to HwndSource (for example, the airspace rule, focus quirks, and so on). In fact, WindowsFormsHost derives from HwndHost.

The HwndHost is your gateway to the traditional world of C++ and MFC applications. However, it also allows you to integrate managed DirectX content. Currently, WPF does not include any DirectX interoperability features, and you can't use the DirectX libraries to render content in a WPF window. However, you can use DirectX to build a separate window and then host that inside a WPF window using the HwndHost. Although DirectX is far beyond the scope of this book (and an order of magnitude more complex than WPF programming), you can download the managed DirectX libraries at `http://msdn.microsoft.com/directx`.

The complement of HwndHost is the HwndSource class. While HwndHost allows you to place any hwnd in a WPF window, HwndSource wraps any WPF visual or element in an hwnd so it can be inserted in a Win32-based application, such as an MFC application. The only limitation is that your application needs a way to access the WPF libraries, which are managed .NET code. This isn't a trivial task. If you're using a C++ application, the simplest approach is to use the Managed Extensions for C++. You can then create your WPF content, create an HwndSource to wrap it, set the HwndHost.RootVisual property to the top-level element, and then place the HwndSource into your window.

You'll find much more content to help you with complex integration projects and legacy code online and in the Visual Studio help.

# The Last Word

In this chapter you considered the interoperability support that allows WPF applications to show Windows Forms content (and vice versa). Then you examined the WindowsFormsHost element, which lets you embed a Windows Forms control in a WPF window, and the ElementHost, which lets you embed a WPF element in a form. Both of these classes provide a simple, effective way to manage the transition from Windows Forms to WPF.

CHAPTER 31

# Multithreading

As you've discovered over the previous 30 chapters, WPF revolutionizes almost all the conventions of Windows programming. It introduces a new approach to everything from defining the content in a window to rendering 3D graphics. WPF even introduces a few new concepts that aren't obviously UI-focused, such as dependency properties and routed events.

Of course, a great number of coding tasks fall outside the scope of user interface programming and haven't changed in the WPF world. For example, WPF applications use the same classes as other .NET applications when contacting databases, manipulating files, and performing diagnostics. There are also a few features that fall somewhere between traditional .NET programming and WPF. These features aren't strictly limited to WPF applications, but they do have specific WPF considerations. One example is the *add-in model*, which allows your WPF application to dynamically load and use separately compiled components with useful bits of functionality. (It's described in the next chapter.) And in this chapter, you'll look at *multithreading*, which allows your WPF application to perform background work while keeping a responsive user interface.

---

**Note** Both multithreading and the add-in model are advanced topics that could occupy an entire book worth of material; therefore, you won't get an exhaustive examination of either feature in this book. However, you will get the basic outline you need to use them with WPF, and you'll establish a solid foundation for future exploration.

---

## Multithreading

Multithreading is the art of executing more than one piece of code at once. The goal of multithreading is usually to create a more responsive interface—one that doesn't freeze up while it's in the midst of other work—although you can also use multithreading to take better advantage of dual-core CPUs when executing a processor-intensive algorithm or to perform other work during a high-latency operation (for example, to perform some calculations while waiting for a response from a web service).

Early in the design of WPF, the creators considered a new threading model. This model—called *thread rental*—allowed user interface objects to be accessed on any thread. To reduce the cost of locking, groups of related objects could be grouped under a single lock (called a *context*). Unfortunately, this design introduced additional complexity for single-threaded applications (which needed to be context-aware) and made it more difficult to interoperate with legacy code (like the Win32 API). Ultimately, the plan was abandoned.

The result is that WPF supports a *single-threaded apartment* model that's very much like the one used in Windows Forms applications. It has a few core rules:

- WPF elements have *thread affinity*. The thread that creates them owns them, and other threads can't interact with them directly. (An *element* is a WPF object that's displayed in a window.)
- WPF objects that have thread affinity derive from DispatcherObject at some point in their class hierarchy. DispatcherObject includes a small set of members that allow you to verify whether code is executing on the right thread to use a specific object and (if not) switch it over.
- In practice, one thread runs your entire application and owns all WPF objects. Although you could use separate threads to show separate windows, this design is rare.

In the following sections, you'll explore the DispatcherObject class and learn the simplest way to perform an asynchronous operation in a WPF application.

## The Dispatcher

A *dispatcher* manages the work that takes place in a WPF application. The dispatcher owns the application thread and manages a queue of work items. As your application runs, the dispatcher accepts new work requests and executes one at a time.

Technically, a dispatcher is created the first time you instantiate a class that derives from DispatcherObject on a new thread. If you create separate threads and use them to show separate windows, you'll wind up with more than one dispatcher. However, most applications keep things simple and stick to one user interface thread and one dispatcher. They then use multithreading to manage data operations and other background tasks.

---

■ **Note** The dispatcher is an instance of the System.Windows.Threading.Dispatcher class. All the dispatcher-related objects are also found in the small System.Windows.Threading namespace, which is new to WPF. (The core threading classes that have existed since .NET 1.0 are found in System.Threading.)

