won't miss this functionality. However, sometimes it's convenient to pop open a window to show some sort of a message or collect input. In a stand-alone Windows application, you'd use a modal dialog box for this task. In an XBAP, there's another possibility—you can use the Popup control that was introduced in Chapter 6.

The basic technique is easy. First, you define the Popup in your markup, making sure to set its StaysOpen property to true so it will remain open until you close it. (There's no point in using the PopupAnimation or AllowsTransparency property, because neither will have any effect in a web page.) Include suitable buttons, such as OK and Cancel, and set the Placement property to Center so the pop-up will appear in the middle of the browser window.

Here's a simple example:

```
<Popup Name="dialogPopUp" StaysOpen="True" Placement="Center" MaxWidth="200">
  <Border>
    <Border.Background>
      <LinearGradientBrush>
        <GradientStop Color="AliceBlue" Offset="1"></GradientStop>
        <GradientStop Color="LightBlue" Offset="0"></GradientStop>
      </LinearGradientBrush>
    </Border.Background>
    <StackPanel Margin="5" Background="White">
      <TextBlock Margin="10" TextWrapping="Wrap">
       Please enter your name.
      </TextBlock>
      <TextBox Name="txtName" Margin="10"></TextBox>
      <StackPanel Orientation="Horizontal" Margin="10">
        <Button Click="dialog_cmdOK_Click" Padding="3" Margin="0,0,5,0">OK</Button>
        <Button Click="dialog_cmdCancel_Click" Padding="3">Cancel</Button>
      </StackPanel>
    </StackPanel>
  </Border>
</Popup>
```

At the appropriate time (for example, when a button is clicked), disable the rest of your user interface and show the Popup. To disable your user interface, you can set the IsEnabled property of some top-level container, such as a StackPanel or a Grid, to false. (You can also set the Background property of the page to

gray, which will draw the user's attention to the Popup.) To show the Popup, simply set its IsVisible property to true.

Here's an event handler that shows the previously defined Popup:

```
private void cmdStart_Click(object sender, RoutedEventArgs e)
{
    DisableMainPage();
}

private void DisableMainPage()
{
    mainPage.IsEnabled = false;
    this.Background = Brushes.LightGray;
    dialogPopUp.IsOpen = true;
}
```

When the user clicks the OK or Cancel button, close the Popup by setting its IsVisible property to false, and reenable the rest of the user interface:

```
private void dialog_cmdOK_Click(object sender, RoutedEventArgs e)
{
    // Copy name from the Popup into the main page.
    lblName.Content = "You entered: " + txtName.Text;
    EnableMainPage();
}

private void dialog_cmdCancel_Click(object sender, RoutedEventArgs e)
{
    EnableMainPage();
}

private void EnableMainPage()
{
    mainPage.IsEnabled = true;
    this.Background = null;
    dialogPopUp.IsOpen = false;
}
```

Figure 24-15 shows the Popup in action.

