PART VII

Documents and Printing

CHAPTER 28

Documents

Using the WPF skills you've picked up so far, you can craft windows and pages that include a wide variety of elements. Displaying fixed text is easy—you simply need to add the TextBlock and Label elements to the mix.

However, using the Label and TextBlock elements isn't a good solution if you need to display large volumes of text (such as a newspaper article or detailed instructions for online help). Large amounts of text are particularly problematic if you want your text to fit in a resizable window in the best possible way. For example, if you pile a large swath of text into a TextBlock and stretch it to fit a wide window, you'll end up with long lines that are difficult to read. Similarly, if you combine text and pictures by using the ordinary TextBlock and Image elements, you'll find that they no longer line up correctly when the window changes size.

To deal with these issues, WPF includes a set of higher-level features that work with *documents*. These features allow you to display large amounts of content in a way that makes them easy to read regardless of the size of the containing window. For example, WPF can hyphenate words (if you have only a narrow space available) or place your text into multiple columns (if you have a wide space to work with).

In this chapter, you'll learn how to use *flow documents* to display content. You'll also learn how to let users edit flow document content with the RichTextBox control. After you've mastered flow documents, you'll take a quick look at XPS, Microsoft's technology for creating print-ready documents. Finally, you'll consider WPF's annotation feature, which allows users to add comments and other markers to documents and store them permanently.

Understanding Documents

WPF separates documents into two broad categories:

> *Fixed documents:* These are typeset, print-ready documents. The positioning of all content is fixed (for example, the way text is wrapped over multiple lines and hyphenated can't change). Although you might choose to read a fixed document on a computer monitor, fixed documents are intended for print output. Conceptually, they're equivalent to Adobe PDF files. WPF includes a single type of fixed document, which uses Microsoft's XPS (XML Paper Specification) standard.

Flow documents: These documents are designed for viewing on a computer. Like fixed documents, flow documents support rich layout. However, WPF can optimize a flow document based on the way you want to view it. WPF can lay out the content dynamically based on details such as the size of the view window, the display resolution, and so on. Conceptually, flow documents are used for many of the same reasons as HTML documents, but they have more-advanced text-layout features.

Although flow documents are obviously more important from an application-building point of view, fixed documents are important for documents that need to be printed without alteration (such as forms and publications).

WPF provides support for both types of documents by using different containers. The DocumentViewer allows you to show fixed documents in a WPF window. The FlowDocumentReader, FlowDocumentPageViewer, and FlowDocumentScrollViewer give you different ways to look at flow documents. All of these containers are read-only. However, WPF includes APIs for creating fixed documents programmatically, and you can use the RichTextBox to allow the user to edit flow content.

In this chapter, you'll spend most of your time exploring flow documents and the ways they can be used in a WPF application. Toward the end of this chapter, you'll take a look at fixed documents, which are more straightforward.

Flow Documents

In a flow document, the content adapts itself to fit the container. Flow content is ideal for onscreen viewing. In fact, it avoids many of the pitfalls of HTML.

Ordinary HTML content uses flow layout to fill the browser window. (This is the same way WPF organizes elements if you use a WrapPanel.) Although this approach is very flexible, it gives a good result for only a small range of window sizes. If you maximize a window on a high-resolution monitor (or, even worse, a wide-screen display), you'll end up with long lines that are extremely difficult to read. Figure 28-1 shows this problem with a portion of a web page from Wikipedia.

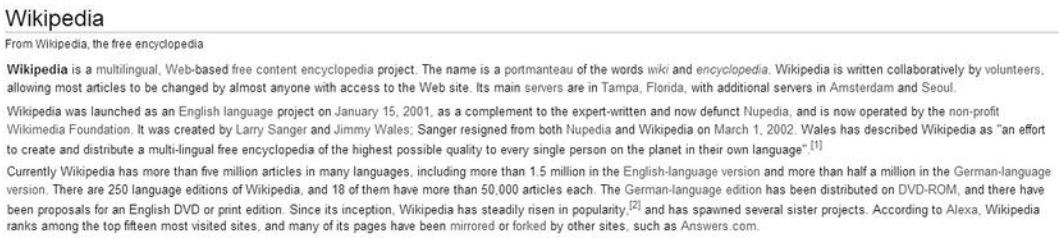

Figure 28-1. *Long lines in flow content*

Many websites avoid this problem by using some sort of fixed layout that forces content to fit a narrow column. (In WPF, you can create this sort of design by placing your content in a column inside a Grid container and setting the ColumnDefinition.MaxWidth property.) This prevents the readability problem, but it results in a fair bit of wasted screen space in large windows. Figure 28-2 shows this problem on a portion of a page from the New York Times website.

