Often, a PageFunction will call another page function. In this case, the recommended way to handle the navigation process once 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.

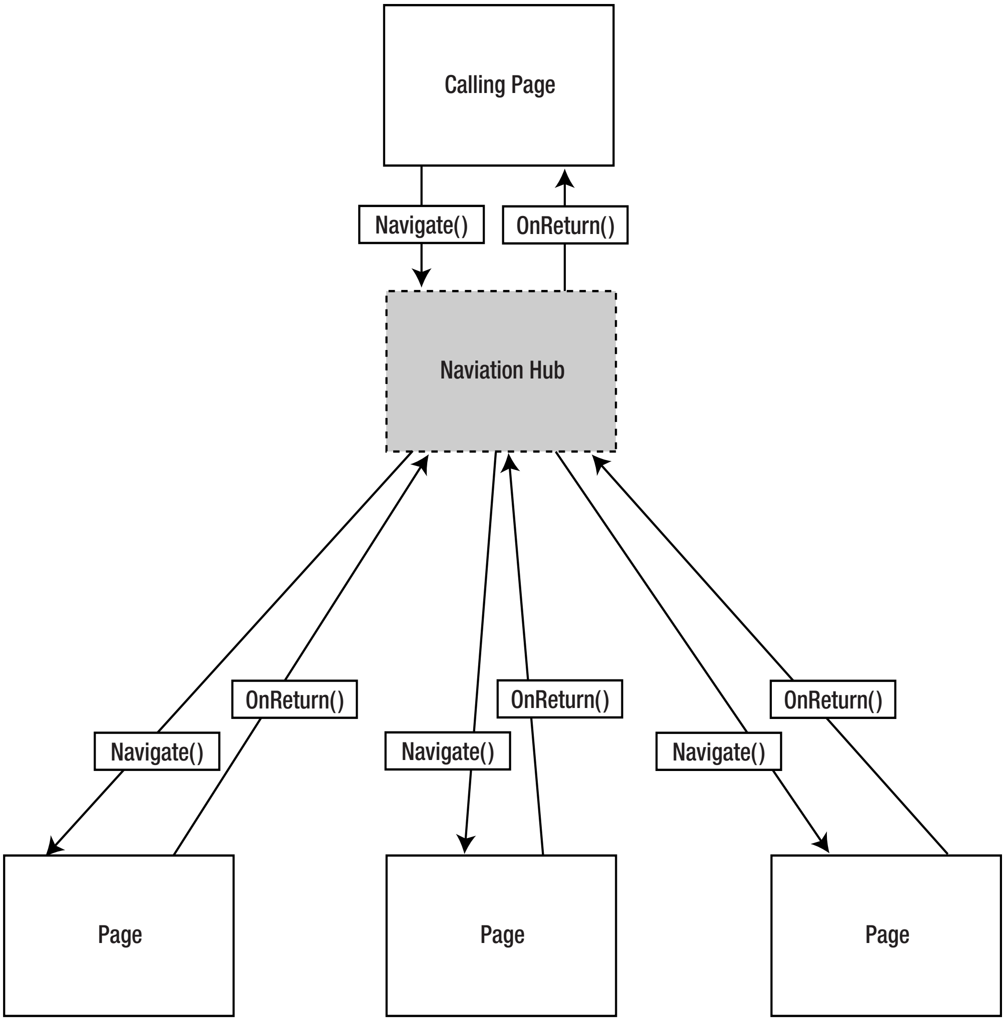


Figure 24-11. Linear navigation

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 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 ActiveX controls (and Silverlight applications), which *are* loaded inside the browser process.

