The following example puts it all together. It maps the sys prefix to the System namespace and uses the System namespace to create three DateTime objects, which are used to fill a list:

```
<Window x:Class="WindowsApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    Width="300" Height="300"
    >
  <ListBox>
    <ListBoxItem>
      <sys:DateTime>10/13/2010 4:30 PM</sys:DateTime>
    </ListBoxItem>
    <ListBoxItem>
      <sys:DateTime>10/29/2010 12:30 PM</sys:DateTime>
    </ListBoxItem>
    <ListBoxItem>
      <sys:DateTime>10/30/2010 2:30 PM</sys:DateTime>
    </ListBoxItem>
  </ListBox>
</Window>
```

Loading and Compiling XAML

As you've already learned, XAML and WPF are separate, albeit complementary, technologies. As a result, it's quite possible to create a WPF application that doesn't use the faintest bit of XAML.

Altogether, there are three distinct coding styles that you can use to create a WPF application:

- **Code-only.** This is the traditional approach used in Visual Studio for Windows Forms applications. It generates a user interface through code statements.
- **Code and uncompiled markup (XAML).** This is a specialized approach that makes sense in certain scenarios where you need highly dynamic user interfaces. You load part of the user interface from a XAML file at runtime using the XamlReader class from the System.Windows.Markup namespace.
- **Code and compiled markup (BAML).** This is the preferred approach for WPF and the one that Visual Studio supports. You create a XAML template for each window, and this XAML is compiled into BAML and embedded in the final assembly. At runtime the compiled BAML is extracted and used to regenerate the user interface.

In the following sections, you'll dig deeper into these three models and how they actually work.

Code-Only

Code-only development is a less common (but still fully supported) avenue for writing a WPF application without any XAML. The obvious disadvantage to code-only development is that it has the potential to be extremely tedious. WPF controls don't include parameterized constructors, so even

adding a simple button to a window takes several lines of code. One potential advantage is that code-only development offers unlimited avenues for customization. For example, you could generate a form full of input controls based on the information in a database record, or you could conditionally decide to add or substitute controls depending on the current user. All you need is a sprinkling of conditional logic. By contrast, when you use XAML documents, they're embedded in your assembly as fixed, unchanging resources.

■ **Note** Even though you probably won't create a code-only WPF application, you probably will use the code-only approach to creating a WPF control at some point when you need an adaptable chunk of user interface.

The following code is for a modest window with a single button and an event handler (see Figure 2-3). When the window is created, the constructor calls an InitializeComponent() method that instantiates and configures the button and the form and hooks up the event handler.

■ **Note** To create this example, you must code the Window1 class from scratch (right-click the Solution Explorer, and choose Add ➤ Class to get started). You can't choose Add ➤ Window, because that will add a code file *and* a XAML template for your window, complete with an automatically generated InitializeComponent() method.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Markup;

public class Window1 : Window
{
    private Button button1;

    public Window1()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        // Configure the form.
        this.Width = this.Height = 285;
        this.Left = this.Top = 100;
        this.Title = "Code-Only Window";

        // Create a container to hold a button.
        DockPanel panel = new DockPanel();

        // Create the button.
```

```
        button1 = new Button();
        button1.Content = "Please click me.";
        button1.Margin = new Thickness(30);

        // Attach the event handler.
        button1.Click += button1_Click;

        // Place the button in the panel.
        IAddChild container = panel;
        container.AddChild(button1);

        // Place the panel in the form.
        container = this;
        container.AddChild(panel);
    }

    private void button1_Click(object sender, RoutedEventArgs e)
    {
        button1.Content = "Thank you.";
    }
}
```

Conceptually, the Window1 class in this example is a lot like a form in a traditional Windows Forms application. It derives from the base Window class and adds a private member variable for every control. For clarity, this class performs its initialization work in a dedicated InitializeComponent() method.

Figure 2-3. *A single-button window*

To get this application started, you can use a Main() method with code like this:

```
public class Program : Application
{
    [STAThread()]
    static void Main()
    {
        Program app = new Program();
        app.MainWindow = new Window1();
        app.MainWindow.ShowDialog();
    }
}
```

Code and Uncompiled XAML

One of the most interesting ways to use XAML is to parse it on the fly with the XamlReader. For example, imagine you start with this XAML content in a file named Window1.xaml:

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Button Name="button1" Margin="30">Please click me.</Button>
</DockPanel>
```

At runtime, you can load this content into a live window to create the same window shown in Figure 2-3. Here's the code that does it:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Markup;
using System.IO;

public class Window1 : Window
{
    private Button button1;

    public Window1()
    {
        InitializeComponent();
    }

    public Window1(string xamlFile)
    {
        // Configure the form.
        this.Width = this.Height = 285;
        this.Left = this.Top = 100;
        this.Title = "Dynamically Loaded XAML";