Figure 28-2. *Wasted space in flow content*

Flow document content in WPF improves upon these current-day approaches by incorporating better pagination, multicolumn display, sophisticated hyphenation and text flow algorithms, and user-adjustable viewing preferences. The end result is that WPF gives the user a much better experience when reading large amounts of content.

Understanding Flow Elements

You build a WPF flow document by using a combination of flow elements. Flow elements have an important difference from the elements you've seen so far. They don't inherit from the familiar UIElement and FrameworkElement classes. Instead, they form an entirely separate branch of classes that derive from ContentElement and FrameworkContentElement.

The content element classes are simpler than the noncontent element classes that you've seen throughout this book. However, content elements support a similar set of basic events, including events for keyboard and mouse handling, drag-and-drop operations, tooltip display, and initialization. The key difference between content and noncontent elements is that content elements do not handle their own rendering. Instead, they require a container that can render all its content elements. This deferred rendering allows the container to introduce various optimizations. For example, it allows the container to determine the best way to wrap lines of text in a paragraph, even though a paragraph is a single element.

■ **Note** Content elements can accept focus, but ordinarily they don't (because the Focusable property is set to false by default). You can make a content element focusable by setting Focusable to true on individual elements, by using an element type style that changes a whole group of elements, or by deriving your own custom element that sets Focusable to true. The Hyperlink is an example of a content element that sets its Focusable property to true.

Figure 28-3 shows the inheritance hierarchy of content elements.

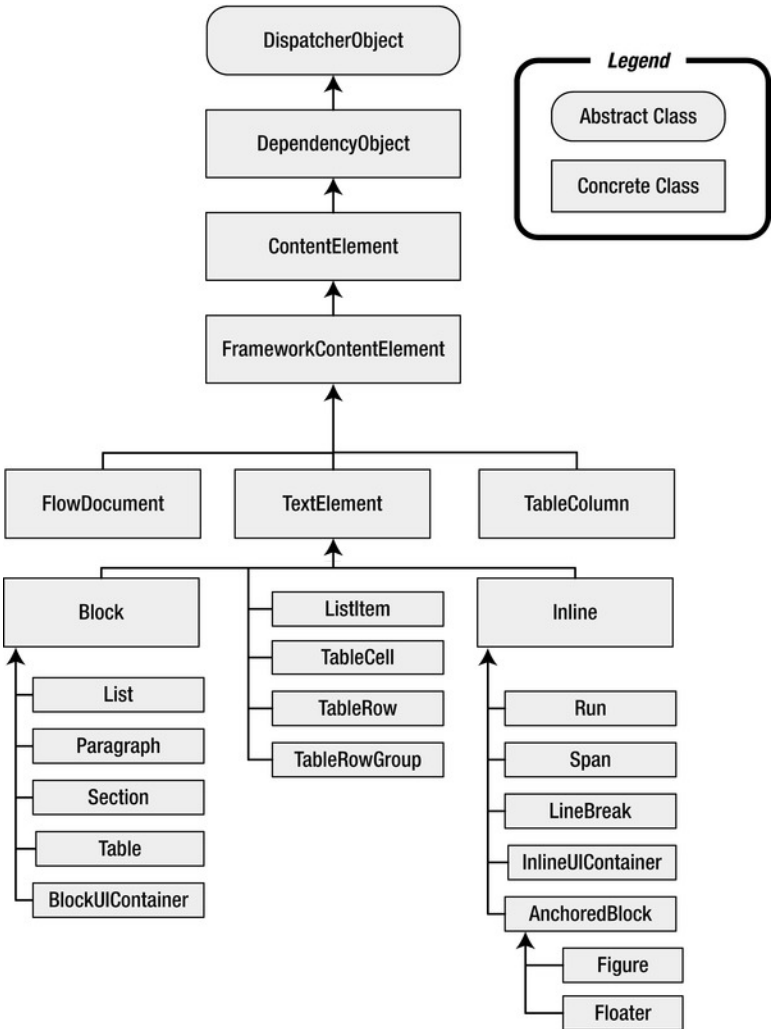

Figure 28-3. *Content elements*

There are two key branches of content elements:*Block elements*: These elements can be used to group other content elements. For example, a Paragraph is a block element. It can hold text that's formatted in various ways. Each section of separately formatted text is a distinct element in the paragraph.*Inline elements*: These elements are nested inside a block element (or another inline element). For example, the Run element wraps a bit of text, which can then be nested in a Paragraph element.

The content model allows multiple layers of nesting. For example, you can place a Bold element inside an Underline element to create text that's both bold and underlined. Similarly, you might create a Section element that wraps together multiple Paragraph elements, each of which contains a variety of inline elements with the actual text content. All of these elements are defined in the System.Windows.Documents namespace.