---

You can retrieve the dispatcher for the current thread using the static Dispatcher.CurrentDispatcher property. Using this Dispatcher object, you can attach event handlers that respond to unhandled exceptions or respond when the dispatcher shuts down. You can also get a reference to the System.Threading.Thread that the dispatcher controls, shut down the dispatcher, or marshal code to the correct thread (a technique you'll see in the next section).

## The DispatcherObject

Most of the time, you won't interact with a dispatcher directly. However, you'll spend plenty of time using instances of DispatcherObject, because every visual WPF object derives from this class. A DispatcherObject is simply an object that's linked to a dispatcher—in other words, an object that's bound to the dispatcher's thread.

The DispatcherObject introduces just three members, as listed in Table 31-1.

*Table 31-1. Members of the DispatcherObject Class*

| Name | Description |
|---|---|
| Dispatcher | Returns the dispatcher that's managing this object |
| CheckAccess() | Returns true if the code is on the right thread to use the object; returns false otherwise |
| VerifyAccess() | Does nothing if the code is on the right thread to use the object; throws an InvalidOperationException otherwise |

WPF objects call VerifyAccess() frequently to protect themselves. They don't call VerifyAccess() in response to every operation (because that would impose too great a performance overhead), but they do call it often enough that you're unlikely to use an object from the wrong thread for very long.

For example, the following code responds to a button click by creating a new System.Threading.Thread object. It then uses that thread to launch a small bit of code that changes a text box in the current window.

```
private void cmdBreakRules_Click(object sender, RoutedEventArgs e)
{
    Thread thread = new Thread(UpdateTextWrong);
    thread.Start();
}

private void UpdateTextWrong()
{
    // Simulate some work taking place with a five-second delay.
    Thread.Sleep(TimeSpan.FromSeconds(5));

    txt.Text = "Here is some new text.";
}
```

This code is destined to fail. The UpdateTextWrong() method will be executed on a new thread, and that thread isn't allowed to access WPF objects. In this case, the TextBox object catches the violation by calling VerifyAccess(), and an InvalidOperationException is thrown.

To correct this code, you need to get a reference to the dispatcher that owns the TextBox object (which is the same dispatcher that owns the window and all the other WPF objects in the application). Once you have access to that dispatcher, you can call Dispatcher.BeginInvoke() to marshal some code to the dispatcher thread. Essentially, BeginInvoke() schedules your code as a task for the dispatcher. The dispatcher then executes that code.

Here's the corrected code:

```
private void cmdFollowRules_Click(object sender, RoutedEventArgs e)
{
    Thread thread = new Thread(UpdateTextRight);
    thread.Start();
}

private void UpdateTextRight()
```

```
{
    // Simulate some work taking place with a five-second delay.
    Thread.Sleep(TimeSpan.FromSeconds(5));

    // Get the dispatcher from the current window, and use it to invoke
    // the update code.
    this.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
      (ThreadStart) delegate() {
                      txt.Text = "Here is some new text.";
                    }
      );
}
```

The Dispatcher.BeginInvoke() method takes two parameters. The first indicates the priority of the task. In most cases, you'll use DispatcherPriority.Normal, but you can also use a lower priority if you have a task that doesn't need to be completed immediately and that should be kept on hold until the dispatcher has nothing else to do. For example, this might make sense if you need to display a status message about a long-running operation somewhere in your user interface. You can use DispatcherPriority.ApplicationIdle to wait until the application has finished all other work or the even more laid-back DispatcherPriority.SystemIdle to wait until the entire system is at rest and the CPU is idle.

You can also use an above-normal priority to get the dispatcher's attention right away. However, it's recommended that you leave higher priorities to input messages (such as key presses). These need to be handled nearly instantaneously, or the application will feel sluggish. On the other hand, adding a few milliseconds of extra time to a background operation won't be noticeable, so a priority of DispatcherPriority.Normal makes more sense in this situation.

The second BeginInvoke() parameter is a delegate that points to the method with the code you want to execute. This could be a method somewhere else in your code, or you can use an anonymous method to define your code inline (as in this example). The inline approach works well for simple operations, like this single-line update. However, if you need to use a more complex process to update the user interface, it's a good idea to factor this code into a separate method.

---

■ **Note** The BeginInvoke() method also has a return value, which isn't used in the earlier example. BeginInvoke() returns a DispatcherOperation object, which allows you to follow the status of your marshaling operation and determine when your code has actually been executed. However, the DispatcherOperation is rarely useful, because the code you pass to BeginInvoke() should take very little time.

---

Remember, if you're performing a time-consuming background operation, you need to perform this operation on a separate thread and *then* marshal its result to the dispatcher thread (at which point you'll update the user interface or change a shared object). It makes no sense to perform your time-consuming code in the method that you pass to BeginInvoke(). For example, this slightly rearranged code still works but is impractical:

```
private void UpdateTextRight()
{
    // Get the dispatcher from the current window.
    this.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
```

```
        (ThreadStart) delegate() {
                        // Simulate some work taking place.
                        Thread.Sleep(TimeSpan.FromSeconds(5));

                        txt.Text = "Here is some new text.";
                      }
        );
}
```

The problem here is that all the work takes place on the dispatcher thread. That means this code ties up the dispatcher in the same way a non-multithreaded application would.