        // Get the XAML content from an external file.
        DependencyObject rootElement;
        using (FileStream fs = new FileStream(xamlFile, FileMode.Open))
        {
            rootElement = (DependencyObject)XamlReader.Load(fs);
```

```
        }

        // Insert the markup into this window.
        this.Content = rootElement;

        // Find the control with the appropriate name.
        button1 = (Button)LogicalTreeHelper.FindLogicalNode(rootElement, "button1");

        // Wire up the event handler.
        button1.Click += button1_Click;
    }

    private void button1_Click(object sender, RoutedEventArgs e)
    {
        button1.Content = "Thank you.";
    }
}
```

Here, the constructor receives the name of the XAML file as an argument (in this case, Window1.xaml). It then opens a FileStream and uses the XamlReader.Load() method to convert the content in this file into a DependencyObject, which is the base from which all WPF controls derive. The DependencyObject can be placed inside any type of container (for example, a Panel), but in this example it's used as the entire content inside the window.

■ **Note** In this example, you're loading an element—the DockPanel object—from the XAML file. Alternatively, you could load an entire XAML window. In this case, you would cast the object returned by XamlReader.Load() to the Window type and then call its Show() or ShowDialog() method to show it. The XAML 2009 example, which is featured later in this chapter, uses this technique.

To manipulate an element—for example, the button in the Window1.xaml file—you need to find the corresponding control object in the dynamically loaded content. The LogicalTreeHelper serves this purpose because it has the ability to search an entire tree of control objects, digging down as many layers as necessary until it finds the object with the name you've specified. An event handler is then attached to the Button.Click event.

Another alternative is to use the FrameworkElement.FindName() method. In this example, the root element is a DockPanel object. Like all the controls in a WPF window, DockPanel derives from FrameworkElement. That means you can replace this code:

```
button1 = (Button)LogicalTreeHelper.FindLogicalNode(rootElement, "button1");
```

with this equivalent approach:

```
FrameworkElement frameworkElement = (FrameworkElement)rootElement;
button1 = (Button)frameworkElement.FindName("button1");
```

In this example, the Window1.xaml file is distributed alongside the application executable, in the same folder. However, even though it isn't compiled as part of the application, you can still add it to your

Visual Studio project. Doing so allows you to manage the file more easily and design the user interface with Visual Studio (assuming you use the file extension .xaml so Visual Studio recognizes that it's a XAML document).

If you use this approach, just make sure that the loose XAML file isn't compiled or embedded in your project like a traditional XAML file. After you add it to your project, select it in the Solution Explorer, and use the Properties window to set the Build Action to None and the Copy to Output Directory to "Copy always."

Obviously, loading XAML dynamically won't be as efficient as compiling the XAML to BAML and then loading the BAML at runtime, particularly if your user interface is complex. However, it opens up a number of possibilities for building dynamic user interfaces. For example, you could create an all-purpose survey application that reads a form file from a web service and then displays the corresponding survey controls (labels, text boxes, check boxes, and so on). The form file would be an ordinary XML document with WPF tags, which you load into an existing form using the XamlReader. To collect the results once the survey is filled out, you simply need to enumerate over all the input controls and grab their content. Another advantage of the loose XAML approach is that it allows you to use the refinements in the XAML 2009 standard, as described later in this chapter.

Code and Compiled XAML

You've already seen the most common way to use XAML with the eight ball example shown in Figure 2-1 and dissected throughout this chapter. This is the method used by Visual Studio, and it has several advantages that this chapter has touched on already:

- Some of the plumbing is automatic. There's no need to perform ID lookup with the LogicalTreeHelper or wire up event handlers in code.
- Reading BAML at runtime is faster than reading XAML.
- Deployment is easier. Because BAML is embedded in your assembly as one or more resources, there's no way to lose it.
- XAML files can be edited in other programs, such as design tools. This opens up the possibility for better collaboration between programmers and designers. (You also get this benefit when using uncompiled XAML, as described in the previous section.)

Visual Studio uses a two-stage compilation process when you're compiling a WPF application. The first step is to compile the XAML files into BAML. For example, if your project includes a file name Window1.xaml, the compiler will create a temporary file named Window1.baml and place it in the obj\Debug subfolder (in your project folder). At the same time, a partial class is created for your window, using the language of your choice. For example, if you're using C#, the compiler will create a file named Window1.g.cs in the obj\Debug folder. The *g* stands for *generated*.

The partial class includes three things:

- Fields for all the controls in your window.
- Code that loads the BAML from the assembly, thereby creating the tree of objects. This happens when the constructor calls InitializeComponent().
- Code that assigns the appropriate control object to each field and connects all the event handlers. This happens in a method named Connect(), which the BAML parser calls every time it finds a named object.

The partial class does *not* include code to instantiate and initialize your controls because that task is performed by the WPF engine when the BAML is processed by the Application.LoadComponent() method.

■ **Note** As part of the XAML compilation process, the XAML compiler needs to create a partial class. This is possible only if the language you're using supports the .NET Code DOM model. C# and VB support Code DOM, but if you're using a third-party language, you'll need to make sure this support exists before you can create compiled XAML applications.

Here's the (slightly abbreviated) Window1.g.cs file from the eight ball example shown in Figure 2-1:

```
public partial class Window1 : System.Windows.Window,
  System.Windows.Markup.IComponentConnector
{
    // The control fields.
    internal System.Windows.Controls.TextBox txtQuestion;
    internal System.Windows.Controls.Button cmdAnswer;
    internal System.Windows.Controls.TextBox txtAnswer;

    private bool _contentLoaded;

    // Load the BAML.
    public void InitializeComponent()
    {
        if (_contentLoaded) {
            return;
        }
        _contentLoaded = true;

        System.Uri resourceLocater = new System.Uri("window1.baml",
          System.UriKind.RelativeOrAbsolute);
        System.Windows.Application.LoadComponent(this, resourceLocater);
    }

    // Hook up each control.
    void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
      object target)
    {
        switch (connectionId)
        {
            case 1:
                txtQuestion = ((System.Windows.Controls.TextBox)(target));
                return;
            case 2:
                cmdAnswer = ((System.Windows.Controls.Button)(target));
                cmdAnswer.Click += new System.Windows.RoutedEventHandler(
                  cmdAnswer_Click);
```

```
                return;
            case 3:
                txtAnswer = ((System.Windows.Controls.TextBox)(target));
                return;
        }
        this._contentLoaded = true;
    }
}
```

When the XAML-to-BAML compilation stage is finished, Visual Studio uses the appropriate language compiler to compile your code and the generated partial class files. In the case of a C# application, it's the csc.exe compiler that handles this task. The compiled code becomes a single assembly (EightBall.exe), and the BAML for each window is embedded as a separate resource.

XAML Only

The previous sections show you how to use XAML from a code-based application. As a .NET developer, this is what you'll spend most of your time doing. However, it's also possible to use a XAML file without creating any code. This is called a *loose* XAML file. Loose XAML files can be opened directly in Internet Explorer. (Assuming you've installed the .NET Framework 3.0 or are running Windows Vista or Windows 7, which has it preinstalled.)

■ **Note** If your XAML file uses code, it can't be opened in Internet Explorer. However, you can build a browser-based application called an XBAP that breaks through this boundary. Chapter 24 describes how.

At this point, it probably seems relatively useless to create a loose XAML file—after all, what's the point of a user interface with no code to drive it? However, as you explore XAML, you'll discover several features that are entirely declarative. These include features such as animation, triggers, data binding, and links (which can point to other loose XAML files). Using these features, you can build a few very simple no-code XAML files. They won't seem like complete applications, but they can accomplish quite a bit more than static HTML pages.

To try a loose XAML page, take a .xaml file and make these changes:

- Remove the Class attribute on the root element.
- Remove any attributes that attach event handlers (such as the Button.Click attribute).
- Change the name of the opening and closing tag from Window to Page. Internet Explorer can show only hosted pages, not stand-alone windows.

You can then double-click your .xaml file to load it in Internet Explorer. Figure 2-4 shows a converted EightBall.xaml page, which is included with the downloadable code for this chapter. You can type in the top text box, but because the application lacks the code-behind file, nothing happens when you click the button. If you want to create a more capable browser-based application that can include code, you'll need to use the XBAP model described in Chapter 24.

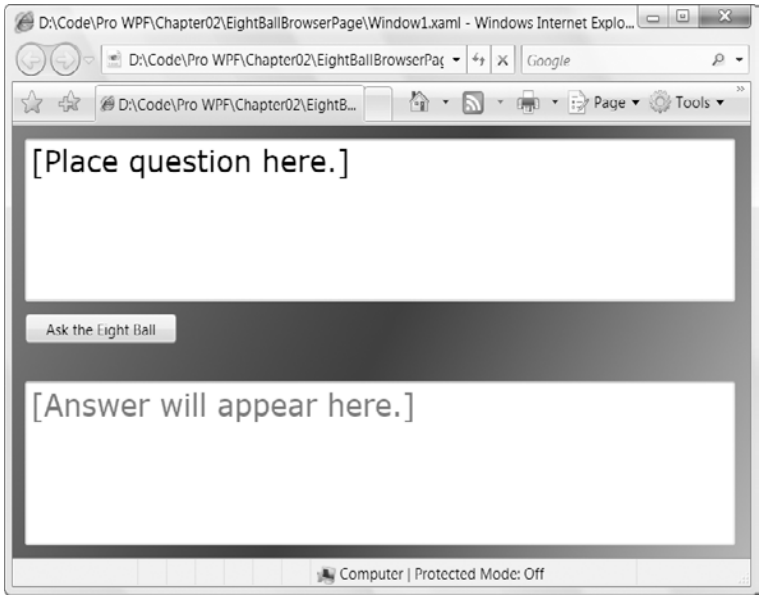

Figure 2-4. *A XAML page in a browser*

XAML 2009

As mentioned earlier in this chapter, WPF 4 introduces a new XAML standard, called XAML 2009. However, WPF doesn't adopt this standard wholeheartedly. If you want to use any of the XAML 2009 refinements today, you need to use loose, uncompiled XAML files, which won't suit the majority of developers.

Even if you decide not to use XAML 2009, it's worth quickly reviewing its new features. That's because XAML 2009 will eventually become the fully integrated, compiled standard in the next version of WPF. The following sections tour through its most important changes, all of which are demonstrated with the sample code for this chapter. Keep in mind that Visual Studio IntelliSense will flag some of these features as design-time mistakes, because it validates them using the original XAML standard. However, they'll work as expected at runtime.

Automatic Event Hookup

In the loose XAML example shown earlier, it was up to your code to manually connect event handlers, which means your code needs to have detailed knowledge of the XAML file content (such as the names of all the elements that raise the events you want to handle).

The XAML 2009 standard has a partial solution. Its parser can automatically connect event handlers, provided the corresponding event handler methods are defined in the root class.

For example, consider this markup:

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel>
    <Button Click="cmd_Click"></Button>
  </StackPanel>
</Window>
```

