Name	Description
IsSnapToTickEnabled	If true, when you move the slider, it automatically snaps into place, jumping to the nearest tick mark. The default is false.
IsSelectionRangeEnabled	If true, you can use a selection range to shade in a portion of the slider bar. You set the position selection range using the SelectionStart and SelectionEnd properties. The selection range has no intrinsic meaning, but you can use it for whatever purpose makes sense. For example, media players sometimes use a shaded background bar to indicate the download progress for a media file.

Figure 6-20 compares Slider controls with different tick settings.

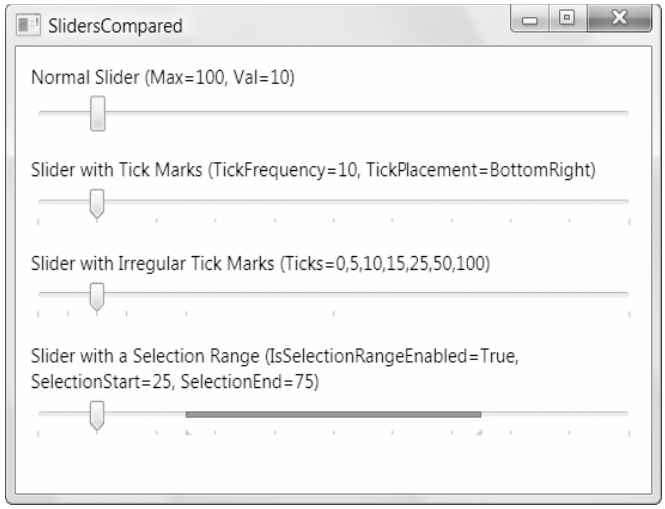

Figure 6-20. *Adding ticks to a slider*

The ProgressBar

The ProgressBar indicates the progress of a long-running task. Unlike the slider, the ProgressBar isn't user-interactive. Instead, it's up to your code to periodically increment the Value property. (Technically speaking, WPF rules suggest the ProgressBar shouldn't be a control because it doesn't respond to mouse actions or keyboard input.) The ProgressBar has a minimum height of four device-independent units. It's up to you to set the Height property (or put it in the appropriate fixed-size container) if you want to see a larger, more traditional bar.

One neat trick that you can perform with the ProgressBar is using it to show a long-running status indicator, even if you don't know how long the task will take. Interestingly (and oddly), you do this by setting the IsIndeterminate property to true:

```
<ProgressBar Height="18"  Width="200" IsIndeterminate="True"></ProgressBar>
```

When setting IsIndeterminate, you no longer use the Minimum, Maximum, and Value properties. Instead, this ProgressBar shows a periodic green pulse that travels from left to right, which is the universal Windows convention indicating that there's work in progress. This sort of indicator makes sense in an application's status bar. For example, you could use it to indicate that you're contacting a remote server for information.

Date Controls

WPF 4 adds two date controls: the Calendar and the DatePicker. Both are designed to allow the user to choose a single date.

The Calendar control displays a calendar that's similar to what you see in the Windows operating system (for example, when you configure the system date). It shows a single month at a time and allows you to step through from month to month (by clicking the arrow buttons) or jump to a specific month (by clicking the month header to view an entire year, and then clicking the month).

The DatePicker requires less space. It's modeled after a simple text box, which holds a date string in long or short date format. The DatePicker provides a drop-down arrow that, when clicked, pops open a full calendar view that's identical to that shown by the Calendar control. This pop-up is displayed over top of any other content, just like a drop-down combo box.

Figure 6-21 shows the two display modes that the Calendar supports, as well as the two date formats that the DatePicker allows.

