This document includes only two elements—the top-level Window element, which represents the entire window, and the Grid, in which you can place all your controls. Although you could use any top-level element, WPF applications rely on just a few:

- Window
- Page (which is similar to Window but used for navigable applications)
- Application (which defines application resources and startup settings)

As in all XML documents, there can only be one top-level element. In the previous example, that means that as soon as you close the Window element with the </Window> tag, you end the document. No more content can follow.

Looking at the start tag for the Window element, you'll find several interesting attributes, including a class name and two XML namespaces (described in the following sections). You'll also find the three properties shown here:

```
Title="Window1" Height="300" Width="300">
```

Each attribute corresponds to a separate property of the Window class. All in all, this tells WPF to create a window with the caption Window1 and to make it 300 by 300 units large.

■ **Note** As you learned in Chapter 1, WPF uses a relative measurement system that isn't what most Windows developers expect. Rather than letting you set sizes using physical pixels, WPF uses *device-independent units* that can scale to fit different monitor resolutions and are defined as 1/96 of an inch. That means the 300-by-300-unit window in the previous example will be rendered as a 300-by-300-*pixel* window if your system DPI setting is the standard 96 dpi. However, on a system with a higher system DPI, more pixels will be used. Chapter 1 has the full story.

XAML Namespaces

Clearly, it's not enough to supply just a class name. The XAML parser also needs to know the .NET namespace where this class is located. For example, the Window class could exist in several places—it might refer to the System.Windows.Window class, or it could refer to a Window class in a third-party component or one you've defined in your application. To figure out which class you really want, the XAML parser examines the XML namespace that's applied to the element.

Here's how it works. In the sample document shown earlier, two namespaces are defined:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

■ **Note** XML namespaces are declared using attributes. These attributes can be placed inside any element start tag. However, convention dictates that all the namespaces you need to use in a document should be declared in the very first tag, as they are in this example. Once a namespace is declared, it can be used anywhere in the document.

The xmlns attribute is a specialized attribute in the world of XML that's reserved for declaring namespaces. This snippet of markup declares two namespaces that you'll find in every WPF XAML document you create:

- `http://schemas.microsoft.com/winfx/2006/xaml/presentation` is the core WPF namespace. It encompasses all the WPF classes, including the controls you use to build user interfaces. In this example, this namespace is declared without a namespace prefix, so it becomes the default namespace for the entire document. In other words, every element is automatically placed in this namespace unless you specify otherwise.
- `http://schemas.microsoft.com/winfx/2006/xaml` is the XAML namespace. It includes various XAML utility features that allow you to influence how your document is interpreted. This namespace is mapped to the prefix *x*. That means you can apply it by placing the namespace prefix before the element name (as in <x:ElementName>).

As you can see, the XML namespace name doesn't match any particular .NET namespace. There are a couple of reasons the creators of XAML chose this design. By convention, XML namespaces are often URIs (as they are here). These URIs look like they point to a location on the Web, but they don't. The URI format is used because it makes it unlikely that different organizations will inadvertently create different XML-based languages with the same namespace. Because the domain `schemas.microsoft.com` is owned by Microsoft, only Microsoft will use it in an XML namespace name.

The other reason that there isn't a one-to-one mapping between the XML namespaces used in XAML and .NET namespaces is because it would significantly complicate your XAML documents. The problem here is that WPF encompasses well over a dozen namespaces (all of which start with System.Windows). If each .NET namespace had a different XML namespace, you'd need to specify the right namespace for each and every control you use, which quickly gets messy. Instead, the creators of WPF chose to combine all of these .NET namespaces into a single XML namespace. This works because within the different .NET namespaces that are part of WPF, there aren't any classes that have the same name.

The namespace information allows the XAML parser to find the right class. For example, when it looks at the Window and Grid elements, it sees that they are placed in the default WPF namespace. It then searches the corresponding .NET namespaces until it finds System.Windows.Window and System.Windows.Controls.Grid.

The Code-Behind Class

XAML allows you to construct a user interface, but in order to make a functioning application you need a way to connect the event handlers that contain your application code. XAML makes this easy using the Class attribute that's shown here:

```
<Window x:Class="WindowsApplication1.Window1"
```

The x namespace prefix places the Class attribute in the XAML namespace, which means this is a more general part of the XAML language. In fact, the Class attribute tells the XAML parser to generate a new class with the specified name. That class derives from the class that's named by the XML element. In other words, this example creates a new class named Window1, which derives from the base Window class.

The Window1 class is generated automatically at compile time. But here's where things get interesting. You can supply a piece of the Window1 class that will be merged into the automatically generated portion. The piece you specify is the perfect container for your event handling code.

■ **Note** This magic happens through the C# feature known as *partial classes*. Partial classes allow you to split a class into two or more separate pieces for development and fuse them together in the compiled assembly. Partial classes can be used in a variety of code management scenarios, but they're most useful in situations like these, where your code needs to be merged with a designer-generated file.