If you pass this document to the XamlReader.Load() method, an error will occur, because there is no Window.cmd_Click() method. But if you derive your own custom class from Window—say, Xaml2009Window—and you use markup like this:

```
<local:Xaml2009Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:local="clr-namespace:NonCompiledXaml;assembly=NonCompiledXaml">
  <StackPanel>
    <Button Click="cmd_Click"></Button>
  </StackPanel>
</local:Xaml2009Window>
```

the parser will be able to create an instance of Xaml2009Window class and will then attach the Button.Click event to the Xaml2009Window.cmd_Click() method automatically. This technique works perfectly well with private methods, but if the method doesn't exist (or if it doesn't have the right signature), an exception will occur.

Rather than loading the XAML in its constructor (as in the previous example), the Xaml2009Window class uses its own static method, named LoadWindowFromXaml(). This design is slightly preferred, because it emphasizes that a nontrivial *process* is required to create the window object—in this case, opening a file. This design also allows for clearer exception handling if the code can't find or access the XAML file. That's because it makes more sense for a method to throw an exception than for a constructor to throw one.

Here's the complete window code:

```
public class Xaml2009Window : Window
{
    public static Xaml2009Window LoadWindowFromXaml(string xamlFile)
    {
        // Get the XAML content from an external file.
        using (FileStream fs = new FileStream(xamlFile, FileMode.Open))
        {
            Xaml2009Window window = (Xaml2009Window)XamlReader.Load(fs);
            return window;
        }
    }

    private void cmd_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("You clicked.");
    }
}
```

You can create an instance of this window by calling the static LoadWindowFromXaml() method elsewhere in your code:

```
Program app = new Program();
app.MainWindow = Xaml2009Window.LoadWindowFromXaml("Xaml2009.xaml");
app.MainWindow.ShowDialog();
```

As you've probably already realized, this model is quite similar to the built-in Visual Studio model that compiles XAML. In both cases, all your event handling code is placed in a custom class that derives from the element you really want (typically, a Window or a Page).

References

In ordinary XAML, there's no easy way for one element to refer to another. The best solution is data binding (as you'll see in Chapter 8), but for simple scenarios it's overkill. XAML 2009 simplifies matters with a markup extension that's specifically designed for references.

The following markup snippet shows two references, which are used to set the Target property of two Label objects. The Label.Target property points to an input control that will receive the focus when the user hits the shortcut key. In this example, the first text box uses the shortcut key F (as signaled by the leading underscore character in the label text). As a result, the shortcut key is Alt+F. If the user presses this key combination, focus switches to the txtFirstName control that's defined just underneath.

```
<Label Target="{x:Reference txtFirstName}">_FirstName</Label>
<TextBox x:Name="txtFirstName"></TextBox>

<Label Target="{x:Reference txtLastName}">_LastName</Label>
<TextBox x:Name="txtLastName"></TextBox>
```

Built-in Types

As you've already learned, your XAML markup can access just about any type in any namespace, as long as you map it to an XML namespace first. Many WPF newcomers are surprised to discover that you need to use manual namespace mapping to use the basic data types of the System namespace, such as String, DateTime, TimeSpan, Boolean, Char, Decimal, Single, Double, Int32, Uri, Byte, and so on. Although it's a relatively minor barrier, it's an extra step and creates a bit of extra clutter:

```
<sys:String xmlns:sys="clr-namespace:System;assembly=mscorlib>A String</sys:String>
```

In XAML 2009, the XAML namespace provides direct access to these data types, with no extra effort required:

```
<x:String>A String</x:String>
```

You can also directly access the List and Dictionary generic collection types.

