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.

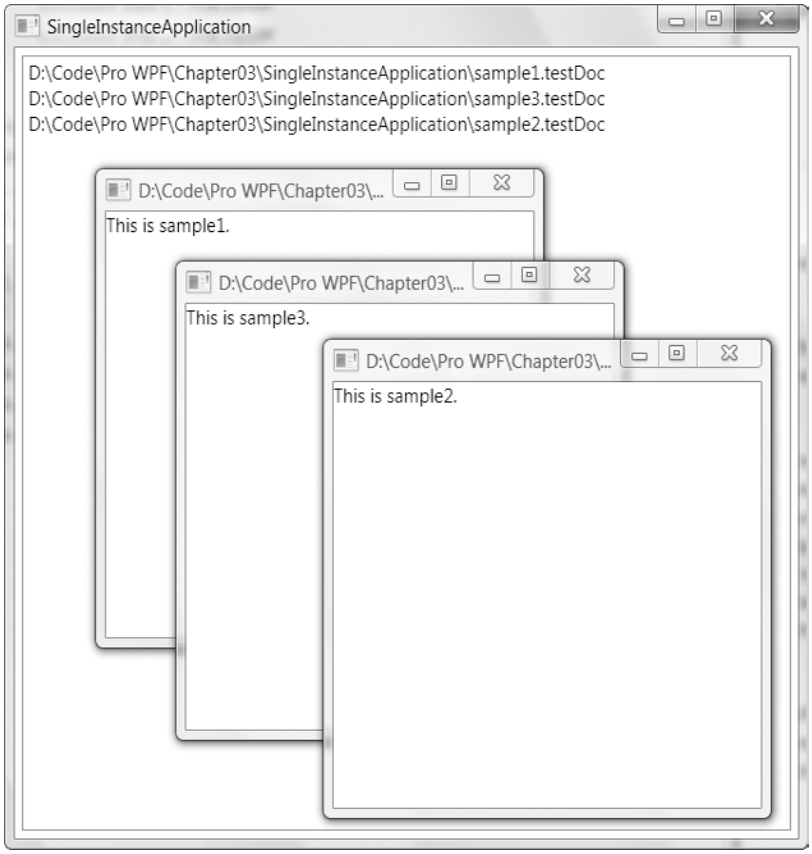

Figure 7-3. *A single-instance application with a central window*

■ **Note** Single-instance application support will eventually make its way to WPF in a future version. For now, this work-around 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 checked 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 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 in Windows 7. 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. This is particularly a problem with the User Account Control (UAC) feature in Windows Vista and Windows 7. In fact, by default, this code will fail with a security-related exception.

In the eyes of UAC, all applications have one of three *run levels*:

- **asInvoker.** The application inherits the process token of the parent process (the process that launched it). The application won't get administrator privileges unless the user specifically requests them, even if the user is logged on as an administrator. This is the default.
- **requireAdministrator.** If the current user is a member of the Administrators group, a UAC confirmation dialog box appears. Once the user accepts this confirmation, the application gets administrator privileges. If the user is not a member of the Administrators group, a dialog box appears where the user can enter the user name and password of an account that does have administrator privileges.
- **highestAvailable.** The application gets the highest privileges according to its group membership. For example, if the current user is a member of the Administrators group, the application gets administrator privileges (once the user accepts the UAC confirmation). The advantage of this run level is that the application will still run if administrator privileges aren't available, unlike requireAdministrator.

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 the 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. Figure 7-4 shows an example where several image resources are grouped in a folder named Images, and two audio fields appear in a folder named Sounds.

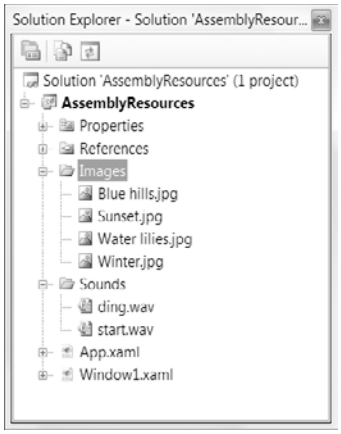

Figure 7-4. *An application with assembly resources*

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 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 11-1, 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 download the free and much more elegant .NET Reflector tool (`http://www.red-gate.com/products/reflector`), which does let you dig into your resources. Figure 7-5 shows the resources for the project shown in Figure 7-4, using .NET Reflector.