■ **Tip** If you're familiar with HTML, this model will seem more than a little familiar. WPF adopts many of the same conventions (such as the distinction between block and inline elements). If you're an HTML pro, you might consider using the surprisingly capable HTML-to-XAML translator at `http://tinyurl.com/mg9f6y`. With the help of this translator, which is implemented in C# code, you can use an HTML page as the starting point for a flow document.

Formatting Content Elements

Although the content elements don't share the same class hierarchy as noncontent elements, they feature many of the same formatting properties as ordinary elements. Table 28-1 lists some properties that you'll recognize from your work with noncontent elements.

Table 28-1. *Basic Formatting Properties for Content Elements*

Name	Description
Foreground and Background	Accept brushes that will be used to paint the foreground text and the background surface. You can also set the Background property on the FlowDocument object that contains all your markup.
FontFamily, FontSize, FontStretch, FontStyle, and FontWeight	Allow you to configure the font that's used to display text. You can also set these properties on the FlowDocument object that contains all your markup.
ToolTip	Allows you to set a tooltip that will appear when the user hovers over this element. You can use a string of text, or a full ToolTip object, as described in Chapter 6.
Style	Identifies the style that should be used to set the properties of an element automatically.

Block elements also add the properties shown in Table 28-2.

Table 28-2. *Additional Formatting Properties for Block Elements*

Name	Description
BorderBrush and BorderThickness	Allow you to create a border that will be shown around the edge of an element.
Margin	Sets the spacing between the current element and its container (or any adjacent elements). When the margin is not set, flow containers add a default space of about 18 units between block elements and the edges of the container. If you don't want this spacing, you can explicitly set smaller margins. However, to reduce the space between two paragraphs, you'll need to shrink both the bottom margin of the first paragraph and the top margin of the second paragraph. If you want all paragraphs to start out with reduced margins, consider using an element-type style rule that acts on all paragraphs.
Padding	Sets the spacing between its edges and any nested elements inside. The default padding is 0.
TextAlignment	Sets the horizontal alignment of nested text content (which can be Left, Right, Center, or Justify). Ordinarily, content is justified.

Name	Description
LineHeight	Sets the spacing between lines in the nested text content. Line height is specified as a number of device-independent pixels. If you don't supply this value, the text is single-spaced based on the characteristics of the font you're using.
LineStackingStrategy	Determines how lines are spaced if they contain mixed font sizes. The default option, MaxHeight, makes the line as tall as the largest text inside. The alternative, BlockLineHeight, uses the height configured in the LineHeight property for all lines, which means the text is spaced based on the font of the paragraph. If this font is smaller than the largest text in the paragraph, the text in some lines may overlap. If it's equal or larger, you'll get a consistent spacing that leaves extra whitespace between some lines.

Along with the properties described in these two tables, there are some additional details that you can tweak in specific elements. Some of these pertain to pagination and multicolumn displays and are discussed in the "Creating Pages and Columns" section later in this chapter. A few other properties of interest include the following:

- TextDecorations, which is provided by the Paragraph and all Inline-derived elements. It takes a value of strikethrough, overline, or (most commonly) underline. You can combine these values to draw multiple lines on a block of text, although it's not common.
- Typography, which is provided by the top-level FlowDocument element, as well as TextBlock and all TextElement-derived types. It provides a Typography object that you can use to alter a variety of details about the way text is rendered (most of which apply to only OpenType fonts).

Constructing a Simple Flow Document

Now that you've taken a look at the content element model, you're ready to assemble some content elements into a simple flow document.

You create a flow document by using the FlowDocument class. Visual Studio allows you to create a new flow document as a separate file, or you can define it inside an existing window by using one of the supported containers. For now, start building a simple flow document by using the FlowDocumentScrollViewer as a container. Here's how your markup should start:

```
<Window x:Class="Documents.FlowContent"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="FlowContent" Height="381" Width="525" >

  <FlowDocumentScrollViewer>
    <FlowDocument>
      ...
    </FlowDocument>
  </FlowDocumentScrollViewer>

</Window>
```

■ **Tip** Currently, there's no WYSIWYG interface for creating flow documents. Some developers are creating tools that can transform files written in Word 2007 XML (known as WordML) to XAML files with flow document markup. However, these tools aren't production ready. In the meantime, you can create a basic text editor by using a RichTextBox (as described in the "Editing a Flow Document" section later in this chapter) and use it to create flow document content.