■ **Note** You won't run into this headache when setting the properties for WPF controls. That's because a value converter will take your string and convert it into the appropriate data type automatically, as explained earlier in this chapter. However, there are some situations where value converters aren't at work and you *do* need specific data types. One example is if you want to use simple data types to store *resources*—objects that can be reused throughout your markup and code. You'll learn to use resources in Chapter 10.

Advanced Object Creation

Ordinary XAML can create just about any type—provided it has a simple no-argument constructor. XAML 2009 removes this limitation and gives you two more powerful ways to create an initialize objects.

First, you can use the <x:Arguments> element to supply constructor arguments. For example, imagine you have a class like this, with no zero-argument constructor:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

You can instantiate it in XAML 2009 like this:

```
<local:Person>
  <x:Arguments>
    <x:String>Joe</x:String>
    <x:String>McDowell</x:String>
  </x:Arguments>
</local:Person>
```

The second approach you can use is to rely on a static method (either in the same class or in another class) that creates a live instance of the object you want. This pattern is called the *factory method*. One example of the factory method is the Guid class in the System namespace, which represents a globally unique identifier. You can't create a Guid object with the new keyword, but you can call the Guid.NewGuid() method, which returns a new instance:

```
Guid myGuid = Guid.NewGuid();
```

In XAML 2009, you can use the same technique through markup. The trick is the x:FactoryMethod attribute. Here's how you can create a Guid in markup, assuming you've mapped the sys namespace prefix to the System namespace:

```
<sys:Guid x:FactoryMethod="Guid.NewGuid"></sys:Guid>
```

XAML 2009 also allows you to instantiate generic collections, which isn't possible in ordinary XAML. (One common workaround is to derive a custom collection class to use as a wrapper and instantiate that in XAML. However, this quickly litters your code with unnecessary one-off classes.) In XAML 2009, the TypeArguments attribute gives you a way to pass type arguments to the generic class.

For example, imagine you want to create a list of Person objects, which you can accomplish with code like this:

```
List<Person> people = new List<Person>();
people.Add(new Person("Joe", "McDowell");
```

In XAML 2009, this markup achieves the same result:

```
<x:List x:TypeArguments="Person">
  <local:Person>
    <x:Arguments>
      <x:String>Joe</x:String>
      <x:String>McDowell</x:String>
    </x:Arguments>
  </local:Person>
</x:List>
```

or, assuming the Person class has a default no-argument constructor, like this:

```
<x:List x:TypeArguments="Person">
  <local:Person FirstName="Joe" LastName="McDowell" />
</x:List>
```

The Last Word

In this chapter, you took a tour through a simple XAML file and learned its syntax at the same time. Here's what you saw:

- You considered key XAML ingredients, such as type converters, markup extensions, and attached properties.
- You learned how to wire up a code-behind class that can handle the events raised by your controls.
- You considered the compilation process that takes a standard WPF application into a compiled executable file. At the same time, you took a look at three variants: creating a WPF application through code alone, creating a WPF page with nothing but XAML, and loading XAML manually at runtime.
- You took a quick look at the changes that are introduced in XAML 2009.

Although you haven't had an exhaustive look at every detail of XAML markup, you've learned enough to reap all its benefits. Now, your attention can shift to the WPF technology itself, which holds some of the most interesting surprises. In the next chapter, you'll consider how controls are organized into realistic windows using the WPF layout panels.

CHAPTER 3

Layout

Half the battle in any user interface design is organizing the content in a way that's attractive, practical, and flexible. But the real challenge is making sure that your layout can adapt itself gracefully to different window sizes.

In WPF, you shape layout using different *containers*. Each container has its own layout logic—some stack elements, others arrange them in a grid of invisible cells, and so on. If you've programmed with Windows Forms, you'll be surprised to find that coordinate-based layout is strongly discouraged in WPF. Instead, the emphasis is on creating more flexible layouts that can adapt to changing content, different languages, and a variety of window sizes. For most developers moving to WPF, the new layout system is a great surprise—and the first real challenge.