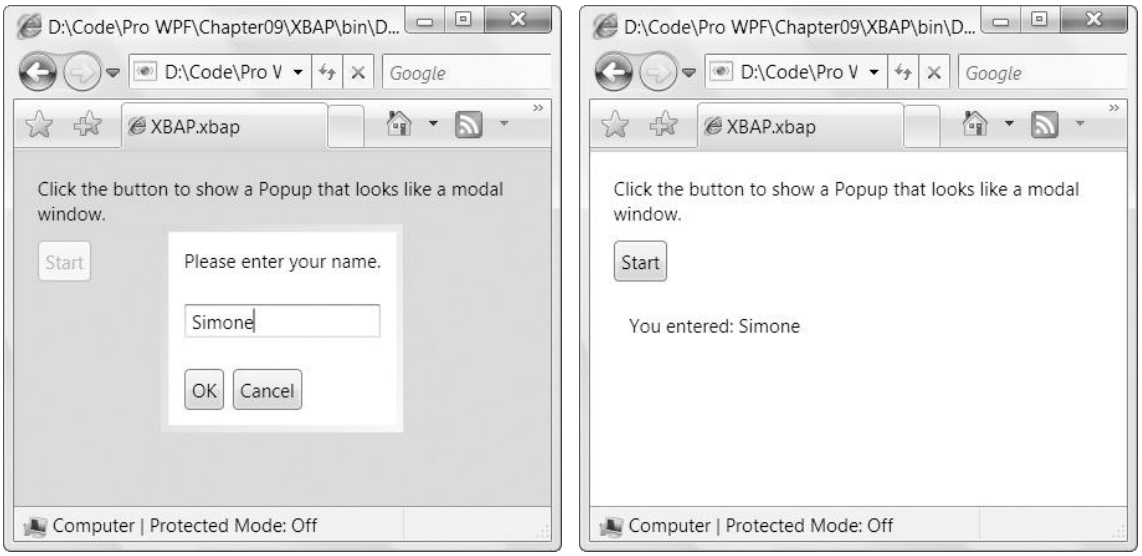

Figure 24-15. *Simulating a dialog box with the Popup*

Using the Popup control to create this work-around has one significant limitation. To ensure that the Popup control can't be used to spoof legitimate system dialog boxes, the Popup window is constrained to the size of the browser window. If you have a large Popup window and a small browser window, this could chop off some of your content. One solution, which is demonstrated with the sample code for this chapter, is to wrap the full content of the Popup control in a ScrollViewer with the VerticalScrollBarVisibility property set to Auto.

There's one other, even stranger option for showing a dialog box in a WPF page. You can use the Windows Forms library from .NET 2.0. You can safely create and show an instance of the System.Windows.Forms.Form class (or any custom form that derives from Form), because it doesn't require unmanaged code permission. In fact, you can even show the form modelessly, so the page remains responsive. The only drawback is that a security balloon automatically appears superimposed over the form and remains until the user clicks the warning message (as shown in Figure 24-16). You're also limited in what you can show *in* the form. Windows Forms controls are acceptable, but WPF content isn't allowed. For an example of this technique, refer to the sample code for this chapter.

Figure 24-16. *Using a .NET 2.0 form for a dialog box*

Embedding an XBAP in a Web Page

Usually, an XBAP is loaded directly in the browser so it takes up all the available space. However, you can have one other option: you can show an XBAP inside a portion of an HTML page, along with other HTML content. All you need to do is create an HTML page that uses the <iframe> tag to point to your .xbap file, as shown here:

```
<html>
  <head>
    <title>An HTML Page That Contains an XBAP</title>
  </head>
  <body>
    <h1>Regular HTML Content</h1>
    <iframe src="BrowserApplication.xbap"></iframe>
    <h1>More HTML Content</h1>
  </body>
</html>
```

Using an <iframe> is a relatively uncommon technique, but it does allow you to pull off a few new tricks, such as displaying more than one XBAP in the same browser window or making it easy to add your application to a site that's powered by a content-management system such as WordPress.

The WebBrowser Control

As you've seen in this chapter, WPF blurs the boundaries between traditional desktop applications and the Web. Using pages, you can create WPF applications with web-style navigation. Using XBAPs, you can run WPF inside a browser window, like a web page. And using the Frame control, you can perform the reverse trick and put an HTML web page into a WPF window.

However, when you use the Frame to show HTML content, you give up all control over that content. You have no way to inspect it or to follow along as the user navigates to a new page by clicking a link. You certainly have no way to call JavaScript methods in an HTML web page or let them call your WPF code. This is where the WebBrowser control comes into the picture.

■ **Tip** The Frame is a good choice if you need a container that can switch seamlessly between WPF and HTML content. The WebBrowser is a better choice if you need to examine the object model of a page, limit or monitor page navigation, or create a path through which JavaScript and WPF code can interact.

Both the WebBrowser and the Frame (when it's displaying HTML content) show a standard Internet Explorer window. This window has all the features and frills of Internet Explorer, including JavaScript, Dynamic HTML, ActiveX controls, and plug-ins. However, the window doesn't include additional details such as a toolbar, address bar, or status bar (although you can add all of these ingredients to your form by using other controls).

The WebBrowser isn't written from scratch in managed code. Like the Frame (when it's displaying HTML content), it wraps the shdocvw.dll COM component, which is a part of Internet Explorer and is included with Windows. As a side effect, the WebBrowser and the Frame have a few graphical limitations that other WPF controls don't share. For example, you can't place other elements on top of the HTML content that's displayed in these controls, and you can't use a transform to skew or rotate it.

■ **Note** As a feature, WPF's ability to show HTML (either through the Frame or the WebBrowser) isn't nearly as useful as the page model or XBAPs. However, you might choose to use it in specialized situations where you have already developed HTML content that you don't want to replace. For example, you might use the WebBrowser to show HTML documentation inside an application, or to allow a user to jump between the functionality in your application and that in a third-party website.

