Chapter 3

The Concept of Content

The Window class has more than 100 public properties, and some of them—such as the Title property that identifies the window—are quite important. But by far the most important property of Window is the Content property. You set the Content property of the window to the object you want in the window’s client area.

You can set the Content property to a string, you can set it to a bitmap, you can set it to a drawing, and you can set it to a button, or a scrollbar, or any one of 50-odd controls supported by the Windows Presentation Foundation. You can set the Content property to just about anything. But there’s only one little problem:

You can only set the Content property to one object.

This restriction is apt to be a bit frustrating in the early stages of working with content. Eventually, of course, you’ll see how to set the Content property to an object that can play host to multiple other objects. For now, working with a single content object will keep us busy enough.

The Window class inherits the Content property from ContentControl, a class that derives from Control and from which Window immediately descends. The ContentControl class exists almost solely to define this Content property and a few related properties and methods.

The Content property is defined as type object, which suggests that it can be set to any object, and that’s just about true. I say “just about” because you cannot set the Content property to another object of type Window. You’ll get a run-time exception that indicates that Window must be the “root of a tree,” not a branch of another Window object.

You can set the Content property to a text string, for example:

DisplaySomeText.cs

```csharp
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.DisplaySomeText
{
    public class DisplaySomeText : Window
    {
        [STAThread]
    }
}
```
This program displays the text “Content can be simple text!” in the upper-left corner of the client area. If you make the window too narrow to fit all the text, you’ll find that the text is truncated rather than automatically wrapped (alas), but you can insert line breaks in the text using the carriage return character (“\r”) or a line feed (“\n”), or both: “\r\n”.

The program includes a using directive for the System.Windows.Media namespace so that you can experiment with properties that affect the color and font of the text. These properties are all defined by the Control class from which Window derives.

Nothing except good taste will prevent you from setting the font used to display text in the client area like so:

```csharp
FontFamily = new FontFamily(“Comic Sans MS”);
FontSize = 48;
```

This code can go anywhere in the constructor of the class.

There is no such thing as a Font class in the WPF. The first line of this code makes reference to a FontFamily object. A font family (also known as a type family) is a collection of related typefaces. Under Windows, font families have familiar names such as Courier New, Times New Roman, Arial, Palatino Linotype, Verdana, and, of course, Comic Sans MS.

A typeface (also known as a face name) is the combination of a font family and a possible variation, such as Times New Roman Bold, Times New Roman Italic, and Times New Roman Bold Italic. Not every variation is available in every font family, and some font families have variations that affect the widths of individual characters, such as Arial Narrow.

The term font is generally used to denote a combination of a particular typeface with a particular size. The common measurement of fonts is the em size. (The term comes from the size of the square piece of metal type used in olden days for the capital M.) The em size is commonly described as the height of the characters in the Latin alphabet—the uppercase and lowercase letters A through Z without diacritical marks—from the very top of the ascenders to the bottom of the descenders. However, the em size is not a metrical concept. It is a typographical
design concept. The actual size of characters in a particular font could be somewhat greater than or less than what the em size implies.

Commonly, the em size is specified in a unit of measurement known as the point. In traditional typography, a point is 0.01384 inch, but in computer typography, the point is assumed to be exactly 1/72 inch. Thus, a 36-point em size (often abbreviated as a 36-point font) refers to characters that are about 1/2 inch tall.

In the Windows Presentation Foundation, you set the em size you want by using the FontSize property. But you don't use points. Like every measurement in the WPF, you specify the FontSize in device-independent units, which are 1/96 inch. Setting the FontSize property to 48 results in an em size of 1/2 inch, which is equivalent to 36 points.

If you're accustomed to specifying em sizes in points, just multiply the point size by 4/3 (or divide by 0.75) when setting the FontSize property. If you're not accustomed to specifying em sizes in points, you should get accustomed to it, and just multiply the point size by 4/3 when setting the FontSize property.

The default FontSize property is 11, which is 8.25 points. Much of The New York Times is printed in an 8-point type. Newsweek uses a 9-point type. This book has 10-point type.