You might assume that you could begin typing your text inside the FlowDocument element, but you can't. Instead, the top level of a flow document must use a block-level element. Here's an example with a Paragraph:

```
<FlowDocumentScrollViewer>
  <FlowDocument>
    <Paragraph>Hello, world of documents.</Paragraph>
  </FlowDocument>
</FlowDocumentScrollViewer>
```

There's no limit on the number of top-level elements you can use. So this example with two paragraphs is also acceptable:

```
<FlowDocumentScrollViewer>
  <FlowDocument>
    <Paragraph>Hello, world of documents.</Paragraph>
    <Paragraph>This is a second paragraph.</Paragraph>
  </FlowDocument>
</FlowDocumentScrollViewer>
```

Figure 28-4 shows the modest result.

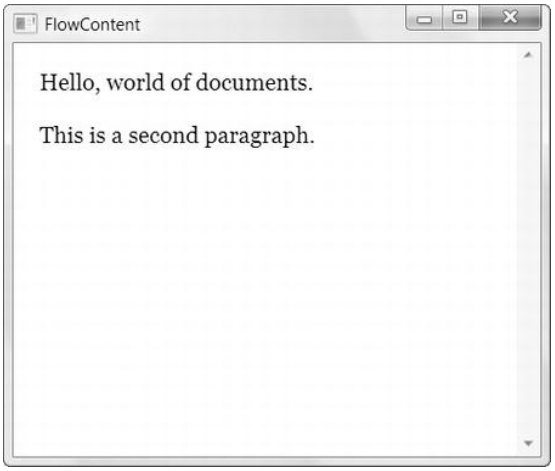

Figure 28-4. *A bare-bones flow document*

The scrollbar is added automatically. The font (Segoe UI) is picked up from the Windows system settings, not the containing window.

■ **Note** Ordinarily, the FlowDocumentScrollViewer allows text to be selected (as in a web browser). This way, a user can copy portions of a document to the Windows clipboard and paste them in other applications. If you don't want this behavior, set the FlowDocumentScrollViewer.IsSelectionEnabled property to false.

Using Block Elements

Creating a basic document is easy, but to get the result you really want, you need to master a range of elements. Among them are the five block elements described in the following sections.

Paragraph and Run

You've already seen the Paragraph element, which represents a paragraph of text. Technically, paragraph doesn't contain text—instead, it contains a collection of inline elements, which are stored in the Paragraph.Inlines collection.

This fact has two consequences. First, it means that a paragraph can contain a whole lot more than text. Second, it means that in order for a paragraph to contain text, the paragraph needs to contain an inline Run element. The Run element contains the actual text, as shown here:

```
<Paragraph>
  <Run>Hello, world of documents.</Run>
</Paragraph>
```

This long-winded syntax wasn't required in the previous example. That's because the Paragraph class is intelligent enough to create a Run implicitly when you place text directly inside.

However, in some cases it's important to understand the behind-the-scenes reality of how a paragraph works. For example, imagine you want to retrieve the text from a paragraph programmatically and you have the following markup:

```
<Paragraph Name="paragraph">Hello, world of documents.</Paragraph>
```

You'll quickly discover that the Paragraph class doesn't contain a Text property. In fact, there's no way to get the text from the paragraph. Instead, to retrieve the text (or change it), you need to grab the nested Run object, as shown here:

```
((Run)paragraph.Inlines.FirstInline).Text = "Hello again.";
```

You can improve the readability of this code by using a Span element to wrap the text you want to modify. You can then give the Span element a name and access it directly. The Span element is described in the "Using Inline Elements" section.

The Paragraph class includes a TextIndent property that allows you to set the amount that the first line should be indented. (By default, it's 0.) You supply a value in device-independent units.

The Paragraph class also includes a few properties that determine how it splits lines over column and page breaks. You'll consider these details in the "Creating Pages and Columns" section later in this chapter.

■ **Note** Unlike HTML, WPF doesn't have block elements for headings. Instead, you simply use paragraphs with different font sizes.

List

The List element represents a bulleted or numeric list. You choose by setting the MarkerStyle property. Table 28-3 lists your options. You can also set the distance between each list item and its marker by using the MarkerOffset property.

Table 28-3. *Values from the TextMarkerStyle Enumeration*

Name	Appears As . . .
Disc	A solid bullet. This is the default.
Box	A solid square box.
Circle	A bullet with no fill.
Square	A square box with no fill.
Decimal	An incrementing number (1, 2, 3). Ordinarily, it starts at 1, but you can adjust the StartingIndex to begin counting at a higher number. Despite the name, a MarkerStyle of Decimal will not show fractional values, just integral numbers.
LowerLatin	A lowercase letter that's incremented automatically (a, b, c).
UpperLatin	An uppercase letter that's incremented automatically (A, B, C).
LowerRoman	A lowercase Roman numeral that's incremented automatically (i, ii, iii, iv).
UpperRoman	An uppercase Roman numeral that's incremented automatically (I, II, III, IV).
None	Nothing.