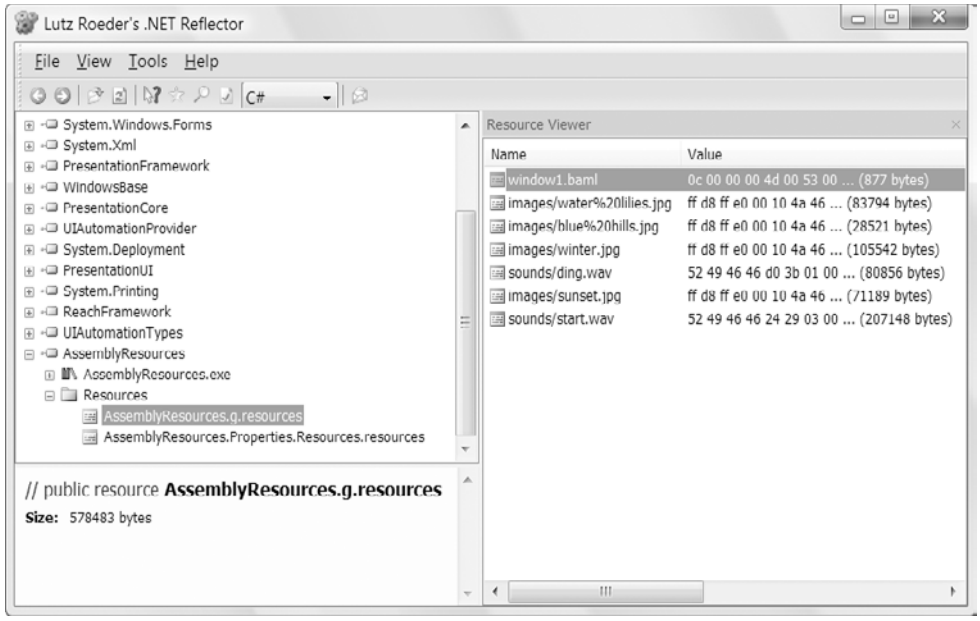

Figure 7-5. *Assembly resources in .NET Reflector*

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 actually 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) using the *pack URI* syntax. The Image and tag in the previous section referenced a resource 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 because 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 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 different 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 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 there aren't yet any tools (including Visual Studio) that have design support for localization. It's likely that better tools will emerge in the future, but WPF still gives you everything you need to localize your application with a bit more work.

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:

- If you want to localize your application into a language that has a dramatically different character set, you'll need to use a different font. You can do this by localizing the FontFamily property in your user interface, or you can use a composite font such as Global User Interface, Global Sans Serif, or Global Serif, which support all languages.
- You may also need to think about how your layout works in a right-to-left layout (rather than the standard English left-to-right layout). For example, Arabic and Hebrew use a right-to-left layout. You can control this behavior by setting the FlowDirection property on each page or window in your application. For more information about right-to-left layouts, see the "Bidirectional Features" topic in the Visual Studio help.

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 66-page WPF localization whitepaper, which is available at `http://wpflocalization.codeplex.com` along with sample code. And expect Microsoft to improve the support for localization in design tools like Visual Studio and Expression Blend in the future.

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 U.S. 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 Visual Studio help.

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. First, 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.

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 ➤ Programs ➤ Microsoft Visual Studio 2008 ➤ Visual Studio Tools ➤ Visual Studio 2008 Command Prompt) 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 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/yf7jvum`. It 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>
```

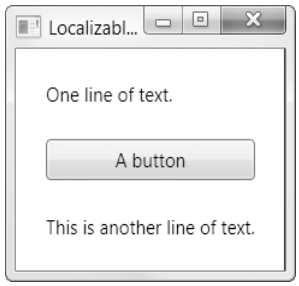

Figure 7-6. *A window that can be localized*

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.$Content	None	True	True	#StackPanel_1;
Window_1:Window.Title	Title	True	True	LocalizableApplication
Window_1:FrameworkElement.Height	None	False	True	300
Window_1:FrameworkElement.Width	None	False	True	300
Window_1:Window.SizeToContent	None	False	True	WidthAndHeight
StackPanel_1:FrameworkElement.Margin	None	False	True	10
TextBlock_1:TextBlock.$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 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).

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.

CHAPTER 8

Element Binding

At its simplest, data binding is a relationship that tells WPF to extract some information from a *source* object and use it to set a property in a *target* object. The target property is always a dependency property, and it's usually in a WPF element—after all, the ultimate goal of WPF data binding is to display some information in your user interface. However, the source object can be just about anything, ranging from another WPF element to an ADO.NET data object (like the DataTable and DataRow) or a data-only object of your own creation.

In this chapter, you'll begin your exploration of data binding by considering the simplest approach: element-to-element binding. In Chapter 19, you'll revisit the data binding story, and learn the most efficient way to shuttle data from a database to your data forms.

Binding Elements Together

The simplest data binding scenario occurs when your source object is a WPF element and your source property is a dependency property. That's because dependency properties have built-in support for change notification, as explained in Chapter 4. As a result, when you change the value of the dependency property in the source object, the bound property in the target object is updated immediately. This is exactly what you want, and it happens without requiring you to build any additional infrastructure.

Note Although it's nice to know that element-to-element binding is the simplest approach, most developers are more interested in finding out which approach is most common in the real world. Overall, the bulk of your data binding work will be spent binding elements to data objects. This allows you to display the information that you've extracted from an external source (such as a database or file). However, element-to-element binding is often useful. For example, you can use element-to-element binding to automate the way elements interact so that when a user modifies a control, another element is updated automatically. This is a valuable shortcut that can save you from writing boilerplate code (and it's a technique that wasn't possible in the previous generation of Windows Forms applications).