You can use a full typeface name in the FontFamily constructor:

```csharp
FontFamily = new FontFamily("Times New Roman Bold Italic");
FontSize = 32;
```

That's a 24-point Times New Roman Bold Italic font. However, it's more common to use the family name in the FontFamily constructor and indicate bold and italic by setting the FontStyle and FontWeight properties:

```csharp
FontFamily = new FontFamily("Times New Roman");
FontSize = 32;
FontStyle = FontStyles.Italic;
FontWeight = FontWeights.Bold;
```

Notice that the FontStyle and FontWeight properties are set to static read-only properties of the FontStyles (plural) and FontWeights (plural) classes. These static properties return objects of type FontStyle and FontWeight, which are structures that have limited use by themselves.

Here's an interesting little variation:

```csharp
FontStyle = FontStyles.Oblique;
```

An italic typeface is often stylistically a bit different from the non-italic (or roman) typeface. Look at the lowercase “a” to see the difference. But an oblique typeface simply slants all the letters of the roman typeface to the right. For some font families, you can set a FontStretch property to a static property of the FontStretches class.
Part I: Code

You’re already familiar with the Background property that colors the background of the client area. The Foreground property colors the text itself. Try this:

```csharp
Brush brush = new LinearGradientBrush(Colors.Black, Colors.White,
    new Point(0, 0), new Point(1, 1));
Background = brush;
Foreground = brush;
```

Both the foreground and background are now colored with the same brush, which you might fear would potentially render the foreground text invisible. This doesn’t happen, however. As you discovered in Chapter 2, gradient brushes used for coloring the background are by default automatically adjusted to the size of the client area. Similarly, the foreground brush is automatically adjusted to the size of the content—the actual text string. Changing the size of the window does not affect that foreground brush. But when you make the client area exactly the same size as the size of the text, the two brushes coincide and the text disappears.

Now try this:

```csharp
SizeToContent = SizeToContent.WidthAndHeight;
```

The SizeToContent property defined by the Window class causes the window to adjust itself to be just as big as the size of its content. If you’re still using the same LinearGradientBrush for foreground and background, you won’t be able to see the text. You set the SizeToContent property to a member of the SizeToContent enumeration: Manual (which is the default), Width, Height, or WidthAndHeight. With the latter three, the window adjusts its width, height, or both the size of its contents. This is a very handy property that you’ll often use when designing dialog boxes or other forms. When setting a window to the size of its content, you’ll often want to suppress the sizing border with the following:

```csharp
ResizeMode = ResizeMode.CanMinimize;
```

or

```csharp
ResizeMode = ResizeMode.NoResize;
```

You can add a border inside the client area with the code shown at the end of Chapter 2:

```csharp
BorderBrush = Brushes.SaddleBrown;
BorderThickness = new Thickness(25, 50, 75, 100);
```

You’ll see that both the foreground brush and SizeToContent take account of this border. The content always appears inside this border.

The display of a text string by the DisplaySomeText program is actually much more generalized than it may at first appear. As you know, all objects have a Tostring method that’s supposed to return a string representation of the object. The window uses the Tostring method to display the object. You can convince yourself of this by setting the Content property to something other than a string:

```csharp
Content = Math.PI;
```
Or try this:

```
Content = DateTime.Now;
```

In both cases, what the window displays is identical to the string returned from `ToString`. If the object is based on a class that doesn’t override `ToString`, the default `ToString` method just displays the fully qualified class name. For example:

```
Content = EventArgs.Empty;
```

That displays the string ”System.EventArgs”. The only exception I’ve found is for arrays. If you do something like this

```
Content = new int[57];
```

the window displays the text string ”Int32[]” Array while the `ToString` method returns ”System.Int32[]”.