---

■ **Note** The dispatcher also provides an Invoke() method. Like BeginInvoke(), Invoke() marshals the code you specify to the dispatcher thread. But unlike BeginInvoke(), Invoke() stalls your thread until the dispatcher executes your code. You might use Invoke() if you need to pause an asynchronous operation until the user has supplied some sort of feedback. For example, you could call Invoke() to run a snippet of code that shows an OK/Cancel dialog box. After the user clicks a button and your marshaled code completes, the Invoke() method will return, and you can act upon the user's response.

---

# The BackgroundWorker

You can perform asynchronous operations in many ways. You've already seen one no-frills approach—creating a new System.Threading.Thread object by hand, supplying your asynchronous code, and launching it with the Thread.Start() method. This approach is powerful, because the Thread object doesn't hold anything back. You can create dozens of threads at will, set their priorities, control their status (for example, pausing, resuming, and aborting them), and so on. However, this approach is also a bit dangerous. If you access shared data, you need to use locking to prevent subtle errors. If you create threads frequently or in large numbers, you'll generate additional, unnecessary overhead.

The techniques to write good multithreading code—and the .NET classes you'll use—aren't WPF-specific. If you've written multithreaded code in a Windows Forms application, you can use the same techniques in the WPF world. In the remainder of this chapter, you'll consider one of the simplest and safest approaches: the System.ComponentModel.BackgroundWorker component.

---

■ **Tip** To see several different approaches, ranging from simple to more complex, you may want to refer to my book *Programming .NET 2.0 Windows Forms and Custom Controls in C#* (Apress, 2005).

---

The BackgroundWorker was introduced in .NET 2.0 to simplify threading considerations in Windows Forms applications. However, the BackgroundWorker is equally at home in WPF. The BackgroundWorker component gives you a nearly foolproof way to run a time-consuming task on a separate thread. It uses the dispatcher behind the scenes and abstracts away the marshaling issues with an event-based model.

As you'll see, the BackgroundWorker also supports two frills: progress events and cancel messages. In both cases the threading details are hidden, making for easy coding.

---

■ **Note** The BackgroundWorker is perfect if you have a single asynchronous task that runs in the background from start to finish (with optional support for progress reporting and cancellation). If you have something else in mind—for example, an asynchronous task that runs throughout the entire life of your application or an asynchronous task that communicates with your application while it does its work, you'll need to design a customized solution using .NET's threading support.

---

## A Simple Asynchronous Operation

To try the BackgroundWorker, it helps to consider a sample application. The basic ingredient for any test is a time-consuming process. The following example uses a common algorithm for finding prime numbers in a given range called the *sieve of Eratosthenes*, which was invented by Eratosthenes himself in about 240 BC. With this algorithm, you begin by making a list of all the integers in a range of numbers. You then strike out the multiples of all primes less than or equal to the square root of the maximum number. The numbers that are left are the primes.

In this example, I won't go into the theory that proves the sieve of Eratosthenes works or show the fairly trivial code that performs it. (Similarly, don't worry about optimizing it or comparing it against other techniques.) However, you will see how to perform the sieve of Eratosthenes algorithm asynchronously.

The full code is available with the online examples for this chapter. It takes this form:

```
public class Worker
{
    public static int[] FindPrimes(int fromNumber, int toNumber)
    {
        // Find the primes between fromNumber and toNumber,
        // and return them as an array of integers.
    }
}
```

The FindPrimes() method takes two parameters that delimit a range of numbers. The code then returns an integer array with all the prime numbers that occur in that range.

Figure 31-1 shows the example we're building. This window allows the user to choose the range of numbers to search. When the user clicks Find Primes, the search begins, but it takes place in the background. When the search is finished, the list of prime numbers appears in the list box.

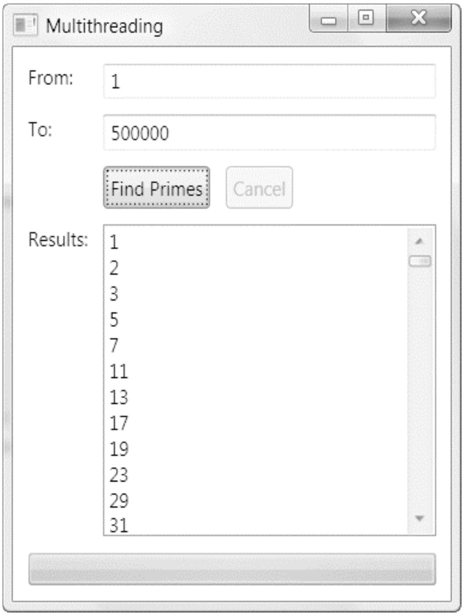


***Figure 31-1.*** *A completed prime number search*

## Creating the BackgroundWorker

To use the BackgroundWorker, you begin by creating an instance. Here, you have two options:

- You can create the BackgroundWorker in your code and attach all the event handlers programmatically.
- You can declare the BackgroundWorker in your XAML. The advantage of this approach is that you can hook up your event handlers using attributes. Because the BackgroundWorker isn't a visible WPF element, you can't place it just anywhere. Instead, you need to declare it as a resource for your window.