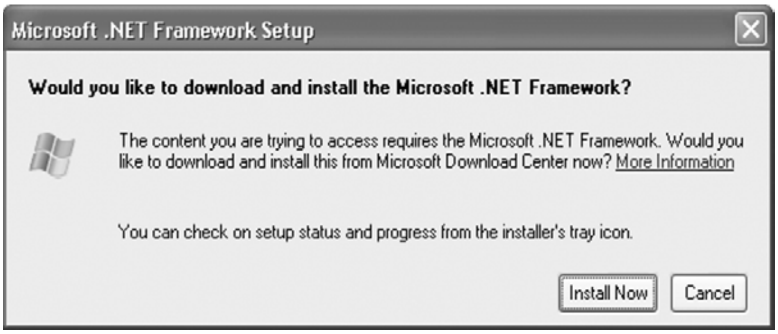
- **They usually have limited permissions.** Although it's possible to configure an XBAP so that it requests full trust permissions, the goal is to use XBAP as a lighter-weight deployment model that allows users to run WPF applications without allowing potentially risky code to execute. The permissions 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 using ClickOnce deployment, which is described in Chapter 33.

XBAP Requirements

Currently, two browsers are able to launch XBAP applications: Internet Explorer (version 6 or later) and Firefox (version 2 or later). The client computer must also have .NET Framework 3.0 or later in order to run *any* WPF application, including an XBAP. The exact version of .NET that you need to run an XBAP depends on the WPF features you're using and the version of .NET you've chosen to target, as described in Chapter 1.

These requirement aren't as limiting as they might seem. First, it's important to remember that Windows Vista includes .NET 3.0. Windows 7 includes .NET 3.5 SP1. So computers running either operating system will automatically recognize XBAPs. In addition, users running Internet Explorer 7 (or later) on Windows XP may not be able to run an XBAP, but their browsers can recognize it. When the user requests an .xbap file in this situation, Internet Explorer will give the user the option to install .NET (as shown in Figure 24-12).


Microsoft .NET Framework Setup

Would you like to download and install the Microsoft .NET Framework?

The content you are trying to access requires the Microsoft .NET Framework. Would you like to download and install this from Microsoft Download Center now? More Information

You can check on setup status and progress from the installer's tray icon.