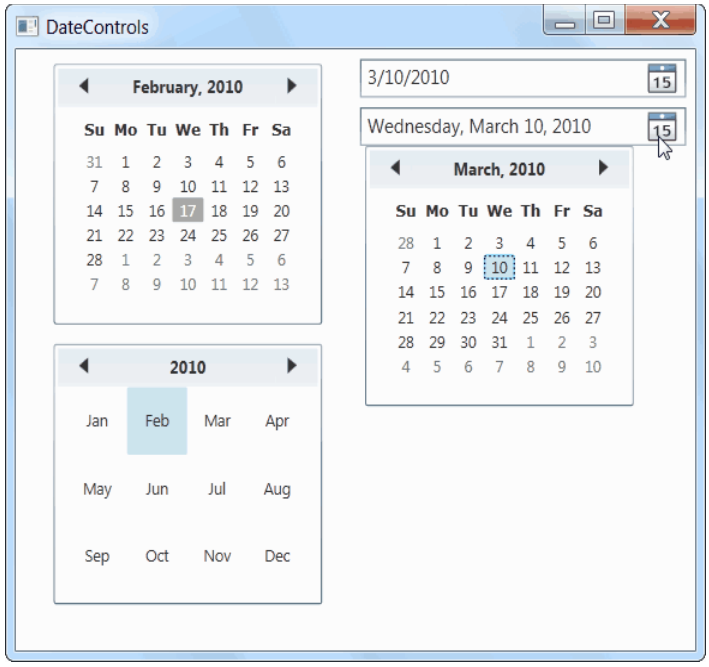

Figure 6-21. *The Calendar and DatePicker*

The Calendar and DatePicker include properties that allow you to determine which dates are shown and which dates are selectable (provided they fall in a contiguous range). Table 6-6 lists the properties you can use.

Table 6-6. *Properties of the Calendar and DatePicker Classes*

Property	Description
DisplayDateStart and DisplayDateEnd	Set the range of dates that are displayed in the calendar view, from the first, earliest date (DisplayDateStart) to the last, most recent date (DisplayDateEnd). The user won't be able to navigate to months that don't have any displayable dates. To show all dates, set DisplayDateStart to DateTime.MinValue and DisplayDateEnd to DateTime.MaxValue.
BlackoutDates	Holds a collection of dates that will be disabled in the calendar and won't be selectable. If these dates are not in the range of displayed dates, or if one of these dates is already selected, you'll receive an exception. To prevent selection of any date in the past, call the BlackoutDates.AddDatesInPast() method.
SelectedDate	Provides the selected date as a DateTime object (or a null value if no date is selected). It can be set programmatically, by the user clicking the date in the calendar, or by the user typing in a date string (in the DatePicker). In the calendar view, the selected date is marked by a shaded square, which is visible only when the date control has focus.
SelectedDates	Provides the selected dates as a collection of DateTime objects. This property is supported by the Calendar, and it's useful only if you've changed the SelectionMode property to allow multiple date selection.
DisplayDate	Determines the date that's displayed initially in the calendar view (using a DateTime object). If it's a null, the SelectedDate is shown. If DisplayDate and SelectedDate are both null, the current date is used. The display date determines the initial month page of the calendar view. When the date control has focus, a square outline is displayed around the appropriate day in that month (which is different from the shaded square used for the currently selected date).
FirstDayOfWeek	Determines the day of the week that will be displayed at the start of each calendar row, in the leftmost position.
IsTodayHighlighted	Determines whether the calendar view uses highlighting to point out the current date.

Property	Description
DisplayMode (Calendar only)	Determines the initial display month of the calendar. If set to Month, the Calendar shows the standard single-month view. If set to Year, the Calendar shows the months in the current year (similar to when the user clicks the month header). Once the user clicks a month, the Calendar shows the full calendar view for that month.
SelectionMode (Calendar only)	Determines the type of date selections that are allowed. The default is SingleDate, which allows a single date to be selected. Other options include None (selection is disabled entirely), SingleRange (a contiguous group of dates can be selected), and MultipleRange (any combination of dates can be selected). In SingleRange or MultipleRange modes, the user can drag to select multiple dates, or click while holding down the Ctrl key. You can use the SelectedDates property to get a collection with all the selected dates.
IsDropDownOpen (DatePicker only)	Determines whether the calendar view drop-down is open in the DatePicker. You can set this property programmatically to show or hide the calendar.
SelectedDateFormat (DatePicker only)	Determines how the selected date will be displayed in the text part of the DatePicker. You can choose Short or Long. The actual display format is based on the client computer's regional settings. For example, if you use Short, the date might be rendered in the yyyy/mm/dd format or dd/mm/yyyy. The long format generally includes the month and day names.

The date controls also provide a few different events. Most useful is SelectedDateChanged (in the DatePicker) or the similar SelectedDatesChanged (in the Calendar), which adds support for multiple date selection. You can react to these events to reject specific date selections, such as dates that fall on a weekend:

```
private void Calendar_SelectedDatesChanged (object sender,
  CalendarDateChangedEventArgs e)
{
    // Check all the newly added items.
    foreach (DateTime selectedDate in e.AddedItems)
    {
        if ((selectedDate.DayOfWeek == DayOfWeek.Saturday) ||
          (selectedDate.DayOfWeek == DayOfWeek.Sunday))
        {
            lblError.Text = "Weekends are not allowed";

            // Remove the selected date.
            ((Calendar)sender).SelectedDates.Remove(selectedDate);
        }
    }
}
```

You can try this out with a Calendar that supports single or multiple selection. If it supports multiple selection, try dragging the mouse over an entire week of dates. All the dates will remain highlighted except for the disallowed weekend dates, which will be unselected automatically.