Both approaches are equivalent. The downloadable sample uses the second approach. The first step is to make the System.ComponentModel namespace accessible in your XAML document through a namespace import. To do this, you need to map the namespace to an XML prefix:

```
<Window x:Class="Multithreading.BackgroundWorkerTest"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:cm="clr-namespace:System.ComponentModel;assembly=System"
    ... >
```

Now you can create an instance of the BackgroundWorker in the Window.Resources collection. When doing this, you need to supply a key name so the object can be retrieved later. In this example, the key name is backgroundWorker:

```
<Window.Resources>
  <cm:BackgroundWorker x:Key="backgroundWorker"></cm:BackgroundWorker>
</Window.Resources>
```

The advantage of declaring the BackgroundWorker in the Window.Resources section is that you can set its properties and attach its event handlers using attributes. For example, here's the BackgroundWorker tag you'll end up with at the end of this example, which enables support for progress notification and cancellation and attaches event handlers to the DoWork, ProgressChanged, and RunWorkerCompleted events:

```
<cm:BackgroundWorker x:Key="backgroundWorker"
  WorkerReportsProgress="True" WorkerSupportsCancellation="True"
  DoWork="backgroundWorker_DoWork"
  ProgressChanged="backgroundWorker_ProgressChanged"
  RunWorkerCompleted="backgroundWorker_RunWorkerCompleted">
</cm:BackgroundWorker>
```

To get access to this resource in your code, you need to pull it out of the Resources collection. In this example, the window performs this step in its constructor so that all your event handling code can access it more easily:

```
public partial class BackgroundWorkerTest : Window
{
    private BackgroundWorker backgroundWorker;

    public BackgroundWorkerTest()
    {
        InitializeComponent();
        backgroundWorker =
          ((BackgroundWorker)this.FindResource("backgroundWorker"));
    }

    ...
}
```

## Running the BackgroundWorker

The first step to using the BackgroundWorker with the prime number search example is to create a custom class that allows you to transmit the input parameters to the BackgroundWorker. When you call BackgroundWorker.RunWorkerAsync(), you can supply any object, which will be delivered to the DoWork event. However, you can supply only a single object, so you need to wrap the to and from numbers into one class, as shown here:

```
public class FindPrimesInput
{
    public int From
    { get; set; }

    public int To
    { get; set; }

    public FindPrimesInput(int from, int to)
```

```
    {
        From = from;
        To = to;
    }
}
```

To start the BackgroundWorker on its way, you need to call the BackgroundWorker.RunWorkerAsync() method and pass in the FindPrimesInput object. Here's the code that does this when the user clicks the Find Primes button:

```
private void cmdFind_Click(object sender, RoutedEventArgs e)
{
    // Disable this button and clear previous results.
    cmdFind.IsEnabled = false;
    cmdCancel.IsEnabled = true;
    lstPrimes.Items.Clear();

    // Get the search range.
    int from, to;
    if (!Int32.TryParse(txtFrom.Text, out from))
    {
        MessageBox.Show("Invalid From value.");
        return;
    }
    if (!Int32.TryParse(txtTo.Text, out to))
    {
        MessageBox.Show("Invalid To value.");
        return;
    }

    // Start the search for primes on another thread.
    FindPrimesInput input = new FindPrimesInput(from, to);
    backgroundWorker.RunWorkerAsync(input);
}
```

When the BackgroundWorker begins executing, it grabs a free thread from the CLR thread pool and then fires the DoWork event from this thread. You handle the DoWork event and begin your time-consuming task. However, you need to be careful not to access shared data (such as fields in your window class) or user interface objects. Once the work is complete, the BackgroundWorker fires the RunWorkerCompleted event to notify your application. This event fires on the dispatcher thread, which allows you to access shared data and your user interface, without incurring any problems.

Once the BackgroundWorker acquires the thread, it fires the DoWork event. You can handle this event to call the Worker.FindPrimes() method. The DoWork event provides a DoWorkEventArgs object, which is the key ingredient for retrieving and returning information. You retrieve the input object through the DoWorkEventArgs.Argument property and return the result by setting the DoWorkEventArgs.Result property.

```
private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // Get the input values.
    FindPrimesInput input = (FindPrimesInput)e.Argument;

    // Start the search for primes and wait.
```

```
    // This is the time-consuming part, but it won't freeze the
    // user interface because it takes place on another thread.
    int[] primes = Worker.FindPrimes(input.From, input.To);

    // Return the result.
    e.Result = primes;
}
```

