becomes a WebException. In order to handle this exception gracefully and prevent your application from shutting down unexpectedly, you need to neutralize it with an event handler like 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** Once 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 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.

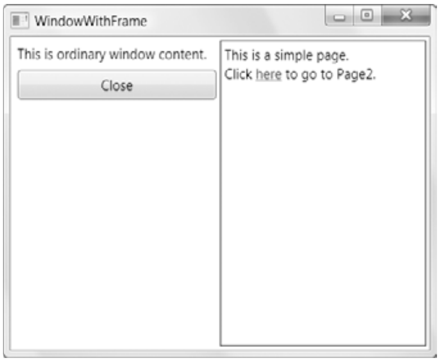
To embed a page inside a window, you simply need to use the Frame class. The Frame class is a content control that can hold any element, but it makes particular sense when used as a container for a page. It includes a property named Source, which points to a XAML page that you want to display.

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.



This is ordinary window content.

This is a simple page.
Click here to go to Page2.

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 once 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.

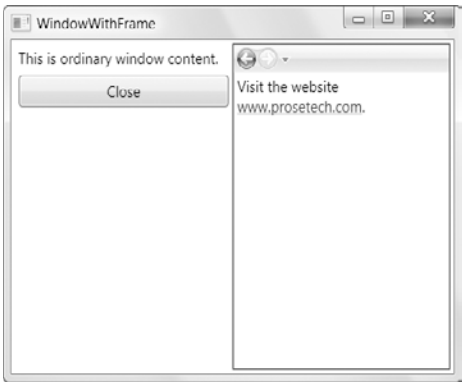


Figure 24-5. A frame with navigation buttons

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 them 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.

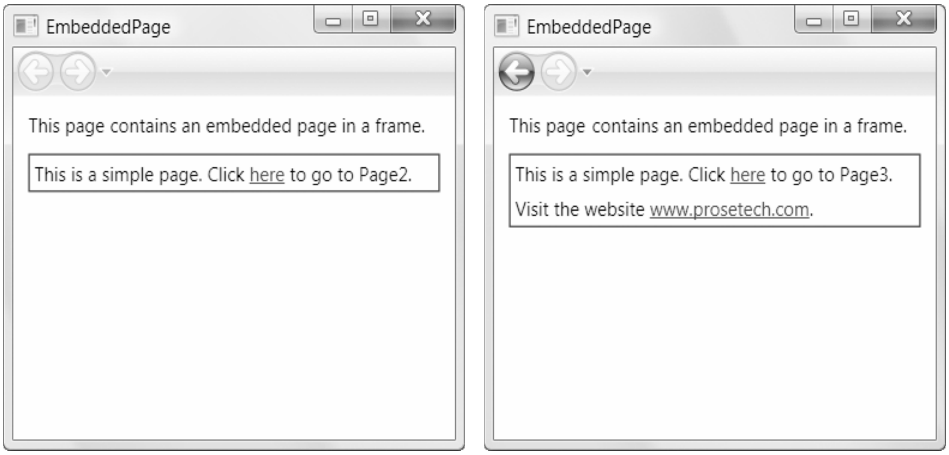
■ **Tip** You can add as many Frame objects as you need to a window. For example, you could easily create a window that allows the user to browse through an application task, help documentation, and an external website, using three separate frames.

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.



This page contains an embedded page in a frame.

This is a simple page. Click here to go to Page2.

This page contains an embedded page in a frame.

This is a simple page. Click here to go to Page3.

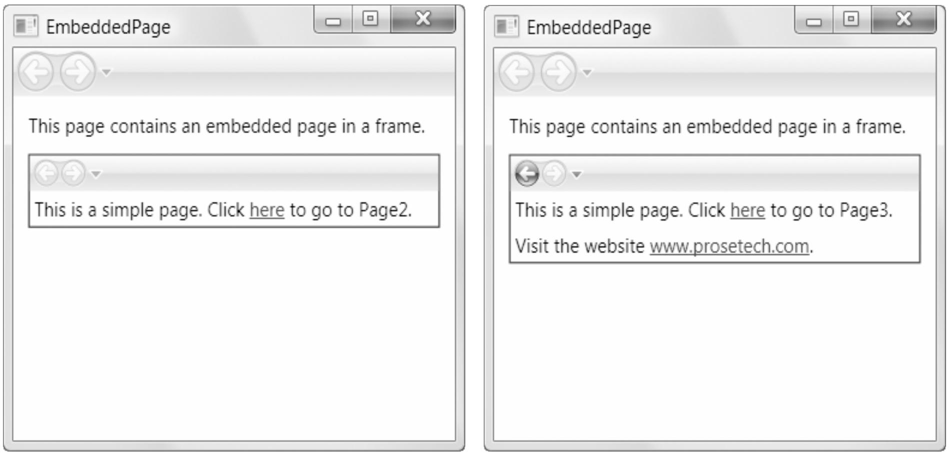
Visit the website www.prosetech.com.

