CHAPTER 7

The Application

While it's running, every WPF application is represented by an instance of the System.Windows.Application class. This class tracks all the open windows in your application, decides when your application shuts down, and fires application events that you can handle to perform initialization and cleanup.

In this chapter, you'll explore the Application class in detail. You'll learn how you can use it to perform tasks such as catching unhandled errors, showing a splash screen, and retrieving command-line parameters. You'll even consider an ambitious example that uses instance handling and registered file types, allowing the application to manage an unlimited number of documents under one roof.

Once you understand the infrastructure that underpins the Application class, you'll consider how to create and use *assembly resources*. Every resource is a chunk of binary data that you embed in your compiled application. As you'll see, this makes resources the perfect repository for pictures, sounds, and even localized data in multiple languages.

The Application Life Cycle

In WPF, applications go through a straightforward life cycle. Shortly after your application begins, the application object is created. As your application runs, various application events fire, which you may choose to monitor. Finally, when the application object is released, your application ends.

Note WPF allows you to create full-fledged applications that give the illusion of running inside a web browser. These applications are called XAML Browser Applications (XBAPs), and you'll learn how to create them (and how to take advantage of the browser's page-based navigation system) in Chapter 24. However, it's worth noting that XBAPs use the same Application class, fire the same lifetime events, and use assembly resources in the same way as standard window-based WPF applications.

Creating an Application Object

The simplest way to use the Application class is to create it by hand. The following example shows the bare minimum: an application entry point (a Main() method) that creates a window named Window1 and fires up a new application:

```
using System;
using System.Windows;

public class Startup
{
    [STAThread()]
    static void Main()
    {
        // Create the application.
        Application app = new Application();

        // Create the main window.
        Window1 win = new Window1();

        // Launch the application and show the main window.
        app.Run(win);
    }
}
```

When you pass a window to the Application.Run() method, that window is set as the main window and exposed to your entire application through the Application.MainWindow property. The Run() method then fires the Application.Startup event and shows the main window.

You could accomplish the same effect with this more long-winded code:

```
// Create the application.
Application app = new Application();

// Create, assign, and show the main window.
Window1 win = new Window1();
app.MainWindow = win;
win.Show();

// Keep the application alive.
app.Run();
```

Both approaches give your application all the momentum it needs. When started in this way, your application continues running until the main window *and every other window* are closed. At that point, the Run() method returns, and any additional code in your Main() method is executed before the application winds down.

■ **Note** If you want to start your application by using a Main() method, you need to designate the class that contains the Main() method as the startup object in Visual Studio. To do so, double-click the Properties node in the Solution Explorer, and change the selection in the Startup Object list. Ordinarily, you don't need to take this step, because Visual Studio creates the Main() method for you based on the XAML application template. You'll learn about the application template in the next section.

Deriving a Custom Application Class

Although the approach shown in the previous section—instantiating the base Application class and calling the Run() method—works perfectly well, it's not the pattern that Visual Studio uses when you create a new WPF application.

Instead, Visual Studio derives a custom class from the Application class. In a simple application, this approach has no meaningful effect. However, if you're planning to handle application events, it provides a neater model, because you can place all your event-handling code in the Application-derived class.

The model Visual Studio uses for the Application class is essentially the same as the model it uses for the windows. The starting point is a XAML template, which is named App.xaml by default. Here's what it looks like (without the resources section, which you'll learn about in Chapter 10):

```
<Application x:Class="TestApplication.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
    >
</Application>
```

As you might remember from Chapter 2, the Class attribute is used in XAML to create a class derived from the element. Thus, this class creates a class that derives from Application, with the name TestApplication.App. (TestApplication is the name of the project, which is the same as the namespace where the class is defined, and App is the name that Visual Studio uses for the custom class that derives from Application. If you want, you can change the class name to something more exciting.)

The Application tag not only creates a custom application class, but also sets the StartupUri property to identify the XAML document that represents the main window. As a result, you don't need to explicitly instantiate this window by using code—the XAML parser will do it for you.

As with windows, the application class is defined in two separate portions that are fused together at compile time. The automatically generated portion isn't visible in your project, but it contains the Main() entry point and the code for starting the application. It looks something like this:

```
using System;
using System.Windows;

public partial class App : Application
{
    [STAThread()]
    public static void Main()
    {
        TestApplication.App app = new TestApplication.App();
        app.InitializeComponent();
        app.Run();
    }

    public void InitializeComponent()
    {
        this.StartupUri = new Uri("Window1.xaml", System.UriKind.Relative);
    }
}
```

If you're really interested in seeing the custom application class that the XAML template creates, look for the App.g.cs file in the obj\Debug folder inside your project directory.

The only difference between the automatically generated code shown here and a custom application class that you might create on your own is that the automatically generated class uses the StartupUri property instead of setting the MainWindow property or passing the main window as a parameter to the Run() method. You're free to create a custom application class that uses this approach, as long as you use the same URI format. You need to create a relative Uri object that names a XAML document that's in your project. (This XAML document is compiled and embedded in your application assembly as a BAML resource. The resource name is the name of the original XAML file. In the previous example, the application contains a resource named Window1.xaml with the compiled XAML.)

■ **Note** The URI system you see here is an all-purpose way to refer to resources in your application. You'll learn more about how it works in the "Pack URIs" section later in this chapter.

The second portion of the custom application class is stored in your project in a file such as App.xaml.cs. It contains the event-handling code you add. Initially, it's empty:

```
public partial class App : Application
{
}
```

This file is merged with the automatically generated application code through the magic of partial classes.

Application Shutdown

Ordinarily, the Application class keeps your application alive as long as at least one window is still open. If this isn't the behavior you want, you can adjust the Application.ShutdownMode. If you're instantiating your Application object by hand, you need to set the ShutdownMode property before you call Run(). If you're using the App.xaml file, you can simply set the ShutdownMode property in the XAML markup.

You have three choices for the shutdown mode, as listed in Table 7-1.

Table 7-1. *Values from the ShutdownMode Enumeration*

Value	**Description**
OnLastWindowClose	This is the default behavior—your application keeps running as long as there is at least one window in existence. If you close the main window, the Application.MainWindow property still refers to the object that represents the closed window. (Optionally, you can use code to reassign the MainWindow property to point to a different window.)
OnMainWindowClose	This is the traditional approach—your application stays alive only as long as the main window is open.
OnExplicitShutdown	The application never ends (even if all the windows are closed) unless you call Application.Shutdown(). This approach might make sense if your application is a front end for a long-running background task or if you just want to use more-complex logic to decide when your application should close (at which point you'll call the Application.Shutdown() method).

For example, if you want to use the OnMainWindowClose approach and you're using the App.xaml file, you need to make this addition:

```
<Application x:Class="TestApplication.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml" ShutdownMode="OnMainWindowClose"
    >
</Application>
```

No matter which shutdown method you choose, you can always use the Application.Shutdown() method to end your application immediately. (Of course, when you call the Shutdown() method, your application doesn't necessarily stop running right away. Calling Application.Shutdown() causes the Application.Run() method to return immediately, but there may be additional code that runs in the Main() method or responds to the Application.Exit event.)

■ **Note** When ShutdownMode is OnMainWindowClose and you close the main window, the Application object will automatically close all the other windows before the Run() method returns. The same is true if you call Application.Shutdown(). This is significant, because these windows may have event-handling code that fires when they are being closed.

Application Events

Initially, the App.xaml.cs file doesn't contain any code. Although no code is required, you can add code that handles application events. The Application class provides a small set of useful events. Table 7-2 lists the most important ones. It leaves out the events that are used solely for navigation applications (which are discussed in Chapter 24).

Table 7-2. *Application Events*

Name	Description
Startup	Occurs after the Application.Run() method is called and just before the main window is shown (if you passed the main window to the Run() method). You can use this event to check for any command-line arguments, which are provided as an array through the StartupEventArgs.Args property. You can also use this event to create and show the main window (instead of using the StartupUri property in the App.xaml file).
Exit	Occurs when the application is being shut down for any reason, just before the Run() method returns. You can't cancel the shutdown at this point, although the code in your Main() method could relaunch the application. You can use the Exit event to set the integer exit code that's returned from the Run() method.
SessionEnding	Occurs when the Windows session is ending—for example, when the user is logging off or shutting down the computer. (You can find out which one it is by examining the SessionEndingCancelEventArgs.ReasonSessionEnding property.) You can also cancel the shutdown by setting SessionEndingCancelEventArgs.Cancel to true. If you don't, WPF will call the Application.Shutdown() method when your event handler ends.