Here’s a program that sets the `Content` property to an empty text string, but then adds characters to this text string based on input from the keyboard. It’s similar to the TypeYourTitle program from the first chapter, but this one also lets you enter carriage returns and tabs.

```csharp
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.RecordKeystrokes
{
    public class RecordKeystrokes : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new RecordKeystrokes());
        }

        public RecordKeystrokes()
        {
            Title = "Record Keystrokes";
            Content = "";
        }
        protected override void OnTextInput(TextCompositionEventArgs args)
        {
            base.OnTextInput(args);
            string str = Content as string;
            if (args.Text == "\b")
            {
```
```csharp
if (str.Length > 0)
    str = str.Substring(0, str.Length - 1);
else
    
    str += args.Text;

Content = str;
}
}
```

The RecordKeystrokes program works only because the `Content` property is changing with each keystroke, and the `Window` class is responding to this change by redrawing the client area. It is very easy to make a little variation of this program that won’t work at all. For example, define an empty string as a field:

```csharp
string str = "";
```

Set the `Content` property to this variable in the constructor:

```csharp
Content = str;
```

Remove that same statement from the `OnTextInput` override, and also remove the definition of `str`. Now the program doesn’t work. Take a look at the compound assignment statement in `OnTextInput`:

```csharp
str += args.Text;
```

It’s equivalent to the following statement:

```csharp
str = str + args.Text;
```

The problem with both statements is that the `string` object returned from the concatenation operation is not the same `string` object that entered it. Strings, remember, are immutable. The concatenation operation creates a new string, but the `Content` property is still set to the original string.

Now try something a little different. You’ll need a `using` directive for `System.Text` for this. Define a `StringBuilder` object as a field:

```csharp
StringBuilder build = new StringBuilder("text");
```

In the program’s constructor, set `Content` equal to that object:

```csharp
Content = build;
```

You should be confident (and you would be correct) in assuming that the window will display “text” because the `ToString` method of the `StringBuilder` object returns the string that it has built.
Replace the code in the `OnTextInput` method with this:

```java
if (args.Text == "\b")
{
    if (build.Length > 0)
        build.Remove(build.Length - 1, 1);
} else
{
    build.Append(args.Text);
}
```

And, again the code doesn’t work. Although there’s only one `StringBuilder` object in this program, the window has no way of knowing when the string stored by this `StringBuilder` object changes, so it doesn’t update the window with the new text string.

I’m pointing out these cases because sometimes objects in the Windows Presentation Foundation seem to update themselves as if by magic. It may seem like magic, but it’s not. There’s always some method of notification in the form of an event. Being aware of what’s going on (or not going on) can help you better understand the environment. The revised RecordKey-strokes program with the `StringBuilder` object won’t even work if you insert the following statement

```
Content = build;
```

at the bottom of the `OnTextInput` override. The window is smart enough to know that you’re assigning the same object to `Content` that you’ve already set to `Content`, so no update is necessary. But try these statements:

```java
Content = null;
Content = build;
```

That code works.

We’ve seen that window content can be plain text. But the purpose of the `Content` property is not to display simple unformatted text. No, no, no, no, no. What the `Content` property really wants is something more graphical in nature, and that’s an instance of any class that derives from `UIElement`.

`UIElement` is an extremely important class in the Windows Presentation Foundation. This is the class that implements keyboard, mouse, and stylus handling. The `UIElement` class also contains an important method named `OnRender`. The `OnRender` method is invoked to obtain the graphical representation of an object. (You’ll see an example at the end of this chapter.)

As far as the `Content` property goes, the world is divided into two groups of objects: Those that derive from `UIElement` and those that do not. In the latter group, the object is displayed with `ToString`; in the former group, it’s displayed with `OnRender`. Classes that derive from `UIElement` (and their instantiated objects) are often referred to collectively as `elements`. 
The only class that inherits directly from `UIElement` is `FrameworkElement`, and all the elements you'll encounter in the Windows Presentation Foundation derive from `FrameworkElement`. In theory, `UIElement` provides the necessary structure of user interface events and screen rendering that can support various programming frameworks. The Windows Presentation Foundation is one such framework and consists of all the classes that derive from `FrameworkElement`. In a practical sense, you probably won't make much of a distinction between the properties, methods, and events defined by `UIElement` and those defined by `FrameworkElement`.