In this chapter, you'll see how the WPF layout model works, and you'll begin using the basic layout containers. You'll also consider several common layout examples—everything from a basic dialog box to a resizable split window—in order to learn the fundamentals of WPF layout.

What's New WPF 4 still uses the same flexible layout system, but it adds one minor frill that can save some serious headaches. That feature is *layout rounding*, and it ensures that layout containers don't attempt to put content in fractional-pixel positions, which can blur shapes and images. To learn more, see the "Layout Rounding" section in this chapter.

Understanding Layout in WPF

The WPF layout model represents a dramatic shift in the way Windows developers approach user interfaces. In order to understand the new WPF layout model, it helps to take a look at what's come before.

In .NET 1.*x*, Windows Forms provided a fairly primitive layout system. Controls were fixed in place using hard-coded coordinates. The only saving grace was *anchoring* and *docking*, two features that allowed controls to move or resize themselves along with their container. Anchoring and docking were great for creating simple resizable windows—for example, keeping an OK and Cancel button stuck to the bottom-right corner of a window, or allowing a TreeView to expand to fill an entire form—but they couldn't handle serious layout challenges. For example, anchoring and docking couldn't implement bi-pane proportional resizing (dividing extra space equally among two regions). They also weren't much help if you had highly dynamic content, such as a label that might expand to hold more text than anticipated, causing it to overlap other nearby controls.

In .NET 2.0, Windows Forms filled the gaps with two new layout containers: the FlowLayoutPanel and TableLayoutPanel. Using these controls, you could create more sophisticated web-like interfaces.

Both layout containers allowed their contained controls to grow and bump other controls out of the way. This made it easier to deal with dynamic content, create modular interfaces, and localize your application. However, the layout panels still felt like an add-on to the core Windows Forms layout system, which used fixed coordinates. The layout panels were an elegant solution, but you could see the duct tape holding it all together.

WPF introduces a new layout system that's heavily influenced by the developments in Windows Forms. This system reverses the .NET 2.0 model (coordinate-based layout with optional flow-based layout panels) by making flow-based layout the standard and giving only rudimentary support for coordinate-based layout. The benefits of this shift are enormous. Developers can now create resolution-independent, size-independent interfaces that scale well on different monitors, adjust themselves when content changes, and handle the transition to other languages effortlessly. However, before you can take advantage of these changes, you'll need to start thinking about layout a little differently.

The WPF Layout Philosophy

A WPF window can hold only a single element. To fit in more than one element and create a more practical user interface, you need to place a container in your window and then add other elements to that container.

■ **Note** This limitation stems from the fact that the Window class is derived from ContentControl, which you'll study more closely in Chapter 6.

In WPF, layout is determined by the container that you use. Although there are several containers to choose from, the "ideal" WPF window follows a few key principles:

- **Elements (such as controls) should not be explicitly sized.** Instead, they grow to fit their content. For example, a button expands as you add more text. You can limit controls to acceptable sizes by setting a maximum and minimum size.
- **Elements do not indicate their position with screen coordinates.** Instead, they are arranged by their container based on their size, order, and (optionally) other information that's specific to the layout container. If you need to add whitespace between elements, you use the Margin property.

■ **Tip** Hard-coded sizes and positions are evil because they limit your ability to localize your interface, and they make it much more difficult to deal with dynamic content.

- **Layout containers "share" the available space among their children.** They attempt to give each element its preferred size (based on its content) if the space is available. They can also distribute extra space to one or more children.
- **Layout containers can be nested.** A typical user interface begins with the Grid, WPF's most capable container, and contains other layout containers that arrange smaller groups of elements, such as captioned text boxes, items in a list, icons on a toolbar, a column of buttons, and so on.

Although there are exceptions to these rules, they reflect the overall design goals of WPF. In other words, if you follow these guidelines when you build a WPF application, you'll create a better, more flexible user interface. If you break these rules, you'll end up with a user interface that isn't well suited to WPF and is much more difficult to maintain.

The Layout Process

WPF layout takes place in two stages: a *measure* stage and an *arrange* stage. In the measure stage, the container loops through its child elements and asks them to provide their preferred size. In the arrange stage, the container places the child elements in the appropriate position.

Of course, an element can't always get its preferred size—sometimes the container isn't large enough to accommodate it. In this case, the container must truncate the offending element to fit the visible area. As you'll see, you can often avoid this situation by setting a minimum window size.

■ **Note** Layout containers don't provide any scrolling support. Instead, scrolling is provided by a specialized content control—the ScrollViewer—that can be used just about anywhere. You'll learn about the ScrollViewer in Chapter 6.

The Layout Containers

All the WPF layout containers are panels that derive from the abstract System.Windows.Controls.Panel class (see Figure 3-1). The Panel class adds a small set of members, including the three public properties that are detailed in Table 3-1.