Name	Description
Activated	Occurs when one of the windows in the application is activated. This occurs when you switch from another Windows program to this application. It also occurs the first time you show a window.
Deactivated	Occurs when a window in the application is deactivated. This occurs when you switch to another Windows program.
DispatcherUnhandledException	Occurs when an unhandled exception is generated anywhere in your application (on the main application thread). The application dispatcher catches these exceptions. By responding to this event, you can log critical errors, and you can even choose to neutralize the exception and continue running your application by setting the DispatcherUnhandledExceptionEventArgs.Handled property to true. You should take this step only if you can be guaranteed that the application is still in a valid state and can continue.

You have two choices for handling events: attach an event handler or override the corresponding protected method. If you choose to handle application events, you don't need to use delegate code to wire up your event handler. Instead, you can attach it by using an attribute in the App.xaml file. For example, if you have this event handler:

```
private void App_DispatcherUnhandledException(object sender,
  DispatcherUnhandledExceptionEventArgs e)
{
    MessageBox.Show("An unhandled " + e.Exception.GetType().ToString() +
      " exception was caught and ignored.");
    e.Handled = true;
}
```

you can connect it with this XAML:

```
<Application x:Class="PreventSessionEnd.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
    DispatcherUnhandledException="App_DispatcherUnhandledException"
    >
</Application>
```

For each application event (listed in Table 7-2), a corresponding method is called to raise the event. The method name is the same as the event name, except it's prefixed with the word *On*, so Startup becomes OnStartup(), Exit becomes OnExit(), and so on. This pattern is extremely common in .NET. The only exception is the DispatcherExceptionUnhandled event—there's no OnDispatcherExceptionUnhandled() method, so you always need to use an event handler.

Here's a custom application class that overrides OnSessionEnding() and prevents both the system and itself from shutting down if a flag is set:

```
public partial class App : Application
{
    private bool unsavedData = false;
    public bool UnsavedData
    {
        get { return unsavedData; }
        set { unsavedData = value; }
    }

    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        UnsavedData = true;
    }

    protected override void OnSessionEnding(SessionEndingCancelEventArgs e)
    {
        base.OnSessionEnding(e);

        if (UnsavedData)
        {
            e.Cancel = true;
            MessageBox.Show(
              "The application attempted to be closed as a result of " +
              e.ReasonSessionEnding.ToString() +
              ". This is not allowed, as you have unsaved data.");
        }
    }
}
```

When overriding application methods, it's a good idea to begin by calling the base class implementation. Ordinarily, the base class implementation does little more than raise the corresponding application event.

Obviously, a more sophisticated implementation of this technique wouldn't use a message box—it would show some sort of confirmation dialog box that would give the user the choice of continuing (and quitting both the application and Windows) or canceling the shutdown.

Application Tasks

Now that you understand how the Application object fits into a WPF application, you're ready to take a look at how you can apply it to a few common scenarios. In the following sections, you'll consider how you can show a splash screen, process command-line arguments, support interaction between windows, add document tracking, and create a single-instance application.

■ **Note** The following sections work perfectly well for ordinary window-based WPF applications, but don't apply to browser-based WPF applications (XBAPs), which are discussed in Chapter 24. XBAPs have a built-in browser splash screen, can't receive command-line arguments, don't use multiple windows, and don't make sense as single-instance applications.

Showing a Splash Screen

As fast as they are, WPF applications don't start instantaneously. When you first fire up an application, there's a delay while the common language runtime (CLR) initializes the .NET environment and then starts your application

This delay isn't necessarily a problem. Ordinarily, this small span of time passes, and then your first window appears. But if you have more time-consuming initialization steps to take care of, or if you just want the professional polish of showing an opening graphic, you can use WPF's simple splash-screen feature.

Here's how to add a splash screen:

1. Add an image file to your project. (Typically, this is a .bmp, .png, or .jpg.)
2. Select the file in the Solution Explorer.
3. Change the Build Action to SplashScreen.

The next time you run your application, this graphic will be shown immediately, in the center of the screen. Once the runtime environment is ready, and after the Application_Startup method has finished, your application's first window appears, and the splash-screen graphic fades away quickly (in about 300 milliseconds).

This feature sounds straightforward, and it is. Just remember that the splash screen is shown without any adornments. No window border is drawn around it, so it's up to you to make sure that's a part of your splash-screen graphic. There's also no way to get fancy with a splash-screen graphic by showing a sequence of multiple images or an animation. If you want that, you need to take the traditional approach: create a startup window that runs your initialization code while showing the graphical display you want.

Incidentally, when you add a splash screen, the WPF compiler adds code like this to the automatically generated App.g.cs file:

```
SplashScreen splashScreen = new SplashScreen("splashScreenImage.png");

// Show the splash screen.
// The true parameter sets the splashScreen to fade away automatically
// after the first window appears.
splashScreen.Show(true);

// Start the application.
MyApplication.App app = new MyApplication.App();
app.InitializeComponent();
app.Run();
// The splash screen begins its automatic fade-out now.
```

You could write this sort of logic yourself instead of using the SplashScreen build action. But there's little point, as the only detail you can change is the speed with which the splash screen fades. To do that, you pass false to the SpashScreen.Show() method (so WPF won't fade it automatically). It's then up to you to hide the splash screen at the appropriate time by calling SplashScreen.Close() and supplying a TimeSpan that indicates how long the fadeout should take.

Handling Command-Line Arguments

To process command-line arguments, you react to the Application.Startup event. The arguments are provided as an array of strings through the StartupEventArgs.Args property.

For example, imagine you want to load a document when its name is passed as a command-line argument. In this case, it makes sense to read the command-line arguments and perform the extra initialization you need. The following example implements this pattern by responding to the Application.Startup event. It doesn't set the Application.StartupUri property at any point; instead, the main window is instantiated by using code.

```
public partial class App : Application
{
    private static void App_Startup(object sender, StartupEventArgs e)
    {
        // Create, but don't show the main window.
        FileViewer win = new FileViewer();

        if (e.Args.Length > 0)
        {
            string file = e.Args[0];
            if (System.IO.File.Exists(file))
            {
                // Configure the main window.
                win.LoadFile(file);
            }
        }
        else
        {
            // (Perform alternate initialization here when
            //  no command-line arguments are supplied.)
        }

        // This window will automatically be set as the Application.MainWindow.
        win.Show();
    }
}
```

This method initializes the main window, which is then shown when the App_Startup() method ends. This code assumes that the FileViewer class has a public method (that you've added) named LoadFile(). Here's one possible example, which simply reads (and displays) the text in the file you've identified:

```
public partial class FileViewer : Window
{
    ...

    public void LoadFile(string path)
    {
        this.Content = File.ReadAllText(path);
        this.Title = path;
    }
}
```

You can try an example of this technique with the sample code for this chapter.

■ **Note** At first glance, the code in the LoadFile() method looks a little strange. It sets the Content property of the current Window, which determines what the window displays in its client area. Interestingly enough, WPF windows are actually a type of content control (meaning they derive from the ContentControl class). As a result, they can contain (and display) a single object. It's up to you whether that object is a string, a control, or (more usefully) a panel that can host multiple controls.

Accessing the Current Application

You can get the current application instance from anywhere in your application by using the static Application.Current property. This allows rudimentary interaction between windows, because any window can get access to the current Application object, and through that, obtain a reference to the main window.

```
Window main = Application.Current.MainWindow;
MessageBox.Show("The main window is " + main.Title);
```

Of course, if you want to access any methods, properties, or events that you've added to your custom main window class, you need to cast the window object to the right type. If the main window is an instance of a custom MainWindow class, you can use code like this:

```
MainWindow main = (MainWindow)Application.Current.MainWindow;
main.DoSomething();
```

A window can also examine the contents of the Application.Windows collection, which provides references to *all* the currently open windows:

```
foreach (Window window in Application.Current.Windows)
{
    MessageBox.Show(window.Title + " is open.");
}
```

In practice, most applications prefer to use a more structured form of interaction between windows. If you have several long-running windows that are open at the same time and they need to communicate in some way, it makes more sense to hold references to these windows in a custom application class. That way, you can always find the exact window you need. Similarly, if you have a document-based application, you might choose to create a collection that tracks document windows but nothing else. The next section considers this technique.