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, there are some cases where 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.



EmbeddedPage

This page contains an embedded page in a frame.

This is a simple page. Click here to go to Page2.

EmbeddedPage

This page contains an embedded page in a frame.

This is a simple page. Click here to go to Page3.

Visit the website www.prosetech.com.

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 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 like 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 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 because you might choose to create an application that navigates to XAML pages that are 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 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 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" 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 information in 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 decide to which page to 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*. Once you attach an event handler to the navigation service to 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 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 1KB 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.

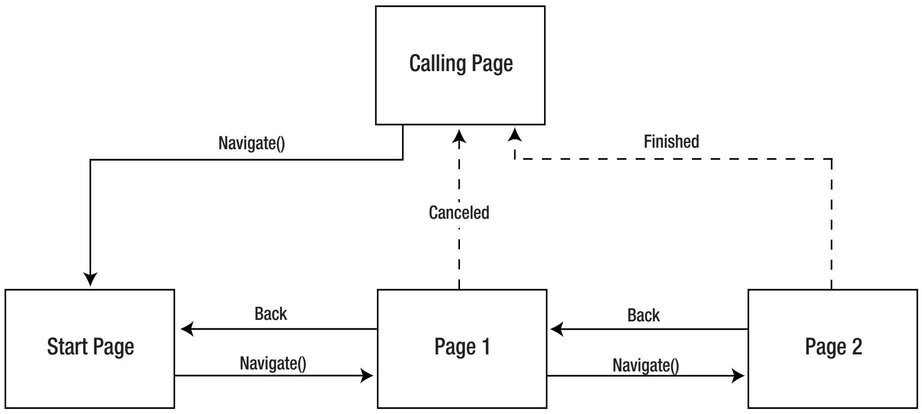


Figure 24-8. Linear navigation

If you try to implement this design 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 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 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.

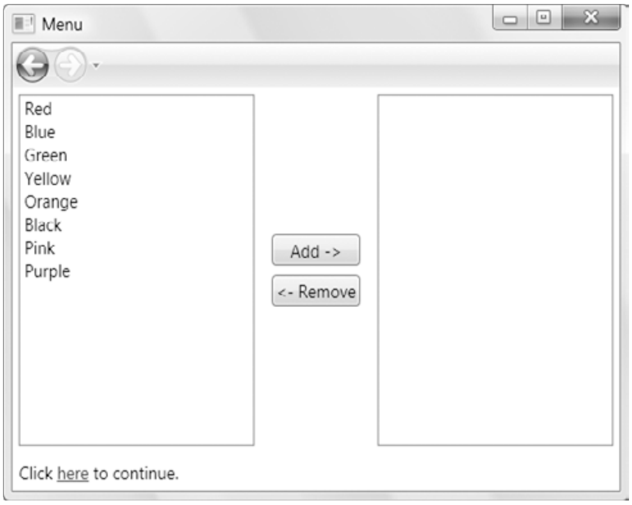


Figure 24-9. A dynamic list

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 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() 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 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 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 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).

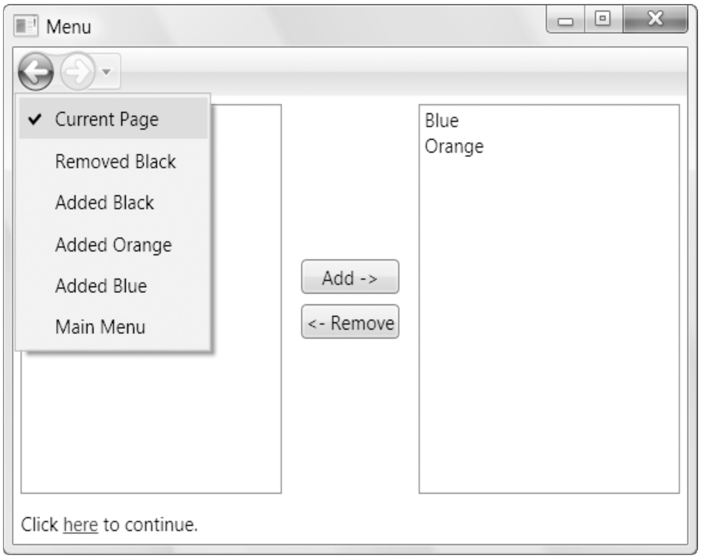


Figure 24-10. *Custom entries in the journal*

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 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 ➤ 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.