You nest ListItem elements inside the List element to represent individual items in the list. However, each ListItem must itself include a suitable block element (such as a Paragraph). Here's an example that creates two lists, one with bullets and one with numbers:

```
<Paragraph>Top programming languages:</Paragraph>
<List>
  <ListItem>
    <Paragraph>C#</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>C++</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Perl</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Logo</Paragraph>
  </ListItem>
</List>

<Paragraph Margin="0,30,0,0">To-do list:</Paragraph>
<List MarkerStyle="Decimal">
  <ListItem>
    <Paragraph>Program a WPF application</Paragraph>
  </ListItem>
  <ListItem>
```

```
    <Paragraph>Bake bread</Paragraph>
  </ListItem>
</List>
```

Figure 28-5 shows the result.

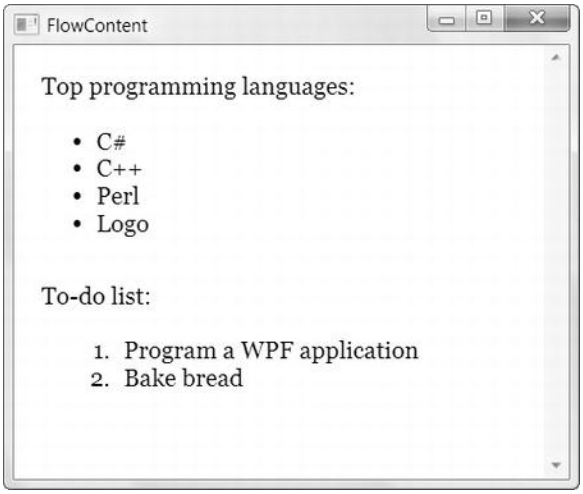

Figure 28-5. *Two lists*

Table

The Table element is designed to display tabular information. It's modeled after the HTML <table> element.

To create a table, you must follow these steps:

1. Place a TableRowGroup element inside the Table. The TableRowGroup holds a group of rows, and every table consists of one or more TableRowGroup elements. On its own, the TableRowGroup doesn't do anything. However, if you use multiple groups and give them each different formatting, you get an easy way to change the overall look of your table without setting repetitive formatting properties on each row.
2. Place a TableRow element inside your TableRowGroup for each row.
3. Place a TableCell element inside each TableRow to represent each column in the row.
4. Place a block element (typically a Paragraph) in each TableCell. This is where you'll add your content for that cell.

Here are the first two rows of the simple table shown in Figure 28-6:

```
<Paragraph FontSize="20pt">Largest Cities in the Year 100</Paragraph>
<Table>
  <TableRowGroup Paragraph.TextAlignment="Center">
    <TableRow FontWeight="Bold" >
```

```
      <TableCell>
        <Paragraph>Rank</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Name</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Population</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow>
      <TableCell>
        <Paragraph>1</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Rome</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>450,000</Paragraph>
      </TableCell>
    </TableRow>
    ...
  </TableRowGroup>
</Table>
```

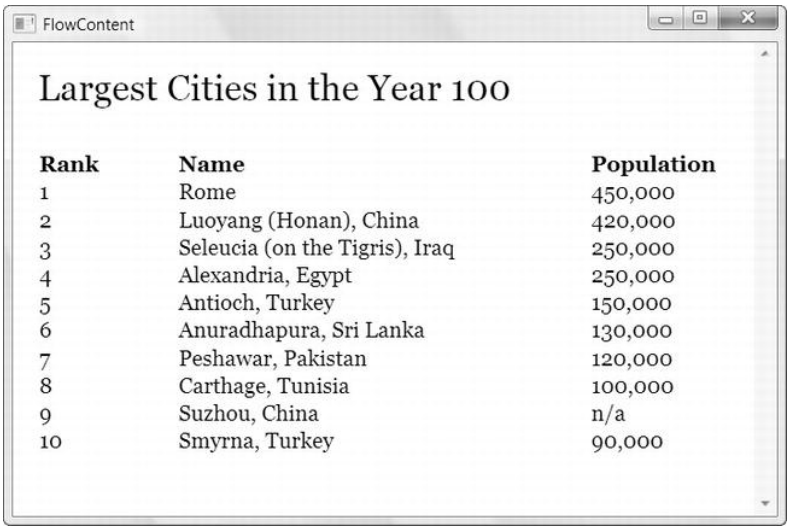

Figure 28-6. *A basic table*

■ **Note** Unlike a Grid, cells in a Table are filled by position. You must include a TableCell element for each cell in the table, and you must place each row and value in the correct display order.