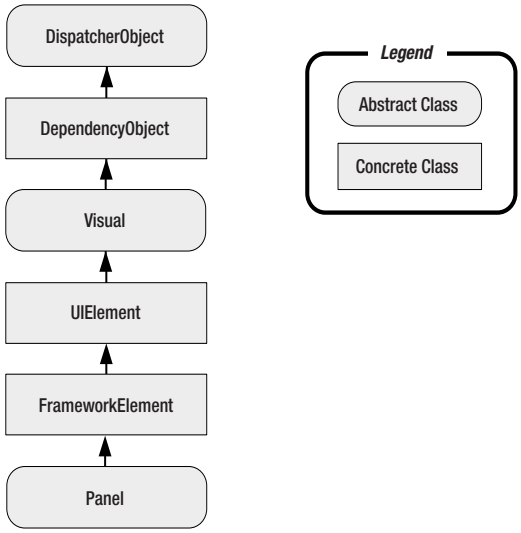

Figure 3-1. *The hierarchy of the Panel class*

Table 3-1. *Public Properties of the Panel Class*

Name	Description
Background	The brush that's used to paint the panel background. You must set this property to a non-null value if you want to receive mouse events. (If you want to receive mouse events but you don't want to display a solid background, just set the background color to Transparent.) You'll learn more about basic brushes in Chapter 6 (and more advanced brushes in Chapter 12).
Children	The collection of items that's stored in the panel. This is the first level of items—in other words, these items may themselves contain more items.
IsItemsHost	A Boolean value that's true if the panel is being used to show the items that are associated with an ItemsControl (such as the nodes in a TreeView or the list entries in a ListBox). Most of the time you won't even be aware that a list control is using a behind-the-scenes panel to manage the layout of its items. However, this detail becomes more important if you want to create a customized list that lays out children in a different way (for example, a ListBox that tiles images). You'll use this technique in Chapter 20.

■ **Note** The Panel class also has a bit of internal plumbing you can use if you want to create your own layout container. Most notably, you can override the MeasureOverride() and ArrangeOverride() methods inherited from FrameworkElement to change the way the panel handles the measure stage and the arrange stage when organizing its child elements. You'll learn how to create a custom panel in Chapter 18.

On its own, the base Panel class is nothing but a starting point for other more specialized classes. WPF provides a number of Panel-derived classes that you can use to arrange layout. The most fundamental of these are listed in Table 3-2. As with all WPF controls and most visual elements, these classes are found in the System.Windows.Controls namespace.

Table 3-2. *Core Layout Panels*

Name	Description
StackPanel	Places elements in a horizontal or vertical stack. This layout container is typically used for small sections of a larger, more complex window.
WrapPanel	Places elements in a series of wrapped lines. In horizontal orientation, the WrapPanel lays items out in a row from left to right and then onto subsequent lines. In vertical orientation, the WrapPanel lays out items in a top-to-bottom column and then uses additional columns to fit the remaining items.

Name	Description
DockPanel	Aligns elements against an entire edge of the container.
Grid	Arranges elements in rows and columns according to an invisible table. This is one of the most flexible and commonly used layout containers.
UniformGrid	Places elements in an invisible table but forces all cells to have the same size. This layout container is used infrequently.
Canvas	Allows elements to be positioned absolutely using fixed coordinates. This layout container is the most similar to traditional Windows Forms, but it doesn't provide anchoring or docking features. As a result, it's an unsuitable choice for a resizable window unless you're willing to do a fair bit of work.

Along with these core containers, you'll encounter several more specialized panels in various controls. These include panels that are dedicated to holding the child items of a particular control, such as TabPanel (the tabs in a TabControl), ToolbarPanel (the buttons in a Toolbar), and ToolbarOverflowPanel (the commands in a Toolbar's overflow menu). There's also a VirtualizingStackPanel, which data-bound list controls use to dramatically reduce their overhead, and an InkCanvas, which is similar to the Canvas but has support for handling stylus input on the TabletPC. (For example, depending on the mode you choose, the InkCanvas supports drawing with the pointer to select onscreen elements. And although it's a little counterintuitive, you can use the InkCanvas with an ordinary computer and a mouse.). You'll learn about the InkCanvas in this chapter and you'll take a closer look at the VirtualizingStackPanel in Chapter 19. You'll learn about the other specialized panels when you consider the related control, elsewhere in this book.

Simple Layout with the StackPanel

The StackPanel is one of the simplest layout containers. It simply stacks its children in a single row or column.

For example, consider this window, which contains a stack of four buttons:

```
<Window x:Class="Layout.SimpleStack"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Layout" Height="223" Width="354"
    >
  <StackPanel>
    <Label>A Button Stack</Label>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </StackPanel>
</Window>
```

Figure 3-2 shows the window that results.