■ **Note** Windows (including the main window) are added to the Windows collection as they're shown, and they're removed when they're closed. For this reason, the position of windows in the collection may change, and you can't assume you'll find a specific window object at a specific position.

Interacting Between Windows

As you've seen, the custom application class is a great place to put code that reacts to different application events. There's one other purpose that an Application class can fill quite nicely: storing references to important windows so one window can access another.

■ **Tip** This technique makes sense when you have a modeless window that lives for a long period of time and might be accessed in various classes (not just the class that created it). If you're simply showing a modal dialog box as part of your application, this technique is overkill. In this situation, the window won't exist for very long, and the code that creates the window is the only code that needs to access it. (To brush up on the difference between modal windows, which interrupt application flow until they're closed, and modeless windows, which don't, refer to Chapter 23.)

For example, imagine you want to keep track of all the document windows that your application uses. To that end, you might create a dedicated collection in your custom application class. Here's an example that uses a generic List collection to hold a group of custom window objects. In this example, each document window is represented by an instance of a class named Document:

```
public partial class App : Application
{
    private List<Document> documents = new List<Document>();

    public List<Document> Documents
    {
        get { return documents; }
        set { documents = value; }
    }
}
```

Now, when you create a new document, you simply need to remember to add it to the Documents collection. Here's an event handler that responds to a button click and does the deed:

```
private void cmdCreate_Click(object sender, RoutedEventArgs e)
{
    Document doc = new Document();
    doc.Owner = this;
    doc.Show();
    ((App)Application.Current).Documents.Add(doc);
}
```

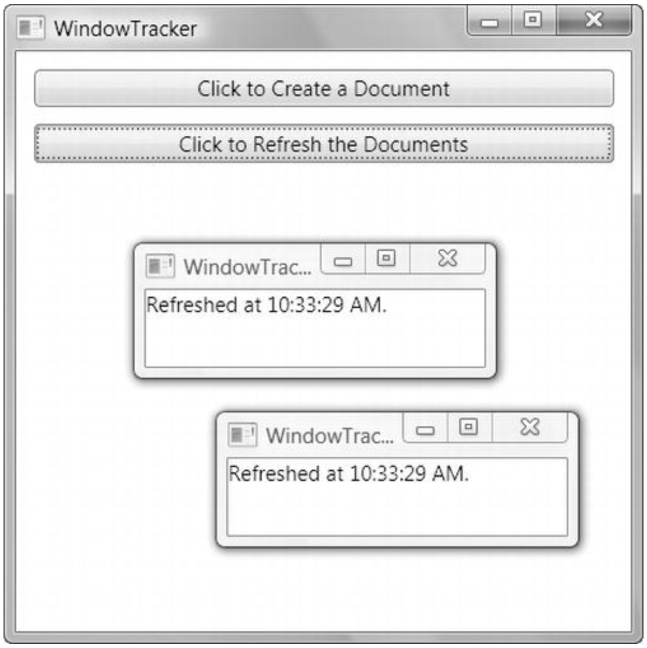
Alternatively, you could respond to an event such as Window.Loaded in the Document class to make sure the document object always registers itself in the Documents collection when it's created.

■ **Note** This code also sets the Window.Owner property so that all the document windows are displayed on top of the main window that creates them. You'll learn more about the Owner property when you consider windows in detail in Chapter 23.

Now you can use that collection elsewhere in your code to loop over all the documents and use public members. In this case, the Document class includes a custom SetContent() method that updates its display:

```
private void cmdUpdate_Click(object sender, RoutedEventArgs e)
{
    foreach (Document doc in ((App)Application.Current).Documents)
    {
        doc.SetContent("Refreshed at " + DateTime.Now.ToLongTimeString() + ".");
    }
}
```

Figure 7-1 demonstrates this application. The actual end result isn't terribly impressive, but the interaction is worth noting. This is a safe, disciplined way for your windows to interact through a custom application class. It's superior to using the Windows property, because it's strongly typed, and it holds only Document windows (not a collection of all the windows in your application). It also gives you the ability to categorize the windows in another, more useful way—for example, in a Dictionary collection with a key name for easy lookup. In a document-based application, you might choose to index windows in a collection by file name.



Refreshed at 10:33:29 AM.

Refreshed at 10:33:29 AM.

Figure 7-1. *Allowing windows to interact*

■ **Note** When interacting between windows, don't forget your object-oriented smarts. Always use a layer of custom methods, properties, and events that you've added to the window classes. Never expose the fields or controls of a form to other parts of your code. If you do, you'll quickly wind up with a tightly coupled interface where one window reaches deep into the inner workings of another, and you won't be able to enhance either class without breaking the murky interdependencies between them.

Single-Instance Applications

Ordinarily, you can launch as many copies of a WPF application as you want. In some scenarios, this design makes perfect sense. However, in other cases it's a problem, particularly when building document-based applications.

For example, consider Microsoft Word. No matter how many documents you open (or how you open them), only a single instance of winword.exe is loaded at a time. As you open new documents, they appear in the new windows, but a single application remains in control of all the document windows. This design is the best approach if you want to reduce the overhead of your application, centralize certain features (for example, create a single print queue manager), or integrate disparate windows (for example, offer a feature that tiles all the currently open document windows next to each other).

WPF doesn't provide a native solution for single-instance applications, but you can use several workarounds. The basic technique is to check whether another instance of your application is already running when the Application.Startup event fires. The simplest way to do this is to use a systemwide *mutex* (a synchronization object provided by the operating system that allows for interprocess communication). This approach is simple but limited. Most significantly, there's no way for the new instance of an application to communicate with the existing instance. This is a problem in a document-based application, because the new instance may need to tell the existing instance to open a specific document if it's passed on the command line. (For example, when you double-click a .doc file in Windows Explorer and Word is already running, you expect Word to load the requested file.) This communication is more complex, and it's usually performed through remoting or Windows Communication Foundation (WCF). A proper implementation needs to include a way to discover the remoting server and use it to transfer command-line arguments.

But the simplest approach, and the one that's currently recommended by the WPF team, is to use the built-in support that's provided in Windows Forms and originally intended for Visual Basic applications. This approach handles the messy plumbing behind the scenes.

So, how can you use a feature that's designed for Windows Forms and Visual Basic to manage a WPF application in C#? Essentially, the old-style application class acts as a wrapper for your WPF application class. When your application is launched, you'll create the old-style application class, which will then create the WPF application class. The old-style application class handles the instance management, while the WPF application class handles the real application. Figure 7-2 shows how these parts interact.

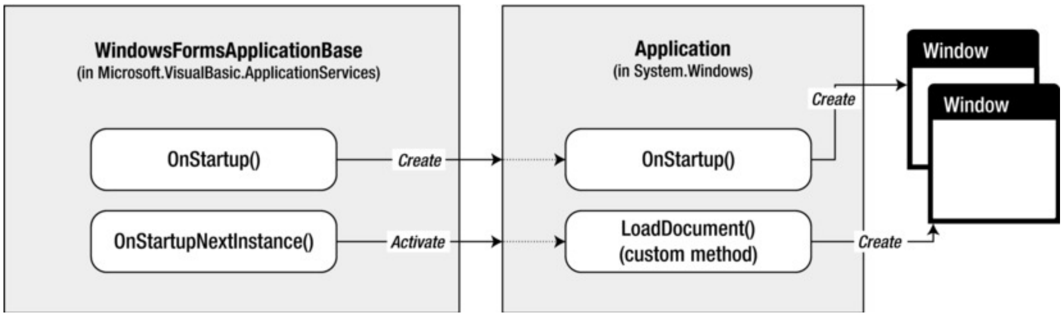


Figure 7-2. *Wrapping the WPF application with a WindowsFormsApplicationBase*

Creating the Single-Instance Application Wrapper

The first step to use this approach is to add a reference to the Microsoft.VisualBasic.dll assembly and derive a custom class from the Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase class. This class provides three important members that you use for instance management:

- The IsSingleInstance property enables a single-instance application. You set this property to true in the constructor.
- The OnStartup() method is triggered when the application starts. You override this method and create the WPF application object at this point.
- The OnStartupNextInstance() method is triggered when another instance of the application starts up. This method provides access to the command-line arguments. At this point, you'll probably call a method in your WPF application class to show a new window but not create another application object.