One popular class that inherits from `FrameworkElement` is `Image`. Here's the class hierarchy:

```
Object
  DispatcherObject (abstract)
    DependencyObject
      Visual (abstract)
        UIElement
          FrameworkElement
            Image
```

The `Image` class lets you easily include an image in a document or an application. Here's a program that pulls a bitmap from my Web site and displays it in the window:

```csharp
ShowMyFace.cs
//------------------------------------------------------------------------------
// ShowMyFace.cs (c) 2006 by Charles Petzold
//------------------------------------------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.ShowMyFace
{
    class ShowMyFace : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new ShowMyFace());
        }
    }
}
```
Chapter 3: The Concept of Content

```java
public ShowMyFace()
{
    Title = "Show My Face";

    Uri uri = new Uri("http://www.charlespetzold.com/PetzoldTattoo.jpg");
    BitmapImage bitmap = new BitmapImage(uri);
    Image img = new Image();
    img.Source = bitmap;
    Content = img;
}
```

A couple of steps are required to display an image. The program first creates an object of `Uri` that indicates the location of the bitmap. This is passed to the `BitmapImage` constructor, which actually loads the image into memory. (Many popular formats are supported, including GIF, TIFF, JPEG, and PNG.) The `Image` class displays the image in the window. It is an instance of the `Image` class that you set to the `Content` property of the window.

You can find the `Image` class in the `System.Windows.Controls` namespace. Strictly speaking, `Image` is not considered a control and does not derive from the `Control` class. But the `System.Windows.Controls` namespace is so important that I’ll include it in most programs from here on. The `BitmapImage` class is located in the `System.Windows.Media.Imaging` namespace, which is certainly an important namespace if you’re working with bitmaps. However, I won’t generally include a using directive for this namespace unless the program requires it.

Alternatively, you can load an image file from a local disk drive. You’ll need a fully qualified file name as the argument in the `Uri` constructor, or you’ll need to preface the relative file name with “file://”. Here’s some replacement code that retrieves the image of the fly fisherman scooping up a yummy trout:

```java
Uri uri = new Uri(
    System.IO.Path.Combine(
        Environment.GetEnvironmentVariable("windir"), "Gone Fishing.bmp"));
```

The `Environment.GetEnvironmentVariable` method retrieves the "windir" environment variable, which is a string like "C:\WINDOWS". The `Path.Combine` method combines that path name with the file name of the desired bitmap so that I (the lazy programmer) don’t have to worry about inserting slashes correctly. You can either preface the `Path` class with `System.IO` (as I’ve done) or include a using directive for that namespace.

Rather than passing the `Uri` object to the `BitmapImage` constructor, you can set the `Uri` object to the `UriSource` property of `BitmapImage`. However, it’s recommended that you surround the setting of this property with calls to the `BeginInit` and `EndInit` methods in `BitmapImage`:

```java
bitmap.BeginInit();
bitmap.UriSource = uri;
bitmap.EndInit();
```
As you know, bitmaps have a pixel width and a pixel height, which is to the number of picture elements actually encoded in the file. The BitmapImage class inherits integer read-only PixelWidth and PixelHeight properties from BitmapSource that reveals this information. Bitmaps often (but not always) have embedded resolution information. Sometimes this resolution information is important (a scan of a rare postage stamp) and sometimes it’s not (a photo of your postal carrier). The resolution information in dots per inch is available from the read-only double values DpiX and DpiY. The BitmapImage class also includes read-only double properties named Width and Height. These values are calculated with the following formulas:

\[
\text{Width} = \frac{96 \cdot \text{PixelWidth}}{\text{DpiX}}
\]

\[
\text{Height} = \frac{96 \cdot \text{PixelWidth}}{\text{DpiY}}
\]

Without the value of 96 in the numerator, these formulas would give a width and height of the image in inches. The 96 converts inches into device-independent units. The Height and Width properties constitute the metrical size of the bitmap in device-independent units. BitmapImage also inherits from BitmapSource a Format property that provides the color format of the bitmap; for those bitmaps that have a color table, the Palette property gives access to that.