If you don't supply explicit column widths, WPF splits the space evenly between all its columns. You can override this behavior by supplying a set of TableColumn objects for the Table.Rows property and setting the Width of each one. Here's the markup that the previous example uses to make the middle column three times as big as the first and last columns:

```
<Table.Columns>
  <TableColumn Width="*"></TableColumn>
  <TableColumn Width="3*"></TableColumn>
  <TableColumn Width="*"></TableColumn>
</Table.Columns>
```

There are a few more tricks you can perform with a table. You can set the ColumnSpan and RowSpan properties of a cell to make it stretch over multiple rows. You can also use the CellSpacing property of the table to set the number of units of space that are used to pad between cells. You can also apply individual formatting (such as text and background colors) to different cells. However, don't expect to find good support for table borders. You can use the BorderThickness and BorderBrush properties of the TableCell, but this forces you to draw a separate border around the edge of each cell with separate borders. These borders don't look quite right when you use them on a group of contiguous cells. Although the Table element provides the BorderThickness and BorderBrush properties, these allow you to draw a border around only the entire table. If you're hoping for a more sophisticated effect (for example, adding lines between columns), you're out of luck.

Another limitation is that columns must be sized explicitly or proportionately (using the asterisk syntax shown previously). However, you can't combine the two approaches. For example, there's no way to create two fixed-width columns and one proportional column to receive the leftover space, as you can with the Grid.

■ **Note** Some content elements are similar to other noncontent elements. However, the content elements are designed solely for use inside a flow document. For example, there's no reason to try to swap a Grid with a Table. The Grid is designed to be the most efficient option when laying out the controls in a window, while a Table is optimized to present text in the most readable way possible in a document.

Section

The Section element doesn't have any built-in formatting of its own. Instead, it's used to wrap other block elements in a convenient package. By grouping elements in a Section element, you can apply common formatting to an entire portion of a document. For example, if you want the same background color and font in several contiguous paragraphs, you can place these paragraphs in a section and then set the Section.Background property, as shown here:

```
<Section FontFamily="Palatino" Background="LightYellow">
  <Paragraph>Lorem ipsum dolor sit amet... </Paragraph>
  <Paragraph>Ut enim ad minim veniam...</Paragraph>
  <Paragraph>Duis aute irure dolor in reprehenderit...</Paragraph>
</Section>
```

This works because the font settings are inherited by the contained paragraphs. The background value is not inherited, but because the background of every paragraph is transparent by default, the section background shows through.

Even better, you can set the Section.Style property to format your section by using a style:

```
<Section Style="IntroText">
```

The Section element is analogous to the element in HTML.

■ **Tip** Many flow documents use style extensively to categorize content formatting based on its type. For example, a book reviewing site might create separate styles for review titles, review text, emphasized pull quotes, and bylines. These styles could then define whatever formatting is appropriate.

BlockUIContainer

The BlockUIContainer allows you to place noncontent elements (classes that derive from UIElement) inside a document, where a block element would otherwise go. For example, you can use the BlockUIContainer to add buttons, check boxes, and even entire layout containers such as the StackPanel and Grid to a document. The only rule is that the BlockUIContainer is limited to a single child.

You might wonder why you would ever want to place controls inside a document. After all, isn't the best rule of thumb to use layout containers for user-interactive portions of your interface, and flow layout for length, read-only blocks of content? However, in real-world applications many types of documents need to provide some sort of user interaction (beyond what the Hyperlink content element provides). For example, if you're using the flow layout system to create online help pages, you might want to include a button that triggers an action.

Here's an example that places a button under a paragraph:

```
<Paragraph>
  You can configure the foof feature using the Foof Options dialog box.
</Paragraph>
<BlockUIContainer>
  <Button HorizontalAlignment="Left" Padding="5">Open Foof Options</Button>
</BlockUIContainer>
```

You can connect an event handler to the Button.Click event in the usual way.

■ **Tip** Mingling content elements and ordinary noncontent elements makes sense if you have a user-interactive document. For example, if you're creating a survey application that lets users fill out different surveys, it may make sense to take advantage of the advanced text layout provided by the flow document model, without sacrificing the user's ability to enter values and make choices via common controls.

Using Inline Elements

WPF provides a larger set of inline elements, which can be placed inside block elements or other inline elements. Most of the inline elements are quite straightforward. Table 28-4 lists your options.