Here's the code for the custom class that's derived from WindowsFormsApplicationBase:

```
public class SingleInstanceApplicationWrapper :
  Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase
{
    public SingleInstanceApplicationWrapper()
    {
        // Enable single-instance mode.
        this.IsSingleInstance = true;
    }

    // Create the WPF application class.
    private WpfApp app;
    protected override bool OnStartup(
      Microsoft.VisualBasic.ApplicationServices.StartupEventArgs e)
    {
        app = new WpfApp();
        app.Run();

        return false;
    }

    // Direct multiple instances.
    protected override void OnStartupNextInstance(
      Microsoft.VisualBasic.ApplicationServices.StartupNextInstanceEventArgs e)
    {
        if (e.CommandLine.Count > 0)
        {
            app.ShowDocument(e.CommandLine[0]);
        }
    }
}
```

When the application starts, this class creates an instance of WpfApp, which is a custom WPF application class (a class that derives from System.Windows.Application). The WpfApp class includes some startup logic that shows a main window, along with a custom ShowDocument() window that loads a document window for a given file. Every time a file name is passed to SingleInstanceApplicationWrapper through the command line, SingleInstanceApplicationWrapper calls WpfApp.ShowDocument().

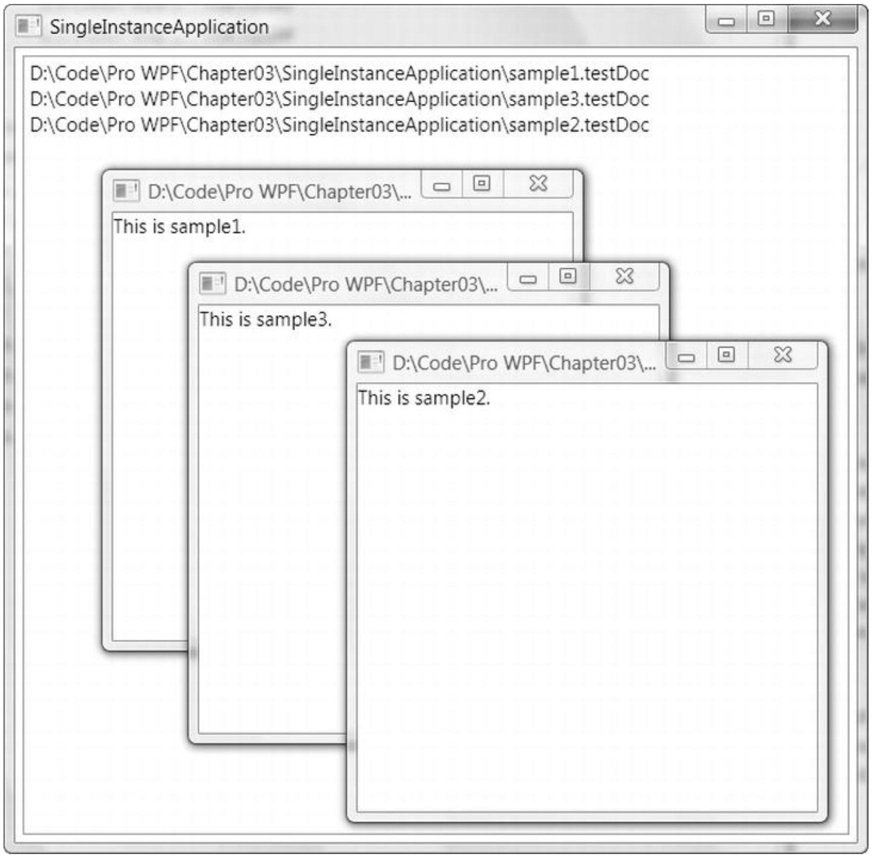
Here's the code for the WpfApp class:

```
public class WpfApp : System.Windows.Application
{
    protected override void OnStartup(System.Windows.StartupEventArgs e)
    {
        base.OnStartup(e);
        WpfApp.current = this;

        // Load the main window.
        DocumentList list = new DocumentList();
        this.MainWindow = list;
        list.Show();

        // Load the document that was specified as an argument.
        if (e.Args.Length > 0) ShowDocument(e.Args[0]);
    }

    public void ShowDocument(string filename)
    {
        try
        {
            Document doc = new Document();
            doc.LoadFile(filename);
            doc.Owner = this.MainWindow;
            doc.Show();

            // If the application is already loaded, it may not be visible.
            // This attempts to give focus to the new window.
            doc.Activate();
        }
        catch
        {
            MessageBox.Show("Could not load document.");
        }
    }
}
```

The only missing detail now (aside from the DocumentList and Document windows) is the entry point for the application. Because the application needs to create the SingleInstanceApplicationWrapper class before the App class, the application must start with a traditional Main() method, rather than an App.xaml file. Here's the code you need:

```
public class Startup
{
    [STAThread]
    public static void Main(string[] args)
    {
        SingleInstanceApplicationWrapper wrapper =
          new SingleInstanceApplicationWrapper();
        wrapper.Run(args);
    }
}
```

These three classes—SingleInstanceApplicationWrapper, WpfApp, and Startup—form the basis for a single-instance WPF application. Using this bare-bones model, it's possible to create a more sophisticated example. For example, the downloadable code for this chapter modifies the WpfApp class so it maintains a list of open documents (as demonstrated earlier). Using WPF data binding with a list (a feature described in Chapter 19), the DocumentList window displays the currently open documents. Figure 7-3 shows an example with three open documents.


SingleInstanceApplication
D:\Code\Pro WPF\Chapter03\...
D:\Code\Pro WPF\Chapter03\...
D:\Code\Pro WPF\Chapter03\...

D:\Code\Pro WPF\Chapter03\SingleInstanceApplication\sample1.testDoc
D:\Code\Pro WPF\Chapter03\SingleInstanceApplication\sample3.testDoc
D:\Code\Pro WPF\Chapter03\SingleInstanceApplication\sample2.testDoc

This is sample1.

This is sample3.

This is sample2.

***Figure 7-3.** A single-instance application with a central window*

■ **Note** Single-instance application support will eventually make its way to WPF in a future version. For now, this workaround provides the same functionality with only a little more work required.

Registering the File Type

To test the single-instance application, you need to register its file extension (.testDoc) with Windows and associate it with your application. That way, when you click a .testDoc file, your application will start up immediately.

One way to create this file-type registration is by hand, using Windows Explorer:

1. Right-click a .testDoc file and choose Open With ➤ Choose Default Program.
2. In the Open With dialog box, click Browse, find your application's .exe file, and double-click it.
3. If you don't want to make your application the default handler for this file type, make sure the option "Always use the selected program to open this type of file" isn't selected in the Open With dialog box. In this case, you won't be able to launch your application by double-clicking the file. However, you will be able to open it by right-clicking the file, choosing Open With, and selecting your application from the list.
4. Click OK.

The other way to create the file-type registration is to run some code that edits the registry. The SingleInstanceApplication example includes a FileRegistrationHelper class that does exactly that:

```
string extension = ".testDoc";
string title = "SingleInstanceApplication";
string extensionDescription = "A Test Document";
FileRegistrationHelper.SetFileAssociation(
  extension, title + "." + extensionDescription);
```

The FileRegistrationHelper registers the .testDoc file extension by using the classes in the Microsoft.Win32 namespace. To see the full code, refer to the downloadable examples for this chapter.

The registration process needs to be executed just once. After the registration is in place, every time you double-click a file with the extension .testDoc, the SingleInstanceApplication is started, and the file is passed as a command-line argument. If the SingleInstanceApplication is already running, the SingleInstanceApplicationWrapper.OnStartupNextInstance() method is called, and the new document is loaded by the existing application.

■ **Tip** When creating a document-based application with a registered file type, you may be interested in using the jump list feature (which is available in Windows 7 or Windows 8). To learn more about this feature, see Chapter 23.

WINDOWS AND UAC

File registration is a task that's usually performed by a setup program. One problem with including it in your application code is that it requires elevated permissions that the user running the application might not have. In fact, even if the user *does* has these privileges, the Windows User Account Control (UAC) feature will still prevent your code from accessing them.

To understand why, you need to realize that in the eyes of UAC, all applications have one of three *run levels*:

Ordinarily, your application runs with the asInvoker run level. To request administrator privileges, you must right-click the application EXE file and choose Run As Administrator when you start it. To get administrator privileges when testing your application in Visual Studio, you must right-click the Visual Studio shortcut and choose Run As Administrator.

If your application needs administrator privileges, you can choose to require them with the requireAdministrator run level or request them with the highestAvailable run level. Either way, you need to create a *manifest*, which is a file with a block of XML that will be embedded in your compiled assembly. To add a manifest, right-click your project in the Solution Explorer and choose Add ➤ New Item. Pick the Application Manifest File template and then click Add.