Regardless of whether the ShowMyFace program is displaying my face, or the trout fisherman, or an image of your choice, you’ll notice that the image is displayed as large as possible within the confines of the window, but without distortion. Unless the aspect ratio of the client area is exactly that of the window, you’ll see some of the client window background on either the top and bottom sides or the left and right sides of the image.

The size of the image within the window is governed by several properties of Image, one of which is Stretch. By default, the Stretch property equals the enumeration value Stretch.Uniform, which means that the image is increased or decreased in size uniformly (that is, the same in both the horizontal and vertical directions) to fill the client area.

You can alternatively set the Stretch property to Stretch.Fill:

```csharp
img.Stretch = Stretch.Fill;
```

This setting causes the image to fill up the entire window, generally distorting the image by increasing or decreasing it by different amounts horizontally or vertically. The Stretch.UniformToFill option stretches the image uniformly, but also completely fills the client area. It performs this amazing feat by truncating the image on one side.

The Stretch.None option causes the image to be displayed in its metrical size, which is obtainable from the Width and Height properties of the BitmapSource.

If you use a Stretch option other than Stretch.None, you can also set the StretchDirection property of Image. The default is the enumeration value StretchDirection.Both, which means that the image can be stretched greater than or less than its metrical size. With StretchDirection.DownOnly, the
image is never larger than its metrical size, and with StretchDirection.UpOnly, the image is never smaller than its metrical size.

Regardless of how you size the image, the image is always (except for Stretch.UniformToFill) positioned in the center of the window. You can change that by setting the HorizontalAlignment and VerticalAlignment properties that Image inherits from FrameworkElement. For example, this code moves the image to the upper right of the client area:

```csharp
img.HorizontalAlignment = HorizontalAlignment.Right;
img.VerticalAlignment = VerticalAlignment.Top;
```

The HorizontalAlignment and VerticalAlignment properties have a surprisingly important role in layout in the Windows Presentation Foundation. You'll encounter these properties again and again. If you'd prefer the Image object to be in the upper-right corner, but not flush against the edges, you can set a margin around the Image object:

```csharp
img.Margin = new Thickness(10);
```

Margin is defined in FrameworkElement and often used to insert a little breathing room around elements. You can use the Thickness structure to define either the same margin on all four sides (in this case 10/96 inch or about 0.1 inches) or different margins on all four sides. As you'll recall, the four-argument constructor of Thickness sets the fields in left-top-right-bottom order:

```csharp
img.Margin = new Thickness(192, 96, 48, 0);
```

Now the margin is 2 inches on the left, 1 inch on the top, 1/2 inch on the right, and nothing on the bottom. You can see how the margins are respected when you make the window too small for both the image and the margins: The image disappears before the margins do.

The Image object also has Width and Height properties that it inherits from FrameworkElement. These are read/write double values, and if you check their values, you'll see that they are undefined, which is indicated by values of “not a number,” or NaN. (It’s the same as with the Window object.) You can also set a precise Width and Height of the Image object, although these may not be consistent with some Stretch settings.

You can also size the window to the metrical size of the image:

```csharp
SizeToContent = SizeToContent.WidthAndHeight;
```

Setting the Foreground property of the Window object has no effect on the display of the image; the Foreground property only comes into play when the window content is text or (as we'll see) another type of element that displays text.

Normally, setting the Background property of the Window object only has an effect in the areas of the client area not covered by the image. But try this:

```csharp
img.Opacity = 0.5;
Background = new LinearGradientBrush(Colors.Red, Colors.Blue,
                                        new Point(0, 0), new Point(1, 1));
```
Part I: Code

Now the brush shows through the image. The Opacity property (which Image inherits from UIElement) is 1 by default, but you can set it to any value between 0 and 1 to make an element transparent. (It won’t work with the Window object itself, however.)