Once the method completes, the BackgroundWorker fires the RunWorkerCompletedEventArgs on the dispatcher thread. At this point, you can retrieve the result from the RunWorkerCompletedEventArgs.Result property. You can then update the interface and access window-level variables without worry.

```
private void backgroundWorker_RunWorkerCompleted(object sender,
  RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        // An error was thrown by the DoWork event handler.
        MessageBox.Show(e.Error.Message, "An Error Occurred");
    }
    else
    {
        int[] primes = (int[])e.Result;
        foreach (int prime in primes)
        {
            lstPrimes.Items.Add(prime);
        }
    }

    cmdFind.IsEnabled = true;
    cmdCancel.IsEnabled = false;
    progressBar.Value = 0;
}
```

Notice that you don't need any locking code, and you don't need to use the Dispatcher.BeginInvoke() method. The BackgroundWorker takes care of these issues for you.

Behind the scenes, the BackgroundWorker uses a few multithreading classes that were introduced in .NET 2.0, including AsyncOperationManager, AsyncOperation, and SynchronizationContext. Essentially, the BackgroundWorker uses AsyncOperationManager to manage the background task. The AsyncOperationManager has some built-in intelligence—namely, it's able to get the synchronization context for the current thread. In a Windows Forms application, the AsyncOperationManager gets a WindowsFormsSynchronizationContext object, whereas a WPF application gets a DispatcherSynchronizationContext object. Conceptually, these classes do the same job, but their internal plumbing is different.

## Tracking Progress

The BackgroundWorker also provides built-in support for tracking progress, which is useful for keeping the client informed about how much work has been completed in a long-running task.

To add support for progress, you need to first set the BackgroundWorker.WorkerReportsProgress property to true. Actually, providing and displaying the progress information is a two-step affair. First, the DoWork event handling code needs to call the BackgroundWorker.ReportProgress() method and provide an estimated percent complete (from 0% to 100%). You can do this as little or as often as you like. Every time you call ReportProgress(), the BackgroundWorker fires the ProgressChanged event. You can react to this event to read the new progress percentage and update the user interface. Because the ProgressChanged event fires from the user interface thread, there's no need to use Dispatcher.BeginInvoke().

The FindPrimes() method reports progress in 1% increments, using code like this:

```
int iteration = list.Length / 100;
for (int i = 0; i < list.Length; i++)
{
    ...

    // Report progress only if there is a change of 1%.
    // Also, don't bother performing the calculation if there
    // isn't a BackgroundWorker or if it doesn't support
    // progress notifications.
    if ((i % iteration == 0) &&
      (backgroundWorker != null) && backgroundWorker.WorkerReportsProgress)
    {
        backgroundWorker.ReportProgress(i / iteration);
    }
}
```

Once you've set the BackgroundWorker.WorkerReportsProgress property, you can respond to these progress notifications by handling the ProgressChanged event. In this example, a progress bar is updated accordingly:

```
private void backgroundWorker_ProgressChanged(object sender,
  ProgressChangedEventArgs e)
{
    progressBar.Value = e.ProgressPercentage;
}
```

Figure 31-2 shows the progress meter while the task is in progress.

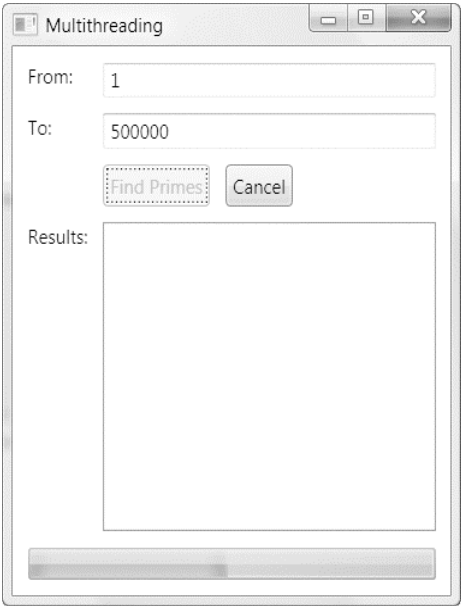


***Figure 31-2.*** *Tracking progress for an asynchronous task*

## Supporting Cancellation

It's just as easy to add support for canceling a long-running task with the BackgroundWorker. The first step is to set the BackgroundWorker.WorkerSupportsCancellation property to true.