The content of the manifest file is a relatively simple block of XML:

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
 xmlns="urn:schemas-microsoft-com:asm.v1"
 xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
 xmlns:asmv2="urn:schemas-microsoft-com:asm.v2">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

To change the run level, simply modify the level attribute of the <requestedExecutionLevel> element. Valid values are asInvoker, requireAdministrator, and highestAvailable.

In some cases, you might want to request administrator privileges in specific scenarios. In the file registration example, you might choose to request administrator privileges only when the application is run for the first time and needs to create the registration. This allows you to avoid unnecessary UAC warnings. The easiest way to implement this pattern is to put the code that requires higher privileges in a separate executable, which you can then call when necessary.

Assembly Resources

Assembly resources in a WPF application work in essentially the same way as assembly resources in other .NET applications. The basic concept is that you add a file to your project, so that Visual Studio can embed it into your compiled application's EXE or DLL file. The key difference between WPF assembly resources and those in other applications is the addressing system that you use to refer to them.

■ **Note** Assembly resources are also known as *binary resources* because they're embedded in compiled assembly as an opaque blob of binary data.

You've already seen assembly resources at work in Chapter 2. That's because every time you compile your application, each XAML file in your project is converted to a BAML file that's more efficient to parse.

These BAML files are embedded in your assembly as individual resources. It's just as easy to add your own resources.

Adding Resources

You can add your own resources by adding a file to your project and setting its Build Action property (in the Properties window) to Resource. Here's the good news: that's all you need to do.

For better organization, you can create subfolders in your project (right-click the Solution Explorer and choose Add ➤ New Folder) and use these to organize different types of resources. In the example in Figure 7-4, several image resources are grouped in a folder named Images, and two audio fields appear in a folder named Sounds.

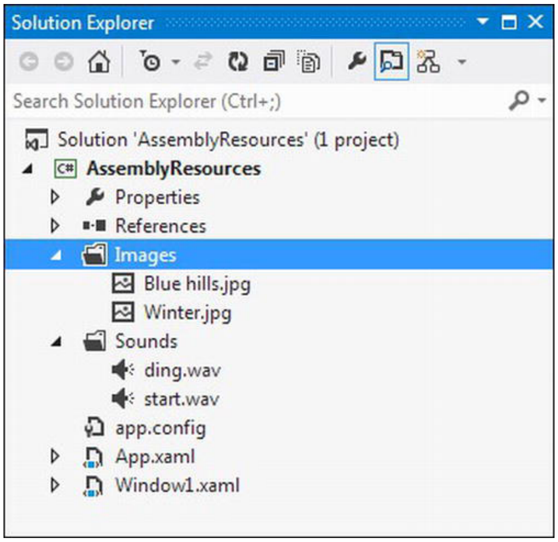


Figure 7-4. *An application with assembly resources*

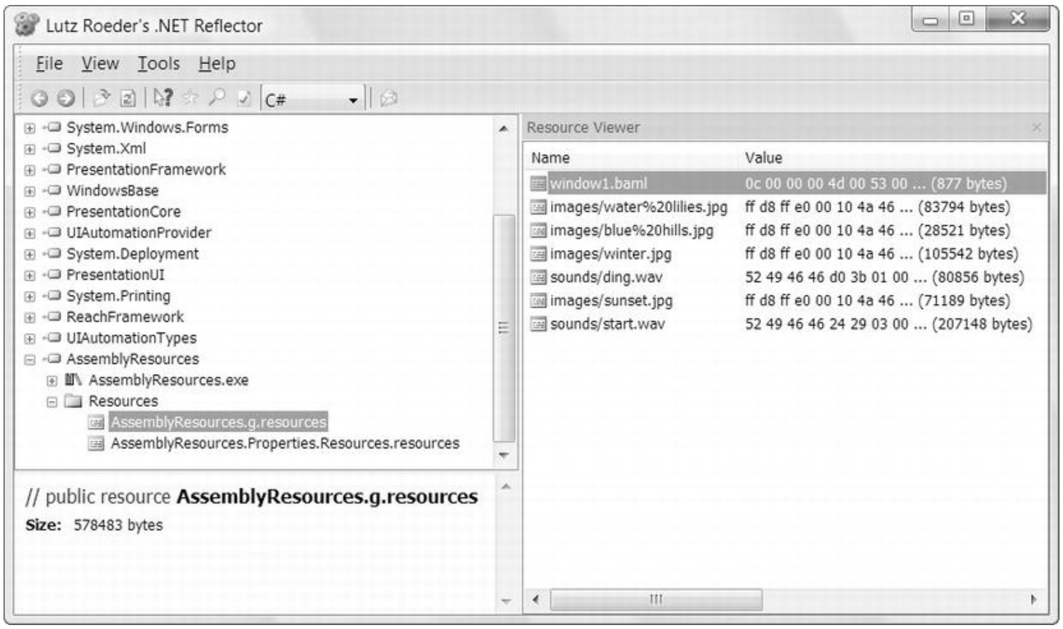
Resources that you add in this way are easy to update. All you need to do is replace the file and recompile your application. For example, if you create the project shown in Figure 7-4, you could copy all new files to the Images folder by using Windows Explorer. As long as you're replacing the contents of files that are included in your project, you don't need to take any special step in Visual Studio (aside from actually compiling your application).

There are a couple of things that you must *not* do in order to use assembly resources successfully:

- Don't make the mistake of setting the Build Action property to Embedded Resource. Even though all assembly resources are embedded resources by definition, the Embedded Resource build action places the binary data in another area where it's more difficult to access. In WPF applications, it's assumed that you always use a build type of Resource.
- Don't use the Resources tab in the Project Properties window. WPF does not support this type of resource URI.

Curious programmers naturally want to know what happens to the resources they embed in their assemblies. WPF merges them all into a single stream (along with BAML resources). This single resource stream is named in this format: *AssemblyName*.g.resources. In Figure 7-5, the application is named AssemblyResources and the resource stream is named AssemblyResources.g.resources.

If you want to actually *see* the embedded resources in a compiled assembly, you can use a disassembler. Unfortunately, the .NET staple—ildasm—doesn't have this feature. However, you can use a more elegant tool such as Reflector (`http://reflector.net`) to dig into your resources. Figure 7-5 shows the resources for the project shown in Figure 7-4, using Reflector.



// public resource **AssemblyResources.g.resources**
Size: 578483 bytes

Name	Value
window1.baml	0c 00 00 00 4d 00 53 00 ... (877 bytes)
images/water%20lilies.jpg	ff d8 ff e0 00 10 4a 46 ... (83794 bytes)
images/blue%20hills.jpg	ff d8 ff e0 00 10 4a 46 ... (28521 bytes)
images/winter.jpg	ff d8 ff e0 00 10 4a 46 ... (105542 bytes)
sounds/ding.wav	52 49 46 46 d0 3b 01 00 ... (80856 bytes)
images/sunset.jpg	ff d8 ff e0 00 10 4a 46 ... (71189 bytes)
sounds/start.wav	52 49 46 46 24 29 03 00 ... (207148 bytes)

Figure 7-5. *Assembly resources in .NET Reflector*

You'll see the BAML resource for the only window in the application, along with all the images and audio files. The spaces in the file names don't cause a problem in WPF, because Visual Studio is intelligent enough to escape them properly. You'll also notice that the file names are changed to lowercase when your application is compiled.

Retrieving Resources

Adding resources is clearly easy enough, but how do you actually *use* them? There's more than one approach that you can use.

The low-level choice is to retrieve a StreamResourceInfo object that wraps your data, and then decide what to do with it. You can do this through code, using the static Application.GetResourceStream() method.

For example, here's the code that gets the StreamResourceInfo object for the winter.jpg image:

```
StreamResourceInfo sri = Application.GetResourceStream(
  new Uri("images/winter.jpg", UriKind.Relative));
```

Once you have a StreamResourceInfo object, you can get two pieces of information. The ContentType property returns a string describing the type of data—in this example, it's image/jpg. The Stream property returns an UnmanagedMemoryStream object, which you can use to read the data, one byte at a time.