Visual Studio helps you out by automatically creating a partial class where you can place your event handling code. For example, if you create an application named WindowsApplication1, which contains a window named Window1 (as in the previous example), Visual Studio will start you out with this basic skeleton of a class:

```
namespace WindowsApplication1
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

When you compile your application, the XAML that defines your user interface (such as Window1.xaml) is translated into a CLR type declaration that is merged with the logic in your code-behind class file (such as Window1.xaml.cs) to form one single unit.

The InitializeComponent() Method

Currently, the Window1 class code doesn't include any real functionality. However, it does include one important detail—the default constructor, which calls InitializeComponent() when you create an instance of the class.

■ **Note** The InitializeComponent()method plays a key role in WPF applications. For that reason, you should never delete the InitializeComponent() call in your window's constructor. Similarly, if you add another constructor to your window class, make sure it also calls InitializeComponent().

The InitializeComponent() method isn't visible in your source code because it's automatically generated when you compile your application. Essentially, all InitializeComponent() does is call the LoadComponent() method of the System.Windows.Application class. The LoadComponent() method extracts the BAML (the compiled XAML) from your assembly and uses it to build your user interface. As it parses the BAML, it creates each control object, sets its properties, and attaches any event handlers.

■ **Note** If you can't stand the suspense, jump ahead to the end of the chapter. You'll see the code for the automatically generated InitializeComponent() method in the section "Code and Compiled XAML."

Naming Elements

There's one more detail to consider. In your code-behind class, you'll often want to manipulate controls programmatically. For example, you might want to read or change properties or attach and detach event handlers on the fly. To make this possible, the control must include a XAML Name attribute. In the previous example, the Grid control does not include a Name attribute, so you won't be able to manipulate it in your code-behind file.

Here's how you can attach a name to the Grid:

```
    <Grid x:Name="grid1">
    </Grid>
```

You can make this change by hand in the XAML document, or you can select the grid in the Visual Studio designer and set the Name property using the Properties window.

Either way, the Name attribute tells the XAML parser to add a field like this to the automatically generated portion of the Window1 class:

```
private System.Windows.Controls.Grid grid1;
```

Now you can interact with the grid in your Window1 class code by using the name grid1:

```
MessageBox.Show(String.Format("The grid is {0}x{1} units in size.",
  grid1.ActualWidth, grid1.ActualHeight));
```

This technique doesn't add much for the simple grid example, but it becomes much more important when you need to read values in input controls such as text boxes and list boxes.

The Name property shown previously is part of the XAML language, and it's used to help integrate your code-behind class. Somewhat confusingly, many classes define their own Name property. (One example is the base FrameworkElement class from which all WPF elements derive.) XAML parsers have a clever way of handling this. You can set either the XAML Name property (using the x: prefix) or the Name property that belongs to the actual element (by leaving out the prefix). Either way, the result is the same—the name you specify is used in the automatically generated code file *and* it's used to set the Name property.

That means the following markup is equivalent to what you've already seen:

```
<Grid Name="grid1">
</Grid>
```

This bit of magic works only if the class that includes the Name property decorates itself with the RuntimeNameProperty attribute. The RuntimeNameProperty indicates which property should be treated as the name for instances of that type. (Obviously, it's usually the property that's named Name.) The FrameworkElement class includes the RuntimeNameProperty attribute, so there's no problem.

■ **Tip** In a traditional Windows Forms application, every control has a name. In a WPF application, there's no such requirement. However, if you create a window by dragging and dropping elements onto the Visual Studio design surface, each element will be given an automatically generated name. This is simply a convenience. If you don't want to interact with an element in your code, you're free to remove its Name attribute from the markup. The examples in this book usually omit element names when they aren't needed, which makes the markup more concise.

By now, you should have a basic understanding of how to interpret a XAML document that defines a window and how that XAML document is converted into a final compiled class (with the addition of any code you've written). In the next section, you'll look at the property syntax in more detail and learn to wire up event handlers.

Properties and Events in XAML

So far, you've considered a relatively unexciting example—a blank window that hosts an empty Grid control. Before going any further, it's worth introducing a more realistic window that includes several controls. Figure 2-1 shows an example with an automatic question answerer.

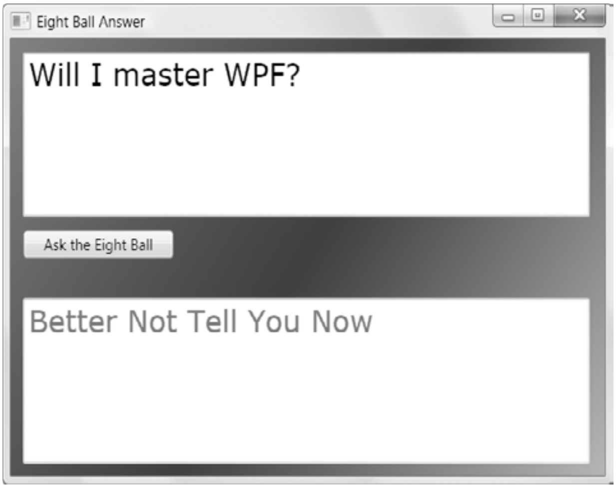

Figure 2-1. Ask the eight ball, and all will be revealed.

The eight ball window includes four controls: a Grid (the most common tool for arranging layout in WPF), two TextBox objects, and a Button. The markup that's required to arrange and configure these controls is significantly longer than the previous examples. Here's an abbreviated listing that replaces some of the details with an ellipsis (...) to expose the overall structure:

```
<Window x:Class="EightBall.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eight Ball Answer" Height="328" Width="412">
  <Grid Name="grid1">
    <Grid.Background>
      ...
    </Grid.Background>
    <Grid.RowDefinitions>
      ...
    </Grid.RowDefinitions>

    <TextBox Name="txtQuestion" ... >
      ...
    </TextBox>

    <Button Name="cmdAnswer" ... >
      ...
    </Button>

    <TextBox Name="txtAnswer" ... >
      ...
    </TextBox>
  </Grid>
</Window>
```

In the following sections, you'll explore the parts of this document—and learn the syntax of XAML along the way.

■ **Note** XAML isn't limited to the classes that are part of WPF. You can use XAML to create an instance of any class that meets a few ground rules. You'll learn how to use your own classes with XAML later in this chapter.

Simple Properties and Type Converters

As you've already seen, the attributes of an element set the properties of the corresponding object. For example, the text boxes in the eight ball example configure the alignment, margin, and font:

```
<TextBox Name="txtQuestion"
  VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
  FontFamily="Verdana" FontSize="24" Foreground="Green" ... >
```

For this to work, the System.Windows.Controls.TextBox class must provide the following properties: VerticalAlignment, HorizontalAlignment, FontFamily, FontSize, and Foreground. You'll learn the specific meaning for each of these properties in the following chapters.

To make this system work, the XAML parser needs to perform a bit more work than you might initially realize. The value in an XML attribute is always a plain-text string. However, object properties can be any .NET type. In the previous example, there are two properties that use enumerations (VerticalAlignment and HorizontalAlignment), one string (FontFamily), one integer (FontSize), and one Brush object (Foreground).

To bridge the gap between string values and nonstring properties, the XAML parser needs to perform a conversion. The conversion is performed by *type converters*, a basic piece of .NET infrastructure that's existed since .NET 1.0.

Essentially, a type converted has one role in life—it provides utility methods that can convert a specific .NET data type to and from any other .NET type, such as a string representation in this case. The XAML parser follows two steps to find a type converter:

1. It examines the property declaration, looking for a TypeConverter attribute. (If present, the TypeConverter attribute indicates what class can perform the conversion.) For example, when you use a property such as Foreground, .NET checks the declaration of the Foreground property.

2. If there's no TypeConverter attribute on the property declaration, the XAML parser checks the class declaration of the corresponding data type. For example, the Foreground property uses a Brush object. The Brush class (and its derivatives) use the BrushConverter because the Brush class is decorated with the TypeConverter(typeof(BrushConverter)) attribute declaration.

If there's no associated type converter on the property declaration or the class declaration, the XAML parser generates an error.

This system is simple but flexible. If you set a type converter at the class level, that converter applies to every property that uses that class. On the other hand, if you want to fine-tune the way type conversion works for a particular property, you can use the TypeConverter attribute on the property declaration instead.

It's technically possible to use type converters in code, but the syntax is a bit convoluted. It's almost always better to set a property directly—not only is it faster, but it also avoids potential errors from mistyping strings, which won't be caught until runtime. (This problem doesn't affect XAML, because the XAML is parsed and validated at compile time.) Of course, before you can set the properties on a WPF element, you need to know a bit more about the basic WPF properties and data types—a job you'll tackle in the next few chapters.

■ **Note** XAML, like all XML-based languages, is *case-sensitive*. That means you can't substitute <button> for <Button>. However, type converters usually aren't case-sensitive, which means both Foreground="White" and Foreground="white" have the same result.

Complex Properties

As handy as type converters are, they aren't practical for all scenarios. For example, some properties are full-fledged objects with their own set of properties. Although it's possible to create a string representation that the type converter could use, that syntax might be difficult to use and prone to error.

Fortunately, XAML provides another option: *property-element syntax*. With property-element syntax, you add a child element with a name in the form Parent.PropertyName. For example, the Grid has a Background property that allows you to supply a brush that's used to paint the area behind the controls. If you want to use a complex brush—one more advanced than a solid color fill—you'll need to add a child tag named Grid.Background, as shown here:

```
<Grid Name="grid1">
  <Grid.Background>
    ...
  </Grid.Background>
  ...
</Grid>
```

The key detail that makes this work is the period (.) in the element name. This distinguishes properties from other types of nested content.

This still leaves one detail—namely, once you've identified the complex property you want to configure, how do you set it? Here's the trick. Inside the nested element, you can add another tag to instantiate a specific class. In the eight ball example (shown in Figure 2-1), the background is filled with a gradient. To define the gradient you want, you need to create a LinearGradientBrush object.

Using the rules of XAML, you can create the LinearGradientBrush object using an element with the name LinearGradientBrush:

```
<Grid Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
    </LinearGradientBrush>
  </Grid.Background>
  ...
</Grid>
```

The LinearGradientBrush is part of the WPF set of namespaces, so you can keep using the default XML namespace for your tags.

However, it's not enough to simply create the LinearGradientBrush—you also need to specify the colors in that gradient. You do this by filling the LinearGradientBrush.GradientStops property with a collection of GradientStop objects. Once again, the GradientStops property is too complex to be set with an attribute value alone. Instead, you need to rely on the property-element syntax:

```
<Grid Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
      <LinearGradientBrush.GradientStops>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Grid.Background>
  ...
</Grid>
```

Finally, you can fill the GradientStops collection with a series of GradientStop objects. Each GradientStop object has an Offset and Color property. You can supply these two values using the ordinary property-attribute syntax:

```
<Grid Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.00" Color="Red" />
        <GradientStop Offset="0.50" Color="Indigo" />
        <GradientStop Offset="1.00" Color="Violet" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Grid.Background>
  ...
</Grid>
```

■ **Note** You can use property-element syntax for any property. But usually you'll use the simpler property-attribute approach if the property has a suitable type converter. Doing so results in more compact code.

Any set of XAML tags can be replaced with a set of code statements that performs the same task. The tags shown previously, which fill the background with a gradient of your choice, are equivalent to the following code:

```
LinearGradientBrush brush = new LinearGradientBrush();

GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Red;
brush.GradientStops.Add(gradientStop1);
```

```
GradientStop gradientStop2 = new GradientStop();
gradientStop2.Offset = 0.5;
gradientStop2.Color = Colors.Indigo;
brush.GradientStops.Add(gradientStop2);

GradientStop gradientStop3 = new GradientStop();
gradientStop3.Offset = 1;
gradientStop3.Color = Colors.Violet;
brush.GradientStops.Add(gradientStop3);

grid1.Background = brush;
```

Markup Extensions

For most properties, the XAML property syntax works perfectly well. But in some cases, it just isn't possible to hard-code the property value. For example, you may want to set a property value to an object that already exists. Or you may want to set a property value *dynamically* by binding it to a property in another control. In both of these cases, you need to use a *markup extension*—specialized syntax that sets a property in a nonstandard way.

Markup extensions can be used in nested tags or in XML attributes, which is more common. When they're used in attributes, they are always bracketed by curly braces {}. For example, here's how you can use the StaticExtension, which allows you to refer to a static property in another class:

```
<Button ... Foreground="{x:Static SystemColors.ActiveCaptionBrush}" >
```

Markup extensions use the syntax {MarkupExtensionClass Argument}. In this case, the markup extension is the StaticExtension class. (By convention, you can drop the final word *Extension* when referring to an extension class.) The x prefix indicates that the StaticExtension is found in one of the XAML namespaces. You'll also encounter markup extensions that are part of the WPF namespaces and don't have the x prefix.

All markup extensions are implemented by classes that derive from System.Windows.Markup.MarkupExtension. The base MarkupExtension class is extremely simple—it provides a single ProvideValue() method that gets the value you want. In other words, when the XAML parser encounters the previous statement, it creates an instance of the StaticExtension class (passing in the string "SystemColors.ActiveCaptionBrush" as an argument to the constructor) and then calls ProvideValue() to get the object returned by the SystemColors.ActiveCaption.Brush static property. The Foreground property of the cmdAnswer button is then set with the retrieved object.

The end result of this piece of XAML is the same as if you'd written this:

```
cmdAnswer.Foreground = SystemColors.ActiveCaptionBrush;
```

Because markup extensions map to classes, they can also be used as nested properties, as you learned in the previous section. For example, you can use the StaticExtension with the Button.Foreground property like this:

```
<Button ... >
  <Button.Foreground>
    <x:Static Member="SystemColors.ActiveCaptionBrush"></x:Static>
  </Button.Foreground>
</Button>
```

Depending on the complexity of the markup extension and the number of properties you want to set, this syntax is sometimes simpler.

Like most markup extensions, the StaticExtension needs to be evaluated at runtime because only then can you determine the current system colors. Some markup extensions can be evaluated at compile time. These include the NullExtension (which represents a null value) and the TypeExtension (which constructs an object that represents a .NET type). Throughout this book, you'll see many examples of markup extensions at work, particularly with resources and data binding.

Attached Properties

Along with ordinary properties, XAML also includes the concept of *attached properties*—properties that may apply to several controls but are defined in a different class. In WPF, attached properties are frequently used to control layout.

Here's how it works. Every control has its own set of intrinsic properties. (For example, a text box has a specific font, text color, and text content as dictated by properties such as FontFamily, Foreground, and Text.) When you place a control inside a container, it gains additional features depending on the type of container. (For example, if you place a text box inside a grid, you need to be able to choose the grid cell where it's positioned.) These additional details are set using attached properties.

Attached properties always use a two-part name in this form: DefiningType.PropertyName. This two-part naming syntax allows the XAML parser to distinguish between a normal property and an attached property.

In the eight ball example, attached properties allow the individual controls to place themselves on separate rows in the (invisible) grid:

```
<TextBox ... Grid.Row="0">
  [Place question here.]
</TextBox>

<Button ... Grid.Row="1">
  Ask the Eight Ball
</Button>

<TextBox ... Grid.Row="2">
  [Answer will appear here.]
</TextBox>
```

Attached properties aren't really properties at all. They're actually translated into method calls. The XAML parser calls the static method that has this form: *DefiningType*.Set*PropertyName*(). For example, in the previous XAML snippet, the defining type is the Grid class, and the property is Row, so the parser calls Grid.SetRow().

When calling SetPropertyName(), the parser passes two parameters: the object that's being modified and the property value that's specified. For example, when you set the Grid.Row property on the TextBox control, the XAML parser executes this code:

```
Grid.SetRow(txtQuestion, 0);
```

This pattern (calling a static method of the defining type) is a convenience that conceals what's really taking place. To the casual eye, this code implies that the row number is stored in the Grid object. However, the row number is actually stored in the object that it *applies to*—in this case, the TextBox object.

This sleight of hand works because the TextBox derives from the DependencyObject base class, as do all WPF controls. And as you'll learn in Chapter 4, the DependencyObject is designed to store a virtually unlimited collection of dependency properties. (The attached properties that were discussed earlier are a special type of dependency property.)

In fact, the Grid.SetRow() method is actually a shortcut that's equivalent to calling DependencyObject.SetValue() method, as shown here:

```
txtQuestion.SetValue(Grid.RowProperty, 0);
```

Attached properties are a core ingredient of WPF. They act as an all-purpose extensibility system. For example, by defining the Row property as an attached property, you guarantee that it's usable with any control. The other option, making it part of a base class such as FrameworkElement, complicates life. Not only would it clutter the public interface with properties that only have meaning in certain circumstances (in this case, when an element is being used inside a Grid), it also makes it impossible to add new types of containers that require new properties.

■ **Note** Attached properties are very similar to *extender providers* in a Windows Forms application. Both allow you to add "virtual" properties to extend another class. The difference is that you must create an instance of an extender provider before you can use it, and the extended property value is stored in the extender provider, not the extended control. The attached property design is a better choice for WPF because it avoids lifetime management issues (for example, deciding when to dispose of an extender provider).

Nesting Elements

As you've seen, XAML documents are arranged as a heavily nested tree of elements. In the current example, a Window element contains a Grid element, which contains TextBox and Button elements.

XAML allows each element to decide how it deals with nested elements. This interaction is mediated through one of three mechanisms that are evaluated in this order:

- If the parent implements IList, the parser calls IList.Add() and passes in the child.
- If the parent implements IDictionary, the parser calls IDictionary.Add() and passes in the child. When using a dictionary collection, you must also set the x:Key attribute to give a key name to each item.
- If the parent is decorated with the ContentProperty attribute, the parser uses the child to set that property.

For example, earlier in this chapter you saw how a LinearGradientBrush can hold a collection of GradientStop objects using syntax like this:

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.00" Color="Red" />
    <GradientStop Offset="0.50" Color="Indigo" />
    <GradientStop Offset="1.00" Color="Violet" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

The XAML parser recognizes the LinearGradientBrush.GradientStops element is a complex property because it includes a period. However, it needs to process the tags inside (the three GradientStop elements) a little differently. In this case, the parser recognizes that the GradientStops property returns a GradientStopCollection object, and the GradientStopCollection implements the IList interface. Thus, it assumes (quite rightly) that each GradientStop should be added to the collection using the IList.Add() method:

```
GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Red;
IList list = brush.GradientStops;
list.Add(gradientStop1);
```

Some properties might support more than one type of collection. In this case, you need to add a tag that specifies the collection class, like this:

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStopCollection>
      <GradientStop Offset="0.00" Color="Red" />
      <GradientStop Offset="0.50" Color="Indigo" />
      <GradientStop Offset="1.00" Color="Violet" />
    </GradientStopCollection>
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

■ **Note** If the collection defaults to null, you need to include the tag that specifies the collection class, thereby creating the collection object. If there's a default instance of the collection and you simply need to fill it, you can omit that part.

Nested content doesn't always indicate a collection. For example, consider the Grid element, which contains several other controls:

```
<Grid Name="grid1">
  ...
  <TextBox Name="txtQuestion" ... >
    ...
  </TextBox>
  <Button Name="cmdAnswer" ... >
    ...
  </Button>
  <TextBox Name="txtAnswer" ... >
    ...
  </TextBox>
</Grid>
```

These nested tags don't correspond to complex properties because they don't include the period. Furthermore, the Grid control isn't a collection and so it doesn't implement IList or IDictionary. What the Grid *does* support is the ContentProperty attribute, which indicates the property that should receive any nested content. Technically, the ContentProperty attribute is applied to the Panel class, from which the Grid derives, and looks like this:

```
[ContentPropertyAttribute("Children")]
public abstract class Panel
```

This indicates that any nested elements should be used to set the Children property. The XAML parser treats the content property differently depending on whether it's a collection property (in which case it implements the IList or IDictionary interface). Because the Panel.Children property returns a UIElementCollection and because UIElementCollection implements IList, the parser uses the IList.Add() method to add nested content to the grid.

In other words, when the XAML parser meets the previous markup, it creates an instance of each nested element and passes it to the Grid using the Grid.Children.Add() method:

```
txtQuestion = new TextBox();
...
grid1.Children.Add(txtQuestion);

cmdAnswer = new Button();
...
grid1.Children.Add(cmdAnswer);

txtAnswer = new TextBox();
...
grid1.Children.Add(txtAnswer);
```

What happens next depends entirely on how the control implements the content property. The Grid displays all the controls it holds in an invisible layout of rows and columns, as you'll see in Chapter 3.

The ContentProperty attribute is frequently used in WPF. Not only is it used for container controls (such as Grid) and controls that contain a collection of visual items (such as the ListBox and TreeView), it's also used for controls that contain singular content. For example, the TextBox and Button are able to hold only a single element or piece of text, but they both use a content property to deal with nested content like this:

```
<TextBox Name="txtQuestion" ... >
  [Place question here.]
</TextBox>
<Button Name="cmdAnswer" ... >
  Ask the Eight Ball
</Button>
<TextBox Name="txtAnswer" ... >
  [Answer will appear here.]
</TextBox>
```

The TextBox class uses the ContentProperty attribute to flag the TextBox.Text property. The Button class uses the ContentProperty attribute to flag the Button.Content property. The XAML parser uses the supplied text to set these properties.

The TextBox.Text property only allows strings. However, Button.Content is much more interesting. As you'll learn in Chapter 6, the Content property accepts any element. For example, here's a button that contains a shape object:

```
<Button Name="cmdAnswer" ... >
  <Rectangle Fill="Blue" Height="10" Width="100" />
</Button>
```

Because the Text and Content properties don't use collections, you can't include more than one piece of content. For example, if you attempt to nest multiple elements inside a Button, the XAML parser will throw an exception. The parser also throws an exception if you supply nontext content (such as a Rectangle).

■ **Note** As a general rule of thumb, all controls that derive from ContentControl allow a single nested element. All controls that derive from ItemsControl allow a collection of items that map to some part of the control (such as a list of items or a tree of nodes). All controls that derive from Panel are containers that are used to organize groups of controls. The ContentControl, ItemsControl, and Panel base classes all use the ContentProperty attribute.

Special Characters and Whitespace

XAML is bound by the rules of XML. For example, XML pays special attention to a few specific characters, such as & and < and >. If you try to use these values to set the content of an element, you'll run into trouble because the XAML parser assumes you're trying to do something else—such as create a nested element.

For example, imagine you want to create a button that contains the text <Click Me>. The following markup won't work:

```
<Button ... >
  <Click Me>
</Button>
```

The problem here is that it looks like you're trying to create an element named Click with an attribute named Me. The solution is to replace the offending characters with entity references—specific codes that the XAML parser will interpret correctly. Table 2-1 lists the character entities you might choose to use. Note that the quotation mark character entity is required only when setting values using an attribute because the quotation mark indicates the beginning and ending of an attribute value.

Table 2-1. *XML Character Entities*

Special Character	**Character Entity**
Less than (<)	<
Greater than (>)	>
Ampersand (&)	&
Quotation mark (")	"

Here's the corrected markup that uses the appropriate character entities:

```
<Button ... >
  &lt;Click Me&gt;
</Button>
```

When the XAML parser reads this, it correctly understands that you want to add the text <Click Me>, and it passes a string with this content, complete with angled brackets, to the Button.Content property.

■ **Note** This limitation is a XAML detail, and it won't affect you if you want to set the Button.Content property in code. Of course, C# has its own special character (the backslash) that must be escaped in string literals for the same reason.

Special characters aren't the only stumbling block you'll run into with XAML. Another issue is whitespace handling. By default, XML collapses all whitespace, which means a long string of spaces, tabs, and hard returns is reduced to a single space. Furthermore, if you add whitespace before or after your element content, this space is ignored completely. You can see this in the EightBall example. The text in the button and the two text boxes is separated from the XAML tags using a hard return and tab to make the markup more readable. However, this extra space doesn't appear in the user interface.

Sometimes this isn't what you want. For example, you may want to include a series of several spaces in your button text. In this case, you need to use the xml:space="preserve" attribute on your element.

The xml:space attribute is part of the XML standard, and it's an all-or-nothing setting. Once you switch it on, all the whitespace inside that element is retained. For example, consider this markup:

```
<TextBox Name="txtQuestion" xml:space="preserve" ...>
      [There is a lot of space inside these quotation marks "         ".]
</TextBox>
```

In this example, the text in the text box will include the hard return and tab that appear before the actual text. It will also include the series of spaces inside the text and the hard return that follows the text.

If you just want to keep the spaces inside, you'll need to use this less-readable markup:

```
<TextBox Name="txtQuestion" xml:space="preserve" ...
 >[There is a lot of space inside these quotation marks "         ".]</TextBox>
```

The trick here is to make sure no whitespace appears between the opening > and your content, or between your content and the closing <.

Once again, this issue applies only to XAML markup. If you set the text in a text box programmatically, all the spaces you include are used.

Events

So far, all the attributes you've seen map to properties. However, attributes can also be used to attach event handlers. The syntax for this is EventName="EventHandlerMethodName".

For example, the Button control provides a Click event. You can attach an event handler like this:

```
<Button ... Click="cmdAnswer_Click">
```

This assumes that there is a method with the name cmdAnswer_Click in the code-behind class. The event handler must have the correct signature (that is, it must match the delegate for the Click event). Here's the method that does the trick:

```
private void cmdAnswer_Click(object sender, RoutedEventArgs e)
{
    this.Cursor = Cursors.Wait;

    // Dramatic delay...
    System.Threading.Thread.Sleep(TimeSpan.FromSeconds(3));

    AnswerGenerator generator = new AnswerGenerator();
    txtAnswer.Text = generator.GetRandomAnswer(txtQuestion.Text);
    this.Cursor = null;
}
```

As you may have noticed from the signature of this event handler, the event model in WPF is different than in earlier versions of .NET. It supports a new model that relies on *event routing*. You'll learn more in Chapter 5.

In many situations, you'll use attributes to set properties and attach event handlers on the same element. WPF always follows the same sequence: first it sets the Name property (if set), then it attaches any event handlers, and lastly it sets the properties. This means that any event handlers that respond to property changes will fire when the property is set for the first time.

■ **Note** It's possible to embed code (such as event handlers) directly in a XAML document using the Code element. However, this technique is thoroughly discouraged, and it doesn't have any practical application in WPF. This approach isn't supported by Visual Studio, and it isn't discussed in this book.

Visual Studio helps you out with IntelliSense when you add an event handler attribute. Once you enter the equals sign (for example, after you've typed **Click=** in the <Button> element), it shows a drop-down list with all the suitable event handlers in your code-behind class, as shown in Figure 2-2. If you need to create a new event handler to handle this event, you simply need to choose <New Event Handler> from the top of the list. Alternatively, you can attach and create event handlers using the Events tab of the Properties window.

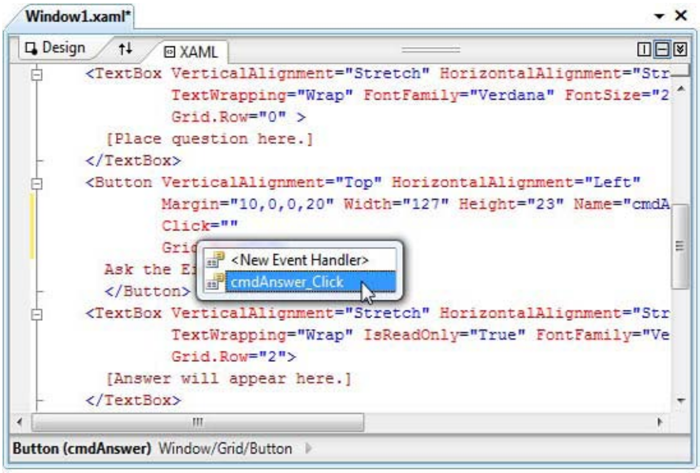

Figure 2-2. *Attaching an event with Visual Studio IntelliSense*

The Full Eight Ball Example

Now that you've considered the fundamentals of XAML, you know enough to walk through the definition for the window in Figure 2-1. Here's the complete XAML markup:

```
<Window x:Class="EightBall.Window1"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="Eight Ball Answer" Height="328" Width="412" >
  <Grid Name="grid1">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <TextBox VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
     Margin="10,10,13,10" Name="txtQuestion"
     TextWrapping="Wrap" FontFamily="Verdana" FontSize="24"
     Grid.Row="0">
      [Place question here.]
    </TextBox>
    <Button VerticalAlignment="Top" HorizontalAlignment="Left"
     Margin="10,0,0,20" Width="127" Height="23" Name="cmdAnswer"
     Click="cmdAnswer_Click" Grid.Row="1">
      Ask the Eight Ball
    </Button>
    <TextBox VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
     Margin="10,10,13,10" Name="txtAnswer" TextWrapping="Wrap"
     IsReadOnly="True" FontFamily="Verdana" FontSize="24" Foreground="Green"
     Grid.Row="2">
```

```
      [Answer will appear here.]
    </TextBox>

    <Grid.Background>
      <LinearGradientBrush>
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0.00" Color="Red" />
          <GradientStop Offset="0.50" Color="Indigo" />
          <GradientStop Offset="1.00" Color="Violet" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>
    </Grid.Background>
  </Grid>
</Window>
```

Remember, you probably won't write the XAML for an entire user interface by hand—doing so would be unbearably tedious. However, you might have good reason to edit the XAML code to make a change that would be awkward to accomplish in the designer. You might also find yourself reviewing XAML to get a better idea of how a window works.

Using Types from Other Namespaces

So far, you've seen how to create a basic user interface in XAML using the classes that are part of WPF. However, XAML is designed as an all-purpose way to instantiate .NET objects, including ones that are in other non-WPF namespaces and those you create yourself.

It might seem odd to consider creating objects that aren't designed for onscreen display in a XAML window, but it makes sense in a number of scenarios. One example is when you use data binding and you want to draw information from another object to display in a control. Another example is if you want to set the property of a WPF object using a non-WPF object.

For example, you can fill a WPF ListBox with data objects. The ListBox will call the ToString() method to get the text to display for each item in the list. (Or for an even better list, you can create a *data template* that extracts multiple pieces of information and formats them appropriately. This technique is described in Chapter 20.)

To use a class that isn't defined in one of the WPF namespaces, you need to map the .NET namespace to an XML namespace. XAML has a special syntax for doing this, which looks like this:

```
xmlns:Prefix="clr-namespace:Namespace;assembly=AssemblyName"
```

Typically, you'll place this namespace mapping in the root element of your XAML document, right after the attributes that declare the WPF and XAML namespaces. You'll also fill in the three italicized bits with the appropriate information, as explained here:

- **Prefix.** This is the XML prefix you want to use to indicate that namespace in your XAML markup. For example, the XAML language uses the x prefix.
- **Namespace.** This is the fully qualified .NET namespace name.
- **AssemblyName.** This is the assembly where the type is declared, without the .dll extension. This assembly must be referenced in your project. If you want to use your project assembly, leave this out.

For example, here's how you would gain access to the basic types in the System namespace and map them to the prefix *sys*:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Here's how you would gain access to the types you've declared in the MyProject namespace of the current project and map them to the prefix *local*:

```
xmlns:local="clr-namespace:MyNamespace"
```

Now, to create an instance of a class in one of these namespaces, you use the namespace prefix:

```
<local:MyObject ...></local:MyObject>
```

■ **Tip** Remember, you can use any namespace prefix you want, as long as you are consistent throughout your XAML document. However, the sys and local prefixes are commonly used when importing the System namespace and the namespace for the current project. You'll see them used throughout this book.

Ideally, every class you want to use in XAML will have a no-argument constructor. If it does, the XAML parser can create the corresponding object, set its properties, and attach any event handlers you supply. XAML doesn't support parameterized constructors, and all the elements in WPF elements include a no-argument constructor. Additionally, you need to be able to set all the details you want using public properties. XAML doesn't allow you to set public fields or call methods.

If the class you want to use doesn't have a no-argument constructor, you're in a bit of a bind. If you're trying to create a simple primitive (such as a string, date, or numeric type), you can supply the string representation of your data as content inside your tag. The XAML parser will then use the type converter to convert that string into the appropriate object. Here's an example with the DateTime structure:

```
<sys:DateTime>10/30/2010 4:30 PM</sys:DateTime>
```

This works because the DateTime class uses the TypeConverter attribute to link itself to the DateTimeConverter. The DateTimeConverter recognizes this string as a valid DateTime object and converts it. When you're using this technique, you can't use attributes to set any properties for your object.

If you want to create a class that doesn't have a no-argument constructor and there isn't a suitable type converter to use, you're out of luck.

■ **Note** Some developers get around these limitations by creating custom wrapper classes. For example, the FileStream class doesn't include a no-argument constructor. However, you could create a wrapper class that does. Your wrapper class would create the required FileStream object in its constructor, retrieve the information it needs, and then close the FileStream. This type of solution is seldom ideal because it invites hard-coding information in your class constructor and it complicates exception handling. In most cases, it's a better idea to manipulate the object with a little event handling code and leave it out of your XAML entirely.