To request a cancellation, your code needs to call the BackgroundWorker.CancelAsync() method. In this example, the cancellation is requested when a Cancel button is clicked:

```
private void cmdCancel_Click(object sender, RoutedEventArgs e)
{
    backgroundWorker.CancelAsync();
}
```

Nothing happens automatically when you call CancelAsync(). Instead, the code that's performing the task needs to explicitly check for the cancel request, perform any required cleanup, and return. Here's the code in the FindPrimes() method that checks for cancellation requests just before it reports progress:

```
for (int i = 0; i < list.Length; i++)
{
    ...
    if ((i % iteration) && (backgroundWorker != null))
    {
        if (backgroundWorker.CancellationPending)
        {
            // Return without doing any more work.
```

```
                return;
            }

            if (backgroundWorker.WorkerReportsProgress)
            {
                backgroundWorker.ReportProgress(i / iteration);
            }
        }
    }
```

The code in your DoWork event handler also needs to explicitly set the DoWorkEventArgs.Cancel property to true to complete the cancellation. You can then return from that method without attempting to build up the string of primes.

```
private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    FindPrimesInput input = (FindPrimesInput)e.Argument;
    int[] primes = Worker.FindPrimes(input.From, input.To,
      backgroundWorker);

    if (backgroundWorker.CancellationPending)
    {
        e.Cancel = true;
        return;
    }

    // Return the result.
    e.Result = primes;
}
```

Even when you cancel an operation, the RunWorkerCompleted event still fires. At this point, you can check whether the task was canceled and handle it accordingly.

```
private void backgroundWorker_RunWorkerCompleted(object sender,
  RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        MessageBox.Show("Search cancelled.");
    }
    else if (e.Error != null)
    {
        // An error was thrown by the DoWork event handler.
        MessageBox.Show(e.Error.Message, "An Error Occurred");
    }
    else
    {
        int[] primes = (int[])e.Result;
        foreach (int prime in primes)
        {
            lstPrimes.Items.Add(prime);
        }
```

```
    }
    cmdFind.IsEnabled = true;
    cmdCancel.IsEnabled = false;
    progressBar.Value = 0;
}
```

Now the BackgroundWorker component allows you to start a search and end it prematurely.

# The Last Word

To design a safe and stable multithreading application, you need to understand WPF's threading rules. In this chapter, you explored these rules and learned how to safely update controls from other threads. You also saw how to build in progress notification, provide cancellation support, and make multithreading easy with BackgroundWorker.

CHAPTER 32

# The Add-in Model

Add-ins (also known as *plug-ins*) are separately compiled components that your application can find, load, and use dynamically. Often, an application is designed to use add-ins so that it can be enhanced in the future without needing to be modified, recompiled, and retested. Add-ins also give you the flexibility to customize separate instances of an application for a particular market or client. But the most common reason to use the add-in model is to allow third-party developers to extend the functionality of your application. For example, add-ins in Adobe Photoshop provide a wide range of picture-processing effects. Add-ins in Firefox provide enhanced web surfing features and entirely new functionality. In both cases, the add-ins are created by third-party developers.

Since .NET 1.0, developers have had all the technology they need to create their own add-in system. The two basic ingredients are *interfaces* (which allow you to define the contracts through which the application interacts with the add-in and the add-in interacts with the application) and *reflection* (which allows your application to dynamically discover and load add-in types from a separate assembly). However, building an add-in system from scratch requires a fair bit of work. You need to devise a way to locate add-ins, and you need to ensure that they're managed correctly (in other words, that they execute in a restricted security context and can be unloaded when necessary).

Fortunately, .NET now has a prebuilt add-in model that saves you the trouble. It uses interfaces and reflection, like the add-in model that you'd probably write yourself. However, it handles the low-level plumbing for tedious tasks such as discovery and hosting. In this chapter, you'll learn how to use the add-in model in a WPF application.

## Choosing Between MAF and MEF

Before you can get started building an extensible application with add-ins, you need to deal with an unexpected headache. Namely, .NET doesn't have just one add-in framework; it has two.

.NET 3.5 introduced an add-in model called the Managed Add-in Framework (MAF). But to make matters even more interesting (and a whole lot more confusing), .NET 4 added a new model called the Managed Extensibility Framework (MEF). Developers, who had to roll their own add-in system not long ago, suddenly have two completely separate technologies that share the same ground. So, what's the difference?

MAF is the more robust framework of the two. It allows you to decouple your add-ins from your application so they depend on nothing more than the interface you define. This gives you welcome flexibility if you want to handle versioning scenarios—for example, if you need to change the interface but continue to support old add-ins for backward compatibility. MAF also allows your application to load add-ins into a separate application domain so that they can crash harmlessly, without affecting the main application. All of these features mean that MAF works well if you have one development team working on an application and another one (or several) working on its add-ins. MAF is also particularly well suited for supporting third-party add-ins.

But MAF's features come at a cost. MAF is a complex framework, and setting up the add-in pipeline is tedious, even for a simple application. This is where MEF starts. It's a lighter-weight option that aims to make extensibility as easy as copying related assemblies into the same folder. But MEF also has a different underlying philosophy than MAF. Whereas MAF is a strict, interface-driven add-in model, MEF is a free-wheelin' system that allows an application to be built out of a collection of parts. Each part can export functionality, and any part can import the functionality of any other part. This system gives developers far more flexibility, and it works particularly well for designing *composable applications* (modular program that are developed by a single development team but need to be assembled in different ways, with differently implemented features, for separate releases). The obvious danger is that MEF is too loose, and a poorly designed application can quickly become a tangle of interrelated parts.