Table 28-4. Inline Content Elements

Name	Description
Run	Contains ordinary text. Although you can apply formatting to a Run element, it's generally preferred to use a Span element instead. Run elements are often created implicitly (such as when you add text to a paragraph).
Span	Wraps any amount of other inline elements. Usually, you'll use a span to specifically format a piece of text. To do so, you wrap the Span element around a Run element and set the properties of the Span element. (For a shortcut, just place text inside the Span element, and the nested Run element will be created automatically.) Another reason to use a Span is to make it easy for your code to find and manipulate a specific piece of text. The Span element is analogous to the element in HTML.
Bold, Italic, and Underline	Apply bold, italic, and underline formatting. These elements derive from Span. Although you can use these tags, it usually makes more sense to wrap the text you want to format inside a Span element and then set the Span.Style property to point to a style that applies the formatting you want. That way, you have the flexibility to easily adjust the formatting characteristics later on, without altering the markup of your document.
Hyperlink	Represents a clickable link inside a flow document. In a window-based application, you can respond to the Click event to perform an action (for example, showing a different document). In a page-based application, you can use the NavigateUri property to let the user browse directly to another page (as explained in Chapter 24).
LineBreak	Adds a line break inside a block element. Before using a line break, consider whether it would be clearer to use increased Margin or Padding values to add whitespace between elements.
InlineUIContainer	Allows you to place noncontent elements (classes that derive from UIElement) where an inline element would otherwise go (for example, in a Paragraph element). The InlineUIContainer is similar to the BlockUIElement, but it's an inline element rather than a block element.
Floater and Figure	Allow you to embed a floating box of content that you can use to highlight important information, display a figure, or show related content (such as advertisements, links, code listings, and so on).

Preserving Whitespace

Ordinarily, whitespace in XML is collapsed. Because XAML is an XML-based language, it follows the same rules.

As a result, if you include a string of spaces in your content, that string is converted to single space. That means this markup

```
<Paragraph>hello      there</Paragraph>
```

is equivalent to this:

```
<Paragraph>hello there</Paragraph>
```

Spaces between content and tags are also collapsed. So this line of markup

```
<Paragraph>        Hello there</Paragraph>
```

becomes

```
<Paragraph>Hello there</Paragraph>
```

For the most part, this behavior makes sense. It allows you to indent your document markup by using line breaks and tabs where convenient, without altering the way that content is interpreted.

Tabs and line breaks are treated in the same way as spaces. They're collapsed to a single space when they appear inside your content, and ignored when they appear on the edges of your content. However, there's one exception to this rule. If you have a space before an inline element, WPF preserves that space. (And if you have several spaces, WPF collapses these spaces to a single space.) That means you can write markup like this:

```
<Paragraph>A common greeting is <Bold>hello</Bold>.</Paragraph>
```

Here, the space between the content "A common greeting is" and the nested Bold element is retained, which is what you want. However, if you rewrote the markup like this, you'd lose the space:

```
<Paragraph>A common greeting is<Bold> hello</Bold>.</Paragraph>
```

In this case, you'll see the text "A common greeting ishello" in your user interface.

In some situations, you might want to add space where it would ordinarily be ignored or include a series of spaces. You can do this by using the xml:space attribute with the value *preserve*, which is an XML convention that tells an XML parser to keep all the whitespace characters in nested content:

```
<Paragraph xml:space="preserve">This     text     is    spaced   out</Paragraph>
```

This seems like the perfect solution, but there are still a few headaches. Now that the XML parser is paying attention to whitespace, you can no longer use line breaks and tabs to indent your content for easier reading. In a long paragraph, this is a significant trade-off that makes the markup more difficult to understand. (Of course, this won't be an issue if you're using another tool to generate the markup for your flow document, in which case you really don't care what the serialized XAML looks like.)

Because you can use the xml:space attribute on any element, you can pay attention to whitespace more selectively. For example, the following markup preserves whitespace in the nested Run element only:

```
<Paragraph>
  <Run xml:space="preserve">This      text     </Run> is spaced out.
</Paragraph>
```

Floater

The Floater element gives you a way to set some content off from the main document. Essentially, this content is placed in a "box" that floats somewhere in your document. (Often, it's displayed off to one side.) Figure 28-7 shows an example with a single line of text.

Figure 28-7. *A floating pull quote*

To create this floater, you simply insert a Floater element somewhere inside another block element (such as a paragraph). The Floater itself can contain one or more block elements. Here's the markup used to create the example in Figure 28-7. (The ellipsis indicates omitted text.)

```
<Paragraph>
  It was a bright cold day in April, and the clocks were striking thirteen ...
</Paragraph>
<Paragraph>The hallway smelt of boiled cabbage and old rag mats.
  <Run xml:space="preserve"> </Run>
  <Floater Style="{StaticResource PullQuote}">
    <Paragraph>"The hallway smelt of boiled cabbage"</Paragraph>
  </Floater>
  At one end of it a coloured poster, too large for indoor display ...
</Paragraph>
```

Here's the style that this Floater uses:

```
<Style x:Key="PullQuote">
  <Setter Property="Paragraph.FontSize" Value="30"></Setter>
  <Setter Property="Paragraph.FontStyle" Value="Italic"></Setter>
  <Setter Property="Paragraph.Foreground" Value="Green"></Setter>
  <Setter Property="Paragraph.Padding" Value="5"></Setter>
  <Setter Property="Paragraph.Margin" Value="5,10,15,10"></Setter>
</Style>
```

Ordinarily, the flow document widens the floater so that all its content fits on one line or, if that's not possible, so that it takes the full width of one column in the document window. (In the current example, there's only one column, so the Floater takes the full width of the document window.)