A full discussion of graphics transforms awaits us in Chapter 29 of this book, but for now you can see how easy it is to rotate a bitmap:

```csharp
img.LayoutTransform = new RotateTransform(45);
```

The Image class does not have its own Background and Foreground properties because those properties are defined by the Control class and Image is not derived from Control. The distinction may be a little confusing at first. In earlier Windows application programming interfaces, virtually everything on the screen was considered a control. That is obviously not the case here. A control is really just another visual object, and is characterized mostly by giving feedback to user input. Elements like Image can obtain user input, of course, because all the keyboard, mouse, and stylus input events are defined by UIElement.

Take a look at the namespace System.Windows.Shapes, which contains an abstract class named Shape and six other classes that derive from it. These classes also derive from UIElement by way of FrameworkElement:

```
Object
  DispatcherObject (abstract)
  DependencyObject
    Visual (abstract)
      UIElement
        FrameworkElement
          Shape (abstract)
            Ellipse
            Line
            Path
            Polygon
            Polyline
            Rectangle
```
While Image is the standard way to display raster images, these Shape classes implement simple two-dimensional vector graphics. Here’s a program that creates an object of type Ellipse:

```csharp
ShapeAnEllipse.cs
//-------------------------------
// ShapeAnEllipse.cs (c) 2006 by Charles Petzold
//-------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Petzold.ShapeAnEllipse
{
    class ShapeAnEllipse : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new ShapeAnEllipse());
        }
        public ShapeAnEllipse()
        {
            Title = "Shape an Ellipse";
            Ellipse elips = new Ellipse();
            elips.Fill = Brushes.AliceBlue;
            elips.StrokeThickness = 24;   // 1/4 inch
            elips.Stroke =
                new LinearGradientBrush(Colors.CadetBlue, Colors.Chocolate,
                    new Point(1, 0), new Point(0, 1));
            Content = elips;
        }
    }
}
```

The ellipse fills the client area. The circumference is a quarter-inch thick and is colored with a gradient brush. The interior is filled with an Alice Blue brush, the only color named after a daughter of Teddy Roosevelt.

Neither the Shape class nor the Ellipse class defines any properties to set a size of the ellipse, but the Ellipse class inherits Width and Height properties from FrameworkElement, and these do just fine:

```csharp
elips.Width = 300;
elips.Height = 300;
```
Just as with Image, you can now use the HorizontalAlignment and VerticalAlignment properties to position the ellipse in the center, horizontally to the left or right, or vertically at the top or bottom:

```csharp
eellipse.HorizontalAlignment = HorizontalAlignment.Left;
eellipse.VerticalAlignment = VerticalAlignment.Bottom;
```

Both the HorizontalAlignment and VerticalAlignment enumerations have members named Center, but they also have members named Stretch, and for many elements, Stretch is the default. Stretch is the default for Ellipse, and that’s why the Ellipse initially fills the client area. The element stretches to the boundaries of its container. In fact, if you set HorizontalAlignment and VerticalAlignment to anything other than Stretch without also setting explicit Width and Height properties, the ellipse collapses into a quarter-inch ball with only its perimeter showing.

If you don’t set the Width and Height properties, however, you can set any or all of the MinWidth, MaxWidth, MinHeight, and MaxHeight properties (all inherited from FrameworkElement) to restrict the ellipse to a particular range of sizes. By default, all these properties are undefined. At any time (except in the constructor of the window), the program can obtain the actual size of the ellipse from the read-only ActualWidth and ActualHeight properties.

If I seem a bit obsessive about the size of elements set as the content of a window, it’s only because this is an important issue. You are probably accustomed to assigning specific sizes to controls and other graphical objects, but the Windows Presentation Foundation doesn’t require that, and it is vital that you get a good feel for the way in which visual objects are sized.

What you won’t find in the Ellipse class are any properties that allow you to position the ellipse at a particular location in the client area of the window. The closest you can come at this point is through setting the HorizontalAlignment and VerticalAlignment properties.