The GetResourceStream() method is really just a helper method that wraps a ResourceManager and ResourceSet classes. These classes are a core part of the .NET Framework resource system, and they've existed since version 1.0. Without the GetResourceStream() method, you would need to specifically access the AssemblyName.g.resources resource stream (which is where all WPF resources are stored) and search for the object you want. Here's the far uglier code that does the trick:

```
Assembly assembly = Assembly.GetAssembly(this.GetType());
string resourceName = assembly.GetName().Name + ".g";
ResourceManager rm = new ResourceManager(resourceName, assembly);

using (ResourceSet set =
  rm.GetResourceSet(CultureInfo.CurrentCulture, true, true))
{
    UnmanagedMemoryStream s;

    // The second parameter (true) performs a case-insensitive resource lookup.
    s = (UnmanagedMemoryStream)set.GetObject("images/winter.jpg", true);
    ...
}
```

The ResourceManager and ResourceSet classes also allow you to do a few things you can't do with the Application class alone. For example, the following snippet of code shows you the name of all the embedded resources in the AssemblyName.g.resources stream:

```
Assembly assembly = Assembly.GetAssembly(this.GetType());
string resourceName = assembly.GetName().Name + ".g";
ResourceManager rm = new ResourceManager(resourceName, assembly);

using (ResourceSet set =
  rm.GetResourceSet(CultureInfo.CurrentCulture, true, true))
{
    foreach (DictionaryEntry res in set)
    {
        MessageBox.Show(res.Key.ToString());
    }
}
```

Resource-Aware Classes

Even with the help of the GetResourceStream() method, you're unlikely to bother retrieving a resource directly. The problem is that this approach gets you a relatively low-level UnmanagedMemoryStream object, which isn't much use on its own. Instead, you'll want to translate the data into something more meaningful, such as a higher-level object with properties and methods.

WPF provides a few classes that work with resources natively. Rather than forcing you to do the work of resource extraction (which is messy and not typesafe), they take the name of the resource you want to use. For example, if you want to show the Blue hills.jpg image in the WPF Image element, you could use this markup:

```
<Image Source="Images/Blue hills.jpg"></Image>
```

Notice that the backslash becomes a forward slash because that's the convention WPF uses with its URIs. (It works *both* ways, but the forward slash is recommended for consistency.)

You can perform the same trick in code. In the case of an Image element, you simply need to set the Source property with a BitmapImage object that identifies the location of the image you want to display as a URI. You could specify a fully qualified file path like this:

```
img.Source = new BitmapImage(new Uri(@"d:\Photo\Backgrounds\arch.jpg"));
```

But if you use a relative URI, you can pull a different resource out of the assembly and pass it to the image, with no UnmanagedMemoryStream object required:

```
img.Source = new BitmapImage(new Uri("images/winter.jpg", UriKind.Relative));
```

This technique constructs a URI that consists of the base application URI with images/winter.jpg added on the end. Most of the time, you don't need to think about this URI syntax—as long as you stick to relative URIs, it all works seamlessly. However, in some cases it's important to understand the URI system in a bit more detail, particularly if you want to access a resource that's embedded in another assembly. The following section digs into WPF's URI syntax.

Pack URIs

WPF lets you address compiled resources (such as the BAML for a page) by using the *pack URI* syntax. The Image and tag in the previous section referenced a resource by using a relative URI, like this:

```
images/winter.jpg
```

This is equivalent to the more cumbersome absolute URI shown here:

```
pack://application:,,,/images/winter.jpg
```

You can use this absolute URI when setting the source of an image, although it doesn't provide any advantage:

```
img.Source = new BitmapImage(new Uri("pack://application:,,,/images/winter.jpg"));
```

■ **Tip** When using an absolute URI, you can use a file path, a UNC path to a network share, a website URL, or a pack URI that points to an assembly resource. Just be aware that if your application can't retrieve the resource from the expected location, an exception will occur. If you've set the URI in XAML, the exception will happen when the page is being created.

The pack URI syntax is borrowed from the XML Paper Specification (XPS) standard. The reason it looks so strange is that it embeds one URI inside another. The three commas are actually three escaped slashes. In other words, the pack URI shown previously contains an application URI that starts with `application:///`.

Resources in Other Assemblies

Pack URIs also allow you to retrieve resources that are embedded in another library (in other words, in a DLL assembly that your application uses). In this case, you need to use the following syntax:

```
pack://application:,,,/AssemblyName;component/ResourceName
```

For example, if your image is embedded in a referenced assembly named ImageLibrary, you would use a URI like this:

```
img.Source = new BitmapImage(
  new Uri("pack://application:,,,/ImageLibrary;component/images/winter.jpg"));
```

Or, more practically, you would use the equivalent relative URI:

```
img.Source = new BitmapImage(
  new Uri("ImageLibrary;component/images/winter.jpg", UriKind.Relative));
```

If you're using a strong-named assembly, you can replace the assembly name with a qualified assembly reference that includes the version, the public key token, or both. You separate each piece of information by using a semicolon and precede the version number with the letter *v*. Here's an example with just a version number:

```
img.Source = new BitmapImage(
  new Uri("ImageLibrary;v1.25;component/images/winter.jpg",
  UriKind.Relative));
```

And here's an example with both the version number and the public-key token:

```
img.Source = new BitmapImage(
  new Uri("ImageLibrary;v1.25;dc642a7f5bd64912;component/images/winter.jpg",
  UriKind.Relative));
```

Content Files

When you embed a file as a resource, you place it into the compiled assembly and ensure it's always available. This is an ideal choice for deployment, and it side-steps possible problems. However, there are some situations where it isn't practical:

- You want to change the resource file without recompiling the application.
- The resource file is very large.
- The resource file is optional and may not be deployed with the assembly.
- The resource is a sound file.

■ **Note** As you'll discover in Chapter 26, the WPF sound classes don't support assembly resources. As a result, there's no way to pull an audio file out of a resource stream and play it—at least not without saving it first. This is a limitation of the underlying bits of technology on which these classes are based (namely, the Win32 API and Media Player).

Obviously, you can deal with this issue by deploying the files with your application and adding code to your application to read these files from the hard drive. However, WPF has a convenient option that can make this process easier to manage. You can specifically mark these noncompiled files as *content* files.

Content files won't be embedded in your assembly. However, WPF adds an AssemblyAssociatedContentFile attribute to your assembly that advertises the existence of each content file. This attribute also records the location of each content file relative to your executable file (indicating whether the content file is in the same folder as the executable file or in a subfolder). Best of all, you can use the same URI system to use content files with resource-aware elements such as the Image class.

To try this out, add a sound file to your project, select it in the Solution Explorer, and change the Build Action in the Properties window to Content. Make sure that the Copy to Output Directory setting is set to Copy Always, so that the sound file is copied to the output directory when you build your project.

Now you can use a relative URI to point a MediaElement to your content file:

```
<MediaElement Name="Sound" Source="Sounds/start.wav"
  LoadedBehavior="Manual"></MediaElement>
```

To see an application that uses both application resources and content files, check out the downloadable code for this chapter.

Localization

Assembly resources also come in handy when you need to localize a window. Using resources, you allow controls to change according to the current culture settings of the Windows operating system. This is particularly useful with text labels and images that need to be translated into different languages.

In some frameworks, localization is performed by providing multiple copies of user-interface details such as string tables and images. In WPF, localization isn't this fine-grained. Instead, the unit of localization is the XAML file (technically, the compiled BAML resource that's embedded in your application). If you want to support three languages, you need to include three BAML resources. WPF chooses the correct one based on the current culture on the computer that's executing the application. (Technically, WPF bases its decision on the CurrentUICulture property of the thread that's hosting the user interface.)

Of course, this process wouldn't make much sense if you need to create (and deploy) an all-in-one assembly with *all* the localized resources. This wouldn't be much better than creating separate versions of your application for every language, because you would need to rebuild your entire application every time you wanted to add support for a new culture (or if you needed to tweak the text in one of the existing resources). Fortunately, .NET solves this problem by using *satellite assemblies*, which are assemblies that work with your application but are stored in separate subfolders. When you create a localized WPF application, you place each localized BAML resource in a separate satellite assembly. To allow your application to use this assembly, you place it in a subfolder under the main application folder, such as fr-FR for French (France). Your application can then bind to this satellite assembly automatically using a technique called *probing*, which has been a part of the .NET Framework since version 1.0.

The challenge in localizing an application is in the workflow—in other words, how do you pull your XAML files out of your project, get them localized, compile them into satellite assemblies, and then bring them back to your application? This is the shakiest part of the localization story in WPF, because tools such as Visual Studio don't have built-in design support for localization. That means you'll need to do a bit more work, and you may want to create your own localization tools.