Install Now Cancel

Figure 24-12. *Trying to launch an XBAP without .NET on Internet Explorer 7*

Creating an XBAP

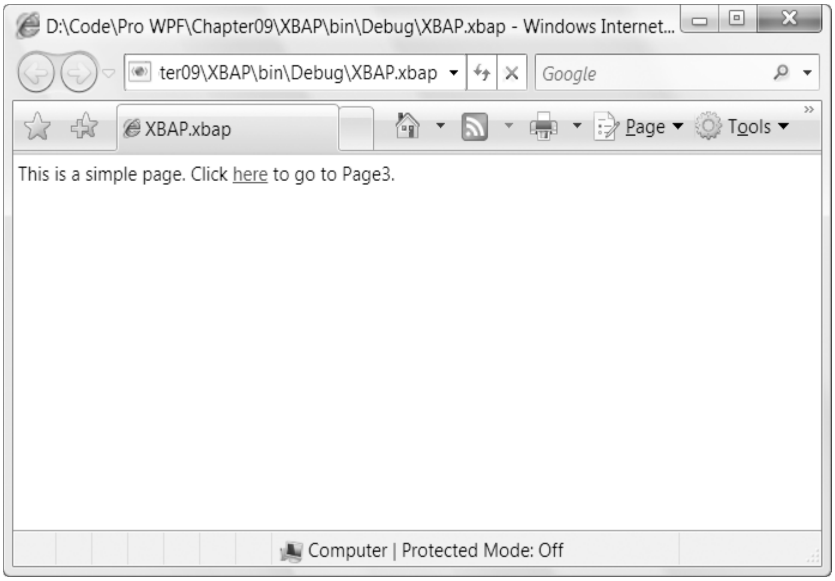
Any page-based application can become an XBAP, although Visual Studio forces you to create a new project with the WPF Browser Application template in order to create one. The difference is four key elements in the .csproj project file, as shown here:

```
<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 filealso 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.

Once 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 24-13 shows an XBAP in Internet Explorer.


D:\Code\Pro WPF\Chapter09\XBAP\bin\Debug\XBAP.xbap - Windows Internet...
ter09\XBAP\bin\Debug\XBAP.xbap
Google
XBAP.xbap
Page
Tools
Computer | Protected Mode: Off

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

Figure 24-13. *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 7 or later, 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 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 actually 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.

Once 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 24-14). 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 is if there's a newer version of the XBAP on the server, as described in the next section.)

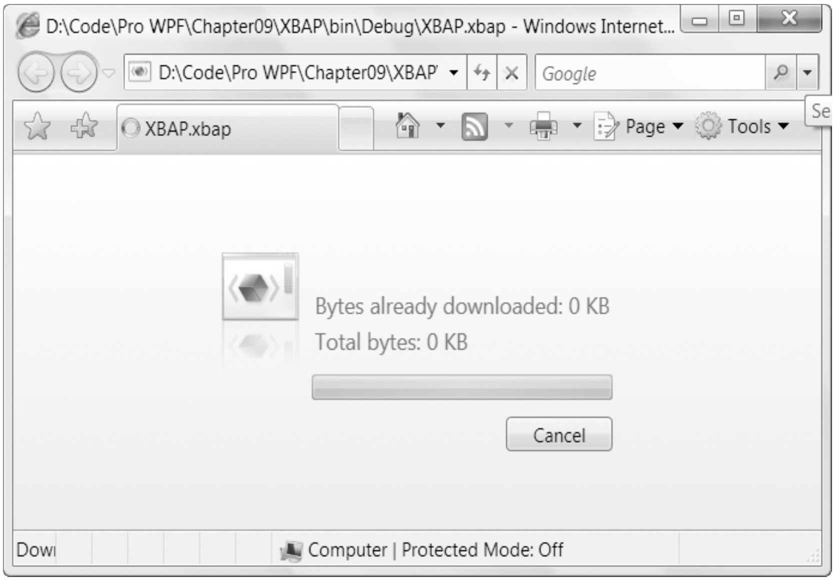

D:\Code\Pro WPF\Chapter09\XBAP\bin\Debug\XBAP.xbap - Windows Internet...
D:\Code\Pro WPF\Chapter09\XBAP
Google
XBAP.xbap
Page
Tools
Bytes already downloaded: 0 KB
Total bytes: 0 KB
Cancel
Computer | Protected Mode: Off

Figure 24-14. 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 like 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 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 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 ➤ 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 like files, the Windows registry, databases, and so on.

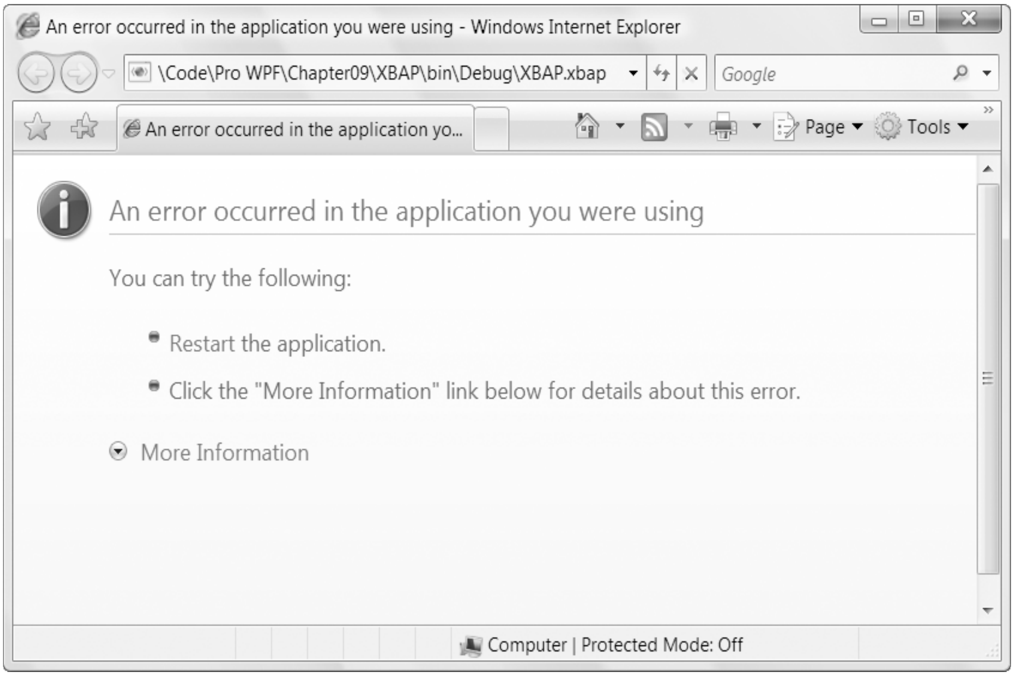
One simple way to find out whether an action is allowed is to write some test code and try it. The WPF documentation also has full details. Table 24-3 provides a quick list of significant supported and disallowed features.

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