Earlier I showed you how to set the Content property of the Window to any text string, and also how to set the font of that text. However, text that you set directly to the Content property of the window must have uniform formatting. You can’t specify that particular words are bold or italic, for example.

If you need to do that, instead of setting the Content property of your window to a string, you can set it to an object of type TextBlock:

```csharp
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Documents;
```
Chapter 3: The Concept of Content

namespace Petzold.FormatTheText
{
    class FormatTheText : window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new FormatTheText());
        }
        public FormatTheText()
        {
            Title = "Format the Text";
            TextBlock txt = new TextBlock();
            txt.FontSize = 32; // 24 points
            txt.Inlines.Add("This is some ");
            txt.Inlines.Add(new Italic(new Run("italic")));
            txt.Inlines.Add(" text, and this is some ");
            txt.Inlines.Add(new Bold(new Run("bold")));
            txt.Inlines.Add(" text, and let's cap it off with some ");
            txt.Inlines.Add(new Bold(new Italic(new Run("bold italic"))));
            txt.Inlines.Add(" text.");
            txt.TextWrapping = TextWrapping.Wrap;
            Content = txt;
        }
    }
}

Although this is the first program in this book that explicitly creates a TextBlock object, you’ve actually seen one before. If you set the Content property to a string, ContentControl (from which Window derives) creates an object of type TextBlock to actually display the string. The TextBlock class derives directly from FrameworkElement. It defines a property named Inlines, which is of type InlineCollection, which is a collection of Inline objects.

TextBlock itself is included in the System.Windows.Controls namespace but Inline is part of System.Windows.Documents, and it doesn’t even derive from UIElement. Here’s a partial class hierarchy showing Inline and some of its descendents:

Object

    DispatcherObject (abstract)

        DependencyObject

            ContentElement

                FrameworkContentElement

                    TextElement (abstract)
**Inline** (abstract)

Run

Span

**Bold**

**Hyperlink**

**Italic**

**Underline**

You might notice a somewhat parallel structure in this class hierarchy with the earlier ones. The **ContentElement** and **FrameworkContentElement** classes are analogous to the **UIElement** and **FrameworkElement** classes. However, the **ContentElement** class contains no **OnRender** method. Objects based on classes that derive from **ContentElement** do not draw themselves on the screen. Instead, they achieve a visual representation on the screen only through a class that drives from **UIElement**, which provides the necessary **OnRender** method.

More specifically, objects of the types **Bold** and **Italic** do not draw themselves. In the **FormatTheText** program, these **Bold** and **Italic** objects are rendered by the **TextBlock** object.

Don’t confuse the **ContentElement** class with **ContentControl**. A **ContentControl** is a control, such as **Window**, which contains a property named **Content**. The **ContentControl** object renders itself on the screen even if its **Content** property is null. But a **ContentElement** object must be part (that is, content) of some other element to be rendered.

The bulk of the **FormatTheText** program is devoted to assembling the **Inlines** collection of the **TextBlock**. The **InlineCollection** class implements **Add** methods for objects of **string**, **Inline**, and **UIElement** (the last of which lets you embed other elements in the **TextBlock**). However, the **Bold** and **Italic** constructors accept only **Inline** objects and not **string** objects, so the program uses a **Run** constructor first for each **Bold** or **Italic** object.

The program sets the **FontSize** property of the **TextBlock**:

```csharp
txt.FontSize = 32;
```

However, the program works the same way if it sets the **FontSize** property of the window instead:

```csharp
FontSize = 32;
```

Similarly, the program can set the **Foreground** property of the window and the **TextBlock** text appears in that color:

```csharp
Foreground = Brushes.CornflowerBlue;
```
Elements on the screen exist in a tree of parent-child hierarchies. The window is parent to the TextBlock, which is parent to a number of Inline elements. These elements inherit the values of the Foreground property and all font-related properties from their parent elements unless these properties are explicitly set on the children. You'll see how this works in Chapter 8.