Building Localizable User Interfaces

Before you begin to translate anything, you need to consider how your application will respond to changing content. For example, if you double the length of all the text in your user interface, how will the overall layout of your window be adjusted? If you've built a truly adaptable layout (as described in Chapter 3), you shouldn't have a problem. Your interface should be able to adjust itself to fit dynamic content. Some good practices that suggest you're on the right track include the following:

- Not using hard-coded widths or heights (or at least not using them with elements that contain nonscrollable text content)

- Setting the Window.SizeToContent property to Width, Height, or WidthAndHeight so it can grow as needed (not always required, depending on the structure of your window, but sometimes useful)
- Using the ScrollViewer to wrap large amounts of text

OTHER CONSIDERATIONS FOR LOCALIZATION

Depending on the languages in which you want to localize your application, there are other considerations that you might need to take into account. Although a discussion of user interface layout in different languages is beyond the scope of this book, here are a couple issues to consider:

Localization is a complex topic. WPF has a solution that's workable, but it's not fully mature. After you've learned the basics, you may want to take a look at Microsoft's slightly dated, but still useful, 66-page WPF localization white paper, which is available at `http://wpflocalization.codeplex.com` along with sample code.

Preparing an Application for Localization

The next step is to switch on localization support for your project. This takes just one change—add the following element to the .csproj file for your project anywhere in the first <PropertyGroup> element:

```
<UICulture>en-US</UICulture>
```

This tells the compiler that the default culture for your application is US English (obviously, you could choose something else if that's appropriate). Once you make this change, the build process changes. The next time you compile your application, you'll end up with a subfolder named en-US. Inside that folder is a satellite assembly with the same name as your application and the extension .resources.dll (for example, LocalizableApplication.resources.dll). This assembly contains all the compiled BAML resources for your application, which were previously stored in your main application assembly.

UNDERSTANDING CULTURES

Technically, you don't localize an application for a specific language but for a *culture*, which takes into account regional variation. Cultures are identified by two identifiers separated by a hyphen. The first portion identifies the language. The second portion identifies the country. Thus, fr-CA is French as spoken in Canada, while fr-FR represents French in France. For a full list of culture names and their two-part identifiers, refer to the System.Globalization.CultureInfo class in the MSDN help (`http://tinyurl.com/cuyhe6p`).

This presumes a fine-grained localization that might be more than you need. Fortunately, you can localize an application based just on a language. For example, if you want to define settings that will be used for any French-language region, you could use fr for your culture. This works as long as there isn't a more specific culture available that matches the current computer exactly.

Now, when you run this application, the CLR automatically looks for satellite assemblies in the correct directory, based on the computer's regional settings, and loads the correct localized resource. For example, if you're running in the fr-FR culture, the CLR will look for an fr-FR subdirectory and use the satellite assemblies it finds there. So, if you want to add support for more cultures to a localized application, you simply need to add more subfolders and satellite assemblies without disturbing the original application executable.

When the CLR begins probing for a satellite assembly, it follows a few simple rules of precedence:

1. It checks for the most specific directory that's available. That means it looks for a satellite assembly that's targeted for the current language and region (such as fr-FR).
2. If it can't find this directory, it looks for a satellite assembly that's targeted for the current language (such as fr).
3. If it can't find this directory, an IOException exception is thrown.

This list is slightly simplified. If you decide to use the Global Assembly Cache (GAC) to share some components over the entire computer, you'll need to realize that .NET actually checks the GAC at the beginning of step 1 and step 2. In other words, in step 1, the CLR checks whether the language- and region-specific version of the assembly is in the GAC and uses it if it is. The same is true for step 2.

Managing the Translation Process

Now you have all the infrastructure you need for localization. All you need to do is create the appropriate satellite assemblies with the alternate versions of your windows (in BAML form) and put these assemblies in the correct folders. Doing this by hand would obviously be a lot of work. Furthermore, localization usually involves a third-party translation service that needs to work with your original text. Obviously, it's too much to expect that your translators will be skilled programmers who can find their way around a Visual Studio project (and you're unlikely to trust them with the code anyway). For all these reasons, you need a way to manage the localization process.

Currently, WPF has a partial solution. It works, but it requires a few trips to the command line, and one piece isn't finalized. The basic process works like this:

1. You flag the elements in your application that need to be localized. Optionally, you may add comments to help the translator.
2. You extract the localizable details to a .csv file (a comma-separated text file) and send it off to your translation service.
3. After you receive the translated version of this file, you run LocBaml again to generate the satellite assembly you need.

You'll follow these steps in the following sections.

Preparing Markup Elements for Localization

The first step is to add a specialized Uid attribute to all the elements you want to localize. Here's an example:

```
<Button x:Uid="Button_1" Margin="10" Padding="3">A button</Button>
```

The Uid attribute plays a role similar to that of the Name attribute—it uniquely identifies a button in the context of a single XAML document. That way, you can specify localized text for just this button. However, there are a few reasons why WPF uses a Uid instead of just reusing the Name value: the name might not be assigned, it might be set according to different conventions and used in code, and so on. In fact, the Name property is itself a localizable piece of information.

■ **Note** Obviously, text isn't the only detail you need to localize. You also need to think about fonts, font sizes, margins, padding, other alignment-related details, and so on. In WPF, every property that may need to be localized is decorated with the System.Windows.LocalizabilityAttribute.

Although you don't need to, you should add the Uid to *every* element in every window of a localizable application. This could add up to a lot of extra work, but the MSBuild tool can do it automatically. Use it like this:

```
msbuild /t:updateuid LocalizableApplication.csproj
```

This assumes you wish to add Uids to an application named LocalizableApplication.

And if you want to check whether your elements all have Uids (and make sure you haven't accidentally duplicated one), you can use MSBuild like this:

```
msbuild /t:checkuid LocalizableApplication.csproj
```

■ **Tip** The easiest way to run MSBuild is to launch the Visual Studio Command Prompt (Start ▯ All Programs ▯ Microsoft Visual Studio 2012 ▯ Visual Studio Tools ▯ Developer Command Prompt for VS2012) so that the path is set to give you easy access. Then you can quickly move to your project folder to run MSBuild.

When you generate Uids by using MSBuild, your Uids are set to match the name of the corresponding control. Here's an example:

```
<Button x:Uid="cmdDoSomething" Name="cmdDoSomething"  Margin="10" Padding="3">
```

If your element doesn't have a name, MSBuild creates a less helpful Uid based on the class name, with a numeric suffix:

```
<TextBlock x:Uid="TextBlock_1" Margin="10">
```

■ **Note** Technically, this step is how you globalize an application—in other words, prepare it for localization into different languages. Even if you don't plan to localize your application right away, there's an argument to be made that you should prepare it for localization anyway. If you do, you may be able to update your application to a different language simply by deploying a satellite assembly. Of course, globalization is not worth the effort if you haven't taken the time to assess your user interface and make sure it uses an adaptable layout that can accommodate changing content (such as buttons with longer captions, and so on).

Extracting Localizable Content

To extract the localizable content of all your elements, you need to use the LocBaml command-line tool. Currently, LocBaml isn't included as a compiled tool. Instead, the source code is available as a sample at `http://tinyurl.com/df3bqg` (look for the "LocBaml Tool Sample" link). The LocBaml sample must be compiled by hand.

When using LocBaml, you *must* be in the folder that contains your compiled assembly (for example, LocalizableApplication\bin\Debug). To extract a list of localizable details, you point LocBaml to your satellite assembly and use the /parse parameter, as shown here:

```
locbaml /parse en-US\LocalizableApplication.resources.dll
```

The LocBaml tool searches your satellite assembly for all its compiled BAML resources and generates a .csv file that has the details. In this example, the .csv file will be named LocalizationApplication. resources.csv.

Each line in the extracted file represents a single localizable property that you've used on an element in your XAML document. Each line consists of the following seven values:

- The name of the BAML resource (for example, LocalizableApplication.g.en-US. resources:window1.baml).
- The Uid of the element and the name of the property to localize. Here's an example: StackPanel_1:System.Windows.FrameworkElement.Margin.
- The localization category. This is a value from the LocalizationCategory enumeration that helps to identify the type of content that this property represents (long text, a title, a font, a button caption, a tooltip, and so on).
- Whether the property is readable (essentially, visible as text in the user interface). All readable values always need to be localized; nonreadable values may or may not require localization.
- Whether the property value can be modified by the translator. This value is always True unless you specifically indicate otherwise.
- Additional comments that you've provided for the translator. If you haven't provided comments, this value is blank.
- The value of the property. This is the detail that needs to be localized.