Navigating to a Page

After you've placed the WebBrowser control on a window, you need to point it to a document. The easiest approach is to set the Source property with a URI. Set this to a remote URL (for example, http://mysite.com/mypage.html) or a fully qualified file path (such as file:///c:\mydocument.text). The URI can point to any file type that Internet Explorer can open, although you'll almost always use the WebBrowser to show HTML pages.

```
<WebBrowser Source="http://www.prosetech.com"></WebBrowser>
```

■ **Note** You can also direct the WebBrowser to a directory. For example, set the Url property to file:///c:\. In this case, the WebBrowser window becomes the familiar Explorer-style file browser, allowing the user to open, copy, paste, and delete files. However, the WebBrowser doesn't provide events or properties that allow you to restrict this ability (or even monitor it), so tread carefully!

In addition to the Source property, you can navigate to a URL by using any of the navigation methods described in Table 24-4.

Table 24-4. *Navigation Methods for the WebBrowser*

Method	Description
Navigate()	Navigates to the new URL you specify. If you use the overloaded method, you can choose to load this document into a specific frame, post back data, and send additional HTML headers.
NavigateToString()	Loads the content from the string you supply, which should contain the full HTML content of a web page. This provides some interesting options, including the ability to retrieve HTML text from a resource in your application and display it.
NavigateToStream()	Loads the content from a stream that contains an HTML document. This allows you to open a file and feed it straight into the WebBrowser for rendering, without needing to hold the whole HTML content in memory at once.
GoBack() and GoForward()	Move to the previous or next document in the navigation history. To avoid errors, you should check the CanGoBack and CanGoForward properties before using these methods, because attempting to move to a document that does not exist (for example, trying to move back while on the first document in the history) will cause an exception.
Refresh()	Reloads the current document.

All WebBrowser navigation is asynchronous. That means your code continues executing while the page is downloading.

The WebBrowser also adds a small set of events, including the following:

- *Navigating* fires when you set a new URL, or the user clicks a link. You can inspect the URL, and cancel navigation by setting e.Cancel to true.
- *Navigated* fires after Navigating, just before the web browser begins downloading the page.
- *LoadCompleted* fires when the page is completely loaded. This is your chance to process the page.

Building a DOM Tree

Using the WebBrowser, you can create C# code that browses through the tree of HTML elements on a page. You can even modify, remove, or insert elements as you go, using a programming model that's similar to the HTML DOM used in web browser scripting languages such as JavaScript. In the following sections, you'll see both techniques.

Before you can use the DOM with the WebBrowser, you need to add a reference to the Microsoft HTML Object Library (mshtml.tlb). This is a COM library, so Visual Studio needs to generate a managed wrapper. To do so, choose Project ä Add Reference, pick the COM tab, select the Microsoft HTML Object Library, and click OK.

The starting point for exploring the content in a web page is the WebBrowser.Document property. This property provides an HTMLDocument object that represents a single web page as a hierarchical collection of IHTMLElement objects. You'll find a distinct IHTMLElement object for each tag in your web page, including paragraphs (<p>), hyperlinks (<a>), images (<img>), and all the other familiar ingredients of HTML markup.

The WebBrowser.Document property is read-only. That means that although you can modify the linked HtmlDocument, you can't create a new HtmlDocument object on the fly. Instead, you need to set the Source property or call the Navigate() method to load a new page. After the WebBrowser.LoadCompleted event fires, you can access the Document property.

■ **Tip** Building the HTMLDocument takes a short but distinctly noticeable amount of time (depending on the size and complexity of the web page). The WebBrowser won't actually build the HTMLDocument for the page until you try to access the Document property for the first time.

Each IHTMLElement object has a few key properties:

- *tagName* is the actual tag, without the angle brackets. For example, an anchor tag takes the form …, and has the tag name A.
- *id* contains the value of the id attribute, if specified. Often, elements are identified with unique id attributes if you need to manipulate them in an automated tool or server-side code.
- *children* provides a collection of IHTMLElement objects, one for each contained tag.
- *innerHTML* shows the full content of the tag, including any nested tags and their content.
- *innerText* shows the full content of the tag and the content of any nested tags. However, it strips out all the HTML tags.
- *outerHTML* and *outerText* play the same role as innerHTML and innerText, except they include the current tag (rather than just its contents).

To get a better understanding of innerText, innerHTML, and outerHTML, consider the following tag:

```
<p>Here is some <i>interesting</i> text.</p>
```

The innerText for this tag is as follows:

```
Here is some interesting text.
```

This is the innerHTML:

```
Here is some <i>interesting</i> text.
```

Finally, the outerHTML is the full tag:

```
<p>Here is some <i>interesting</i> text.</p>
```

In addition, you can retrieve the attribute value for an element by name, by using the IHTMLElement.getAttribute() method.