If you think MAF is the add-in system for you, keep reading—it's the technology that's discussed in this chapter. If you want to check out MEF, you can learn more at Microsoft's MEF community site at `http://www.codeplex.com/MEF`. And if your real interest is not in add-ins but in composable applications, you'll want to check out Microsoft's Composite Application Library (CAL), which is also known by its old code name, Prism. Although MEF is a general-purpose solution for building any sort of modular .NET application, CAL is tailored for WPF. It includes UI-oriented features, such as the ability to let different modules communicate with events and show content in separate display regions. CAL also has support for creating "hybrid" applications that can be compiled for the WPF platform or the browser-based Silverlight platform. You can find the documentations and downloads for CAL at `http://tinyurl.com/5ljve8`, and you can find an introductory article at `http://tinyurl.com/56m33n`.

---

■ **Note** From this point on, when the text refers to "the add-in model," it means the MAF add-in model.

---

# The Add-in Pipeline

The key advantage of the add-in model is that you don't need to write the underlying plumbing for tasks such as discovery. The key disadvantage is the add-in model's sheer complexity. The designers of .NET have taken great care to make the add-in model flexible enough to handle a wide range of versioning and hosting scenarios. The end result is that you must create at least seven (!) separate components to implement the add-in model in an application, even if you don't need to use its most sophisticated features.

The heart of the add-in model is the add-in *pipeline*, which is a chain of components that allow the hosting application to interact with an add-in (see Figure 32-1). At one end of the pipeline is the hosting application. At the other end is the add-in. In between are the five components that govern the interaction.

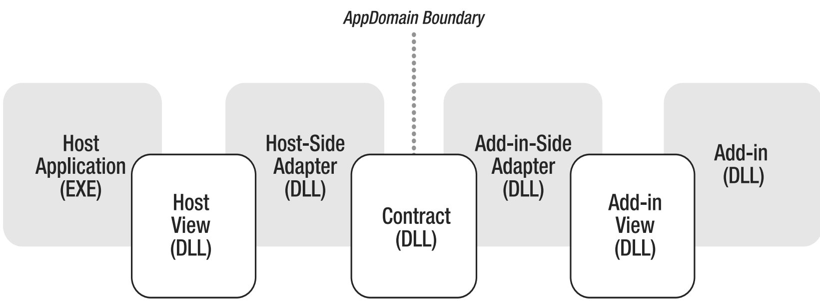


***Figure 32-1.*** *Communicating through the add-in pipeline*

At first glance, this model seems a bit excessive. A simpler scenario would put a single layer (the contract) between the application and the add-in. However, the additional layers (the views and adapters) allow the add-in model to be much more flexible in certain situations (as described in the sidebar "More Advanced Adapters").

## How the Pipeline Works

The *contract* is the cornerstone of the add-in pipeline. It includes one or more interfaces that define how the host application can interact with its add-ins and how the add-ins can interact with the host application. The contract assembly can also include custom serializable types that you plan to use to transmit data between the host application and the add-in.

The add-in pipeline is designed with extensibility and flexibility in mind. It's for this reason that the host application and the add-in don't directly use the contract. Instead, they use their own respective versions of the contract, called *views*. The host application uses the host view, while the add-in uses the add-in view. Typically, the view includes abstract classes that closely match the interfaces in the contract.

Although they're usually quite similar, the contracts and views are completely independent. It's up to the *adapters* to link these two pieces together. The adapters perform this linkage by providing classes that simultaneously inherit from the view classes and implement the contract interfaces. Figure 32-2 shows this design.

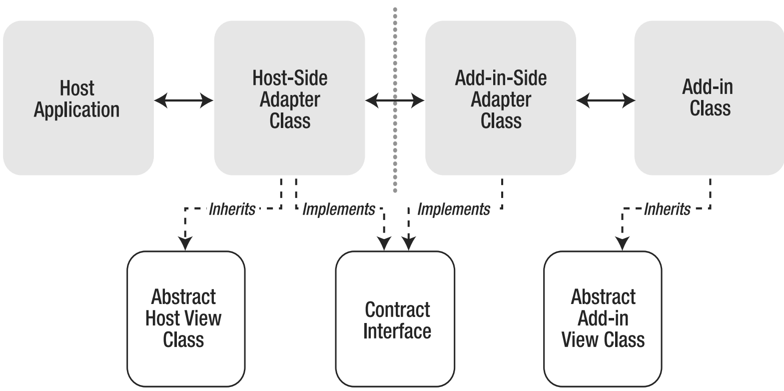


***Figure 32-2.****Class relationships in the pipeline*

Essentially, the adapters bridge the gap between the views and the contract interface. They map calls on a view to calls on the contract interface. They also map calls on the contract interface to the corresponding method on the view. This complicates the design somewhat but adds an all-important extra layer of flexibility.

To understand how the adapters work, consider what happens when an application uses an add-in. First, the host application calls one of the methods in the host view. But remember, the host view is an abstract class. Behind the scenes, the application is actually calling a method on the host adapter *through* the host view. (This is possible because the host adapter class derives from the host view class.) The host adapter then calls the corresponding method in the contract interface, which is implemented by the add-in adapter. Finally, the add-in adapter calls a method in the add-in view. This method is implemented by the add-in, which performs the actual work.