Allowed	Not Allowed
2D and 3D 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 24-15 shows the result of running an ordinary XBAP that attempts to perform a disallowed action and not handling the resulting SecurityException.


An error occurred in the application you were using - Windows Internet Explorer

\Code\Pro WPF\Chapter09\XBAP\bin\Debug\XBAP.xbap

An error occurred in the application you were using

You can try the following:

- Restart the application.
- Click the "More Information" link below for details about this error.

More Information

Computer | Protected Mode: Off

Figure 24-15. *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 install 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 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 section in 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 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 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.

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 on a Windows 7 or Windows Vista computer 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 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 Popup 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 popup 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 24-16 shows the Popup in action.

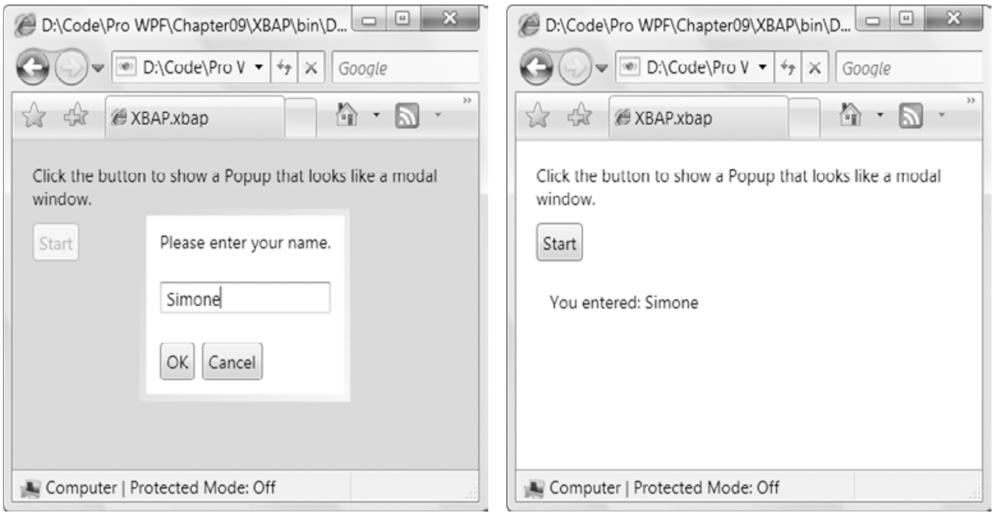
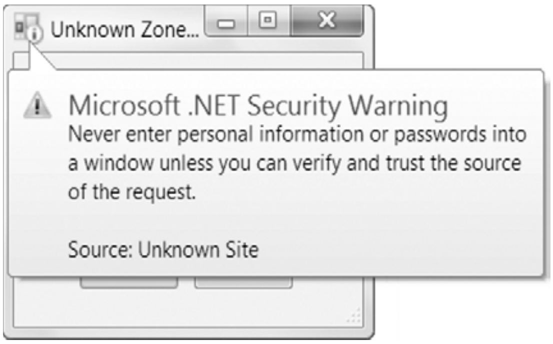

D:\Code\Pro WPF\Chapter09\XBAP\bin\D...
D:\Code\Pro V
Google
XBAP.xbap
Click the button to show a Popup that looks like a modal window.
Start
Please enter your name.
Simone
OK
Cancel
Computer | Protected Mode: Off
D:\Code\Pro WPF\Chapter09\XBAP\bin\D...
D:\Code\Pro V
Google
XBAP.xbap
Click the button to show a Popup that looks like a modal window.
Start
You entered: Simone
Computer | Protected Mode: Off

Figure 24-16. *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 24-17). 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.



Microsoft .NET Security Warning
Never enter personal information or passwords into a window unless you can verify and trust the source of the request.

Source: Unknown Site