To navigate the document model for an HTML page, you simply move through the children collections of each IHTMLElement. The following code performs this task in response to a button click, and builds a tree that shows the structure of elements and the content on the page (see Figure 24-17).

```
private void cmdBuildTree_Click(object sender, System.EventArgs e)
{
    // Analyzing a page takes a nontrivial amount of time.
    // Use the hourglass cursor to warn the user.
    this.Cursor = Cursors.Wait;

    // Get the DOM object from the WebBrowser control.
    HTMLDocument dom = (HTMLDocument)webBrowser.Document;

    // Process all the HTML elements on the page, and display them
    // in the TreeView named treeDOM.
    ProcessElement(dom.documentElement, treeDOM.Items);

    this.Cursor = null;
}

private void ProcessElement(IHTMLElement parentElement,
  ItemCollection nodes)
{
    // Scan through the collection of elements.
    foreach (IHTMLElement element in parentElement.children)
    {
        // Create a new node that shows the tag name.
        TreeViewItem node = new TreeViewItem();
        node.Header = "<" + element.tagName + ">";
        nodes.Add(node);

        if ((element.children.length == 0) && (element.innerText != null))
        {
            // If this element doesn't contain any other elements, add
            // any leftover text content as a new node.
            node.Items.Add(element.innerText);
        }
        else
        {
            // If this element contains other elements, process them recursively.
            ProcessElement(element, node.Items);
        }
    }
}
```

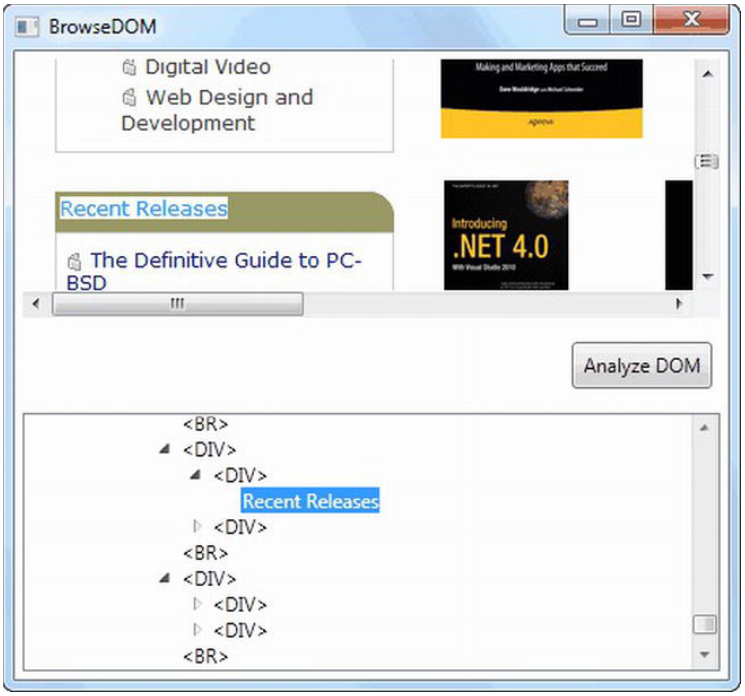

Figure 24-17. *A tree model of a web page*

If you want to find a specific element without digging through all the layers of the web page, you have a couple of simpler options. You can use the HTMLDocument.all collection, which allows you to retrieve any element on the page by using its id attribute. If you need to retrieve an element that doesn't have an id attribute, you can use the HTMLDocument method getElementsByTagName().

Scripting a Web Page with .NET Code

The last trick you'll see with the WebBrowser is something even more intriguing: the ability to react to web-page events in your Windows code.

The WebBrowser makes this technique remarkably simple. The first step is to create a class that will receive the messages from the JavaScript code. To make it scriptable, you must add the ComVisible attribute (from the System.Runtime.InteropServices namespace) to the class declaration:

```
[ComVisible(true)]
public class HtmlBridge
{
    public void WebClick(string source)
    {
        MessageBox.Show("Received: " + source);
    }
}
```

Next, you need to register an instance of this class with the WebBrowser. You do this by setting the WebBrowser.ObjectForScripting property:

```
public MainWindow()
{
    InitializeComponent();
    webBrowser.Navigate("file:///" + System.IO.Path.Combine(
      Path.GetDirectoryName(Application.ResourceAssembly.Location),
      "sample.htm"));
    webBrowser.ObjectForScripting = new HtmlBridge();
}
```

Now the sample.html web page will be able to call any public method in the HtmlBridge class, including HtmlBridge.WebClick().