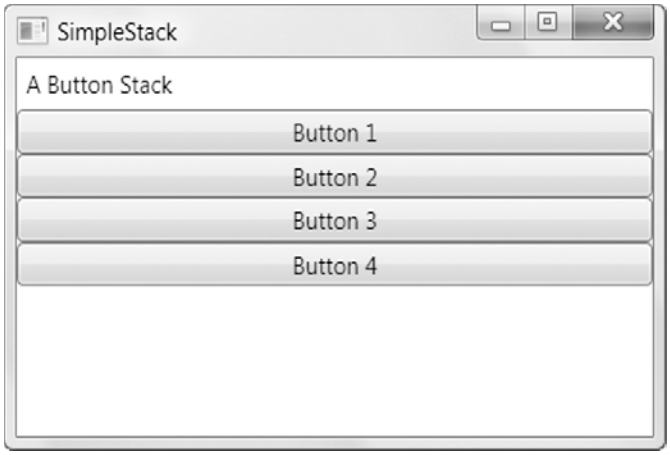

Figure 3-2. *The StackPanel in action*

Adding a Layout Container in Visual Studio

It's relatively easy to create this example using the designer in Visual Studio. Begin by deleting the root Grid element (if it's there). Then, drag a StackPanel into the window. Next, drag the other elements (the label and four buttons) into the window, in the top-to-bottom order you want. If you want to rearrange the order of elements in the StackPanel, you can simply drag any one to a new position.

You need to consider a few quirks when you create a user interface with Visual Studio. When you drag elements from the Toolbox to a window, Visual Studio adds certain details to your markup. First, Visual Studio automatically assigns a name to every new control (which is harmless but unnecessary). It also adds hard-coded Width and Height values, which is much more limiting.

As discussed earlier, explicit sizes limit the flexibility of your user interface. In many cases, it's better to let controls size themselves to fit their content or size themselves to fit their container. In the current example, fixed sizes are a reasonable approach to give the buttons a consistent width. However, a better approach would be to let the largest button size itself to fit its content and have all smaller buttons stretch themselves to match. (This design, which requires the use of a Grid, is described later in this chapter.) And no matter what approach you use with the button, you almost certainly want to remove the hard-coded Width and Height values for the StackPanel, so it can grow or shrink to fit the available space in the window.

By default, a StackPanel arranges elements from top to bottom, making each one as tall as is necessary to display its content. In this example, that means the labels and buttons are sized just large enough to comfortably accommodate the text inside. All elements are stretched to the full width of the StackPanel, which is the width of the window. If you widen the window, the StackPanel widens as well, and the buttons stretch themselves to fit.

The StackPanel can also be used to arrange elements horizontally by setting the Orientation property:

```
<StackPanel Orientation="Horizontal">
```

Now elements are given their minimum width (wide enough to fit their text) and are stretched to the full height of the containing panel. Depending on the current size of the window, this may result in some elements that don't fit, as shown in Figure 3-3.

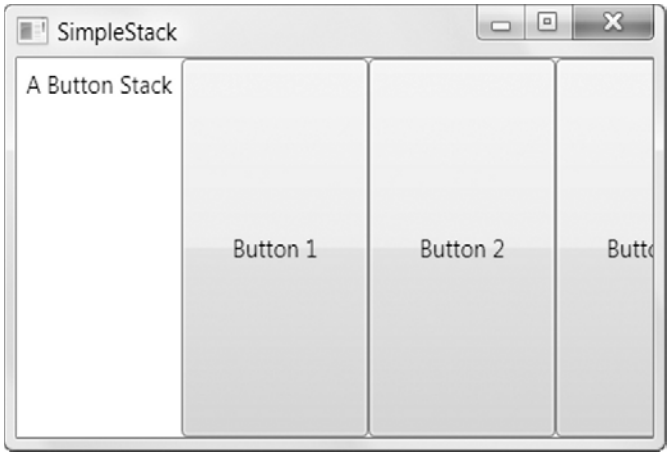

Figure 3-3. *The StackPanel with horizontal orientation*

Clearly, this doesn't provide the flexibility real applications need. Fortunately, you can fine-tune the way the StackPanel and other layout containers work using layout properties, as described next.

Layout Properties

Although layout is determined by the container, the child elements can still get their say. In fact, layout panels work in concert with their children by respecting a small set of layout properties, as listed in Table 3-3.

Table 3-3. *Layout Properties*

Name	Description
HorizontalAlignment	Determines how a child is positioned inside a layout container when there's extra horizontal space available. You can choose Center, Left, Right, or Stretch.
VerticalAlignment	Determines how a child is positioned inside a layout container when there's extra vertical space available. You can choose Center, Top, Bottom, or Stretch.