Figure 24-17. *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. For example, it allows you to display more than one XBAP in the same browser window. It also allows you to create a WPF-driven gadget for Windows Vista or Windows 7.

■ **Note** WPF applications don't have direct support for Windows gadgets, but you can embed a WPF application in a gadget using an <iframe>. The key drawback is that the overhead of WPF application is greater than the overhead of an ordinary HTML and JavaScript web page. There are also some quirks with the way that a WPF application handles mouse input. You can find an example of this technique and a good discussion of its limitations at `http://tinyurl.com/38e5se`.

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 like a toolbar, address bar, or status bar (although you can add all of these ingredients to your form 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

Once 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 (like http://mysite.com/mypage.html) or a fully qualified file path (like 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 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, like 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 like 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 ➤ 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. Once 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 this form <a href="…">…</a>, 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, innertHTML, and outerHTML, consider the following tag:

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

The innerText for this tag is:

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

The innerHTML is:

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

Finally, the outerHTML is the full tag:

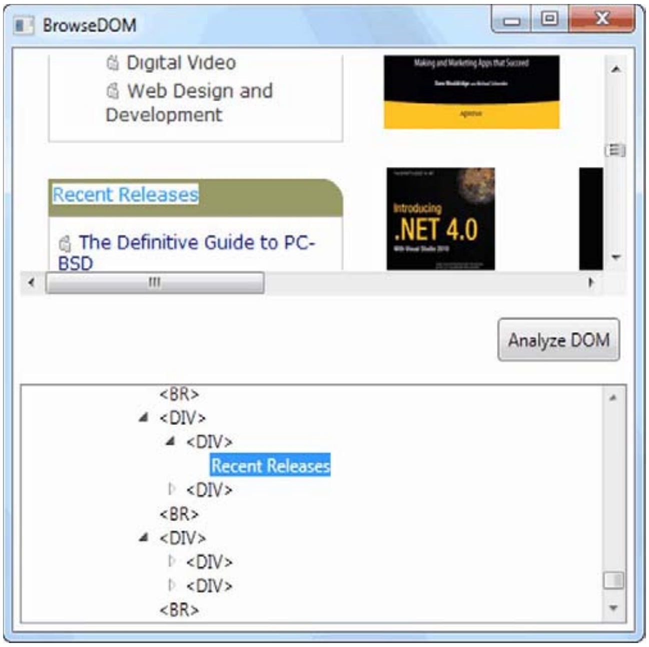
```
<p>Here is some <i>interesting</i> text.</p>
```

In addition, you can retrieve the attribute value for an element by name 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 24-18).

```
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);
            }
        }
    }
```



```
<BR>
◢ <DIV>
    ◢ <DIV>
        Recent Releases
    ▷ <DIV>
<BR>
◢ <DIV>
    ▷ <DIV>
    ▷ <DIV>
<BR>
```

Figure 24-18. 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 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.