## MORE ADVANCED ADAPTERS

If you don't have any specialized versioning or hosting needs, the adapters are fairly straightforward. They simply pass the work along through the pipeline. However, the adapters are also an important extensibility point for more sophisticated scenarios. One example is versioning. Obviously, you can independently update an application or its add-ins without changing the way they interact, as long as you continue to use the same interfaces in the contract. However, in some cases you might need to change the interfaces to expose new features. This causes a bit of a problem, because the old interfaces must still be supported for backward compatibility with old add-ins. After a few revisions, you'll end up with a complex mess of similar yet different interfaces, and the application will need to recognize and support them all.

With the add-in model, you can take a different approach to backward compatibility. Instead of providing multiple interfaces, you can provide a single interface in your contract and use adapters to create different views. For example, a version 1 add-in can work with a version 2 application (which exposes a version 2 contract) as long as you have an add-in adapter that spans the gap. Similarly, if you develop an add-in that uses the version 2 contract, you can use it with the original version 1 application (and version 1 contract) by using a different add-in adapter.

It's possible to work similar magic if you have specialized hosting needs. For example, you can use adapters to load add-ins with different isolation levels or even share them between applications. The hosting application and the add-in don't need to be aware of these details, because the adapters handle all the details.

Even if you don't need to create custom adapters to implement specialized versioning and hosting strategies, you still need to include these components. However, all your add-ins can use the same view and adapter components. In other words, once you've gone to the trouble of setting up the complete pipeline for one add-in, you can add more add-ins without much work, as illustrated in Figure 32-3.

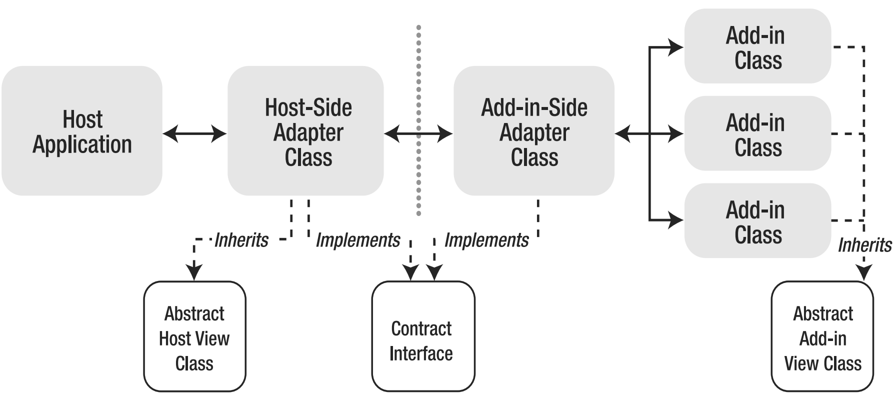


***Figure 32-3.*** *Multiple add-ins that use the same pipeline*

In the following sections, you'll learn how to implement the add-in pipeline for a WPF application.

## The Add-in Folder Structure

To use the add-in pipeline, you must follow a strict directory structure. This directory structure is separate from the application. In other words, it's perfectly acceptable to have your application residing at one location and all the add-ins and pipeline components residing at another location. However, the add-in components must be arranged in specifically named subdirectories with respect to one another. For example, if your add-in system uses the root directory c:\MyApp, you need the following subdirectories:

c:\MyApp\AddInSideAdapters

c:\MyApp\AddInViews

c:\MyApp\Contracts

c:\MyApp\HostSideAdapters

c:\MyApp\AddIns

Finally, the AddIns directory (shown last in this list) must have a separate subdirectory for each add-in your application is using, such as c:\MyApp\AddIns\MyFirstAddIn, c:\MyApp\AddIns\MySecondAddIn, and so on.

In this example, it's assumed that the application executable is deployed in the c:\MyApp subdirectory. In other words, the same directory does double duty as the application folder and as the add-in root. This is a common deployment choice, but it's certainly not a requirement.

---

■ **Note** If you've been paying close attention to the pipeline diagrams, you may have noticed that there's a subdirectory for each component except the host-side views. That's because the host views are used directly by the host application, so they're deployed alongside the application executable. (In this example, that means they are in c:\MyApp.) The add-in views aren't deployed in the same way, because it's likely that several add-ins will use the same add-in view. Thanks to the dedicated AddInViews folder, you need to deploy (and update) just one copy of each add-in view assembly.

---

## Preparing a Solution That Uses the Add-in Model

The add-in folder structure is mandatory. If you leave out one of the subdirectories listed in the previous section, you'll encounter a runtime exception when you search for add-ins.

Currently, Visual Studio doesn't have a template for creating applications that use add-ins. Thus, it's up to you to create these folders and set up your Visual Studio project to use them.

Here's the easiest approach to follow:

1. Create a top-level directory that will hold all the projects you're about to create. For example, you might name this directory c:\AddInTest.