Like the UIElement class, the ContentElement class defines many user-input events. It is possible to attach event handlers to the individual Inline elements that make up the text displayed by TextBlock, and the following program demonstrates this technique.

```
public class ToggleBoldAndItalic : Window
{
    [STAThread]
    public static void Main()
    {
        Application app = new Application();
        app.Run(new ToggleBoldAndItalic());
    }
    public ToggleBoldAndItalic()
    {
        Title = "Toggle Bold & Italic";
        TextBlock text = new TextBlock();
        text.FontSize = 32;
        text.HorizontalAlignment = HorizontalAlignment.Center;
        text.VerticalAlignment = VerticalAlignment.Center;
        Content = text;

        string strQuote = "To be, or not to be, that is the question";
        string[] strWords = strQuote.Split();

        foreach (string str in strWords)
        {
            Run run = new Run(str);
            run.MouseDown += RunOnMouseDown;
            text.Inlines.Add(run);
            text.Inlines.Add(" ");
        }
    }
}
```
void RunOnMouseDown(object sender, MouseButtonEventArgs args)
{
    Run run = sender as Run;

    if (args.ChangedButton == MouseButton.Left)

    if (args.ChangedButton == MouseButton.Right)
}

The constructor breaks up a famous quotation from Hamlet into words, and then creates a Run object based on each word and puts the words back together into the Inlines collection of the TextBlock object. During this process the program also attaches the RunOnMouseDown handler to the MouseDown event of each Run object.

The Run class inherits FontStyle and FontWeight properties from the TextElement class, and the event handler changes these properties based on which mouse button was clicked. For the left mouse button, if the FontStyle property is currently FontStyles.Italic, the event handler sets the property to FontStyles.Normal. If the property is currently FontStyles.Normal, the handler changes it to FontStyles.Italic. Similarly, the FontWeight property is toggled between FontWeight.Normal and FontWeight.Bold.

I mentioned earlier that the Content property of the window really wants an instance of a class that derives from UIElement, because that class defines a method named OnRender that visually renders the object on the screen. The last program in this chapter is named RenderTheGraphic and has two source code files. The first file is a class that defines a custom element. The second file sets an instance of that class as its window content. The following class derives from FrameworkElement, which is the sole class that directly derives from UIElement. It overrides the crucial OnRender method to obtain a DrawingContext object that it uses to draw an ellipse with the DrawEllipse method. The class is a simple imitation of the Ellipse class found in the System.Windows.Shapes namespace.

SimpleEllipse.cs

// SimpleEllipse.cs (c) 2006 by Charles Petzold

using System;
using System.Windows;
using System.Windows.Media;

namespace Petzold.RenderTheGraphic
{
    class SimpleEllipse : FrameworkElement
    {
The `RenderSize` property is determined before `OnRender` is called based on possible `Width` and `Height` settings, and negotiations between this class and the container in which it will appear.

If you have earlier experience with Windows programming (and even if you don’t) you might assume that this method draws an ellipse directly on the screen. It does not. The `DrawEllipse` arguments are retained to render the ellipse on the screen at a later time. This “later time” may be right away, but only by retaining graphics from different sources and compositing them on the screen can the WPF achieve much of its graphical magic.

Here’s a program that creates an object of `SimpleEllipse` and sets its `Content` property to that object:

```csharp
namespace Petzold.RenderTheGraphic
{
    class RenderTheGraphic : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new ReplaceMainWindow());
        }
        public ReplaceMainWindow()
        {
            Title = "Render the Graphic";
            SimpleEllipse elips = new SimpleEllipse();
            Content = elips;
        }
    }
}
```
The ellipse fills the client area. Of course, you'll want to experiment with setting the `Width` and `Height` properties of `SimpleEllipse` and with setting the `HorizontalAlignment` and `VerticalAlignment` properties.

Although this chapter has made use of elements found in the `System.Windows.Controls` namespace, it hasn’t made use of any classes that derive from `Control` (except for `Window` itself, of course). Controls are designed to obtain input from users and put that input to work. You'll see how in the next chapter.