For example, imagine you have the window shown in Figure 7-6. Here's the XAML markup:

```
<Window x:Uid="Window_1" x:Class="LocalizableApplication.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="LocalizableApplication" Height="300" Width="300"
    SizeToContent="WidthAndHeight"
    >
  <StackPanel x:Uid="StackPanel_1" Margin="10">
    <TextBlock x:Uid="TextBlock_1" Margin="10">One line of text.</TextBlock>
    <Button x:Uid="cmdDoSomething" Name="cmdDoSomething"  Margin="10" Padding="3">
     A button</Button>
    <TextBlock x:Uid="TextBlock_2" Margin="10">
     This is another line of text.</TextBlock>
  </StackPanel>
</Window>
```

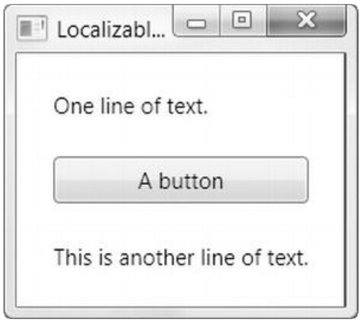


Figure 7-6. A window that can be localized

When you run this through LocBaml, you'll get the information shown in Table 7-3. (For the sake of brevity, the BAML name has been left out, because it's always the same window; the resource key has been shortened so it doesn't use fully qualified names; and the comments, which are blank, have been left out.)

Here's where the current tool support is a bit limited. It's unlikely that a translation service will want to work directly with the .csv file, because it presents information in a rather awkward way. Instead, another tool is needed that parses this file and allows the translator to review it more efficiently. You could easily build a tool that pulls out all this information, displays the values where Readable and Modifiable are true, and allows the user to edit the corresponding value. However, at the time of this writing, WPF doesn't include such a tool.

To perform a simple test, you can open this file directly (use Notepad or Excel) and modify the last piece of information—the value—to supply translated text instead. Here's an example:

```
LocalizableApplication.g.en-US.resources:window1.baml,
TextBlock_1:System.Windows.Controls.TextBlock.$Content,
Text,True,True,,
Une ligne de texte.
```

■ **Note** Although this is really a single line of code, it's broken here to fit on the page.

You don't specify which culture you're using at this point. You do that when you compile the new satellite assembly in the next step.

Table 7-3. A Sample List of Localizable Properties

Resource Key	Localization Category	Readable	Modifiable	Value
Window_1:LocalizableApplication. Window1.$Content	None	True	True	#StackPanel_1;
Window_1:Window.Title	Title	True	True	LocalizableApplication
Window_1:FrameworkElement.Height	None	False	True	300
Window_1:FrameworkElement.Width	None	False	True	300
Window_1:Window.SizeToContent	None	False	True	WidthAndHeight
StackPanel_1:FrameworkElement.Margin	None	False	True	10

Resource Key	Localization Category	Readable	Modifiable	Value
TextBlock_1:TextBlock.$Content	Text	True	True	One line of text
TextBlock_1:FrameworkElement.Margin	None	False	True	10
cmdDoSomething:Button.$Content	Button	True	True	A button
cmdDoSomething:FrameworkElement.Margin	None	False	True	10
cmdDoSomething:Padding	None	False	True	3
TextBlock_2:TextBlock.$Content	Text	True	True	Another line of text
TextBlock_2:FrameworkElement.Margin	None	False	True	10

Building a Satellite Assembly

Now you're ready to build the satellite assemblies for other cultures. Once again, the LocBaml tool takes care of this task, but this time, you use the /generate parameter.

Remember that the satellite assembly will contain an alternate copy of each *complete* window as an embedded BAML resource. In order to create these resources, the LocBaml tool needs to take a look at the original satellite assembly, substitute all the new values from the translated .csv file, and then generate a new satellite assembly. That means you need to point LocBaml to the original satellite assembly and (using the /trans: parameter) the translated list of values. You also need to tell LocBaml which culture this assembly represents (using the /cul: parameter). Remember that cultures are defined using two-part identifiers that are listed in the description of the System.Globalization.CultureInfo class.

Here's an example that pulls it all together:

```
locbaml /generate en-US\LocalizableApplication.resources.dll
        /trans:LocalizableApplication.resources.French.csv
        /cul:fr-FR /out:fr-FR
```

This command does the following:

- Uses the original satellite assembly en-US\LocalizedApplication.resources.dll.
- Uses the translates .csv file French.csv.
- Uses the France French culture.
- Outputs to the fr-FR subfolder (which must already exist). Though this seems implicit based on the culture you're using, you need to supply this detail.

When you run this command line, LocBaml creates a new version of the LocalizableApplication.resources.dll assembly with the translated values and places it in the fr-FR subfolder of the application.

Now when you run the application on a computer that has its culture set to France French, the alternate version of the window will be shown automatically. You can change the culture by using the Regional and Language Options section of the Control Panel. Or for an easier approach to testing, you can use code to change the culture of the current thread. You need to do this before you create or show any windows, so it makes sense to use an application event, or just use your application class constructor, as shown here:

```
public partial class App : System.Windows.Application
{
    public App()
    {
        Thread.CurrentThread.CurrentUICulture =
          new CultureInfo("fr-FR");
    }
}
```

Figure 7-7 shows the result.

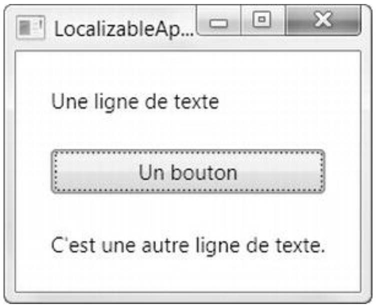
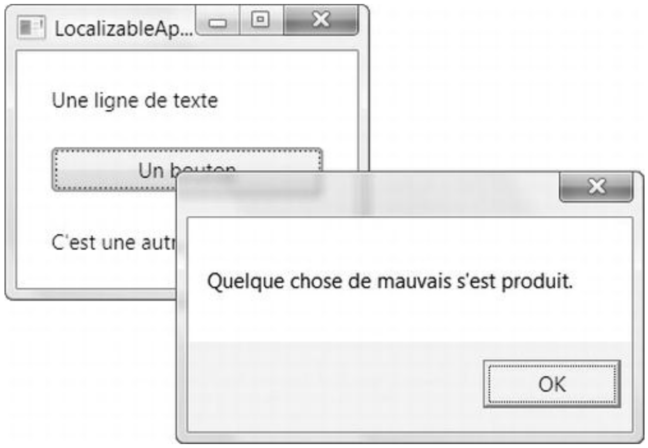


Figure 7-7. *A window that's localized in French*

Not all localizable content is defined as a localizable property in your user interface. For example, you might need to show an error message when something occurs. The best way to handle this situation is to use XAML resources (as described in Chapter 10). For example, you could store your error message strings as resources in a specific window, in the resources for an entire application, or in a resource dictionary that's shared across multiple applications. Here's an example:

```
<Window.Resources>
  <s:String x:Uid="s:String_1" x:Key="Error">Something bad happened.</s:String>
</Window.Resources>
```

When you run LocBaml, the strings in this file are also added to the content that needs to be localized. When compiled, this information is added to the satellite assembly, ensuring that error messages are in the correct language (as shown in Figure 7-8).



Une ligne de texte

C'est une autr

Quelque chose de mauvais s'est produit.

Figure 7-8. *Using a localized string*

■ **Note** An obvious weakness in the current system is that it's difficult to keep up with an evolving user interface. The LocBaml tool always creates a new file, so if you end up moving controls to different windows or replacing one control with another, you'll probably be forced to create a new list of translations from scratch.

The Last Word

In this chapter, you took a detailed look at the WPF application model.

To manage a simple WPF application, you need to do nothing more than create an instance of the Application class and call the Run() method. However, most applications go further and derive a custom class from the Application class. And as you saw, this custom class is an ideal tool for handling application events and an ideal place to track the windows in your application or implement a single-instance pattern.

In the second half of this chapter, you considered assembly resources that allow you to package binary data and embed it in your application. You also took a look at localization and learned how a few command-line tools (msbuild.exe and locbaml.exe) allow you to provide culture-specific versions of your user interface, albeit with a fair bit of manual labor.