In the web page, you use JavaScript code to trigger the event. Here, the trick is the window.external object, which represents the linked .NET object. Using this object, you specify a method that you want to trigger; for example, use window.external.HelloWorld() if you want to call a public method named HelloWorld in the .NET object.

■ **Caution** If you use JavaScript to trigger an event from your web page, make sure that your class doesn't include any other public methods that aren't related to web access. A nefarious user could theoretically find the HTML source, and modify it to call a different method than the one you intend. Ideally, the scriptable class should contain only web-related methods to ensure security.

To build the JavaScript command into your web page, you first need to decide to which web-page event you want to react. Most HTML elements support a small number of events, and some of the most useful include the following:

- *onFocus* occurs when a control receives focus.
- *onBlur* occurs when focus leaves a control.
- *onClick* occurs when the user clicks a control.
- *onChange* occurs when the user changes the value of certain controls.
- *onMouseOver* occurs when the user moves the mouse pointer over a control.

To write a JavaScript command that responds to one of these events, you simply add an attribute with that name to the element tag. For example, if you have an image tag that looks like this:

```
<img border="0" id="img1" src="buttonC.jpg" height="20" width="100">
```

you can add an onClick attribute that triggers the HelloWorld() method in your linked .NET class whenever the user clicks the image:

```
<img onClick="window.external.HelloWorld()" border="0" id="img1"
 src="buttonC.jpg" height="20" width="100">
```

Figure 24-18 shows an application that puts it all together. In this example, a WebBrowser control shows a local HTML file that contains four buttons, each of which is a graphical image. But when the user clicks a button, the image uses the onClick attribute to trigger the HtmlBridge.WebClick() method:

```
<img onClick="window.external.WebClick('Option1')' ... >
```

The WebClick()method then takes over. It could show another web page, open a new window, or modify part of the web page. In this example, it simply displays a message box to confirm that the event

has been received. Each image passes a hard-coded string to the WebClick() method, which identifies the button that triggered the method.

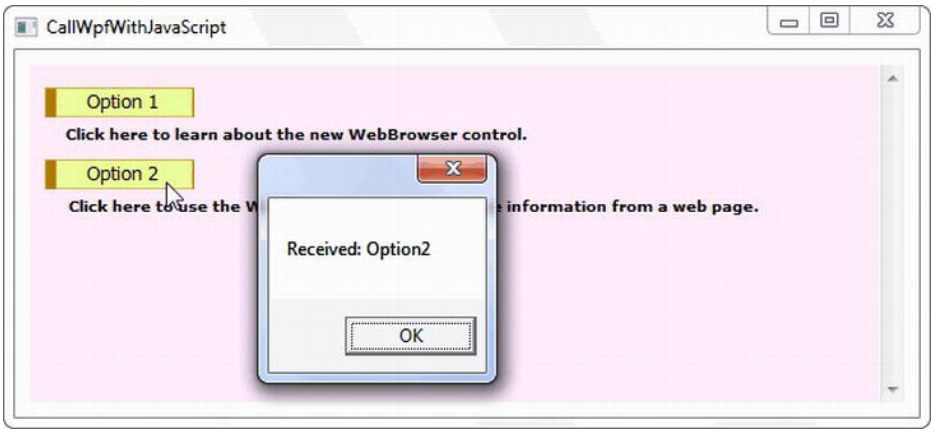

Figure 24-18. *An HTML menu that triggers .NET code*

■ **Caution** Keep in mind that unless your HTML document is compiled into your assembly as an embedded resource or retrieved from some secure location (such as a database), it may be subject to client tampering. For example, if you store HTML documents as separate files, users can easily edit them. If this is a concern, use the embedding techniques described in Chapter 7. You can create file resources, retrieve them as strings, and then show them by using the WebBrowser.NavigateToString() method.

The Last Word

In this chapter, you took a close look at the WPF navigation model. You learned how to build pages, host them in different containers, and use WPF navigation to move from one page to the next.

You also delved into the XBAP model that allows you to create a web-style WPF application that runs in a browser. Because XBAPs still require the .NET Framework, they won't replace the existing web applications that we all know and love. However, they just might provide an alternate way to deliver rich content and graphics to Windows users.

Finally, you learned how to embed web content in a WPF application by using the WebBrowser control, and how to allow your web page script code to trigger methods in your WPF application.