The Calendar also adds a DisplayDateChanged event (when the user browses to a new month). The DatePicker adds CalendarOpened and CalendarClosed events (which fire when the calendar drop-down is displayed and closed) and a DateValidationError event (which fires when the user types a value in the text-entry portion that can't be interpreted as a valid date). Ordinarily, invalid values are discarded when the user opens the calendar view, but here's an option that fills in some text to inform the user of the problem:

```
private void DatePicker_DateValidationError(object sender,
  DatePickerDateValidationErrorEventArgs e)
{
     lblError.Text = "'" + e.Text +
       "' is not a valid value because " + e.Exception.Message;
}
```

The Last Word

In this chapter, you took a tour of the fundamental WPF controls, including basic ingredients such as labels, buttons, text boxes, and lists. Along the way, you learned about some important WPF concepts that underlie the control model, such as brushes, fonts, and the content model. Although most WPF controls are quite easy to use, developers who have this additional understanding—and know how all the different branches of WPF elements relate together—will have an easier time creating well-designed windows.

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 like 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.

What's New The only change to the application model in WPF 4 is the introduction of an underwhelming splash screen feature, which is described in the section "Showing a Splash Screen."

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 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* is 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 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 an 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 it 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 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 like 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.
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 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 (as 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 (and Windows Forms programmers will recognize it well). 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 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 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** If you're a seasoned Windows Forms programmer, 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 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 several different 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 like 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.

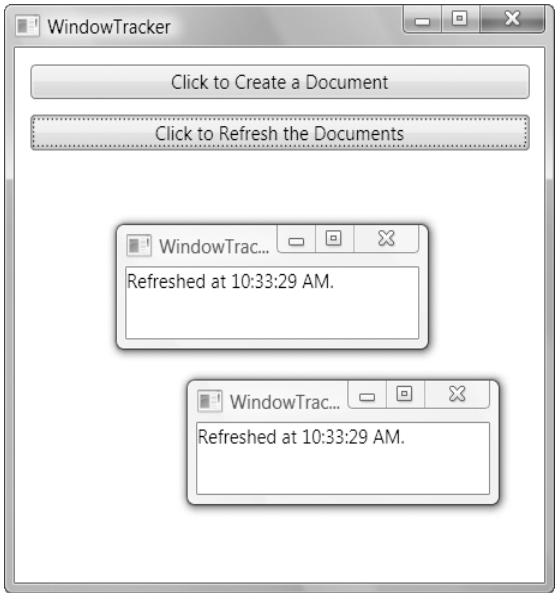

Figure 7-1. *Allowing windows to interact*

■ **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 work-arounds. 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.

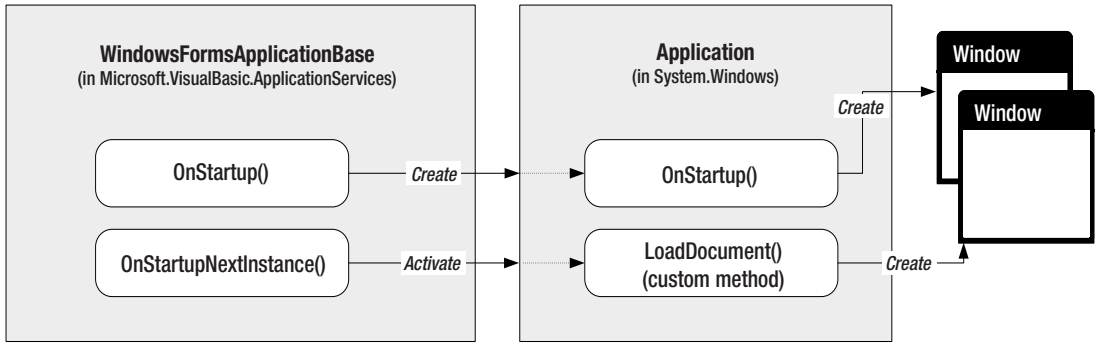

Figure 7-2. *Wrapping the WPF application with a WindowsFormsApplicationBase*

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().