If this isn't what you want, you can specify the width in device-independent units by using the Width property. You can also use the HorizontalAlignment property to indicate whether the floater is centered, placed on the left edge, or placed on the right edge of the line where the Floater element is placed. Here's how you can create the left-aligned floater shown in Figure 28-8:

```
<Floater Style="{StaticResource PullQuote}" Width="205" HorizontalAlignment="Left">
  <Paragraph>"The hallway smelt of boiled cabbage"</Paragraph>
</Floater>
```

The Floater will use the specified width, unless it stretches beyond the bounds of the document window (in which case the floater gets the full width of the window).

Figure 28-8. *A left-aligned floater*

By default, the floating box that's used for the Floater is invisible. However, you can set a shaded background (through the Background property) or a border (through the BorderBrush and BorderThickness properties) to clearly separate this content from the rest of your document. You can also use the Margin property to add space between the floating box and the document, and the Padding property to add space between the edges of the box and its contents.

■ **Note** Ordinarily, the Background, BorderBrush, BorderThickness, Margin, and Padding properties are available only to block elements. However, they're also defined in the Floater and Figure classes, which are inline elements.

You can also use a floater to show a picture. But oddly enough, there is no flow content element that's up to the task. Instead, you'll need to use the Image element in conjunction with the BlockUIContainer or the InlineUIContainer.

However, there's a catch. When inserting a floater that wraps an image, the flow document assumes the figure should be as wide as a full column of text. The Image inside will then stretch to fit, which could result in problems if you're displaying a bitmap and it has to be scaled up or down a large amount. You could change the Image.Stretch property to disable this image-resizing feature, but in that case the floater will still take the full width of the column—it simply leaves extra blank space at the sides of the figure.

The only reasonable solution when embedding a bitmap in a flow document is to set a fixed size for the floater box. You can then choose how the image sizes itself in that box by using the Image.Stretch property. Here's an example:

```
<Paragraph>
  It was a bright cold day in April,
  <Floater Width="100" Padding="5,0,5,0" HorizontalAlignment="Right">
    <BlockUIContainer>
      <Image Source="BigBrother.jpg"></Image>
    </BlockUIContainer>
  </Floater>
  and the clocks ...
</Paragraph>
```

Figure 28-9 shows the result. Notice that the image actually stretches out over two paragraphs, but this doesn't pose a problem. The flow document wraps the text around all the floaters.

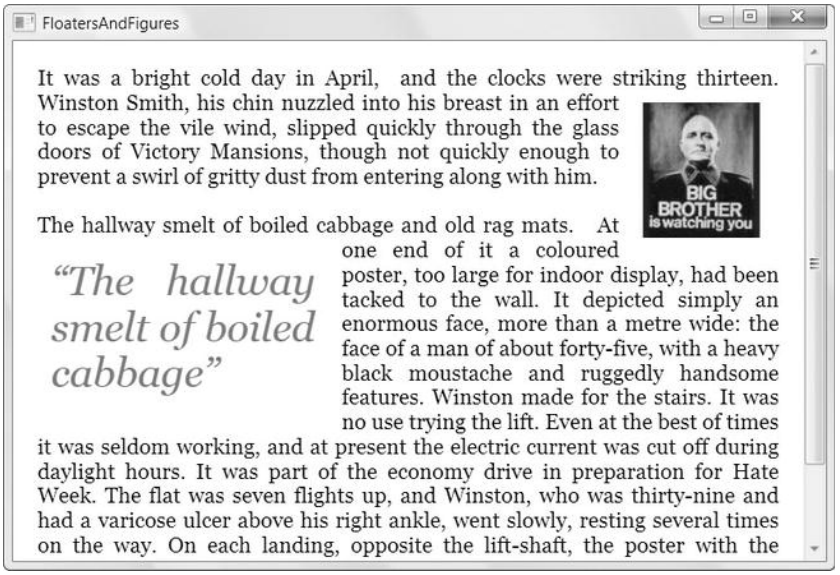

Figure 28-9. *A floater with an image*

■ **Note** Using a fixed-size floater also gives the most sensible result when you use zooming. As the zoom percentage changes, so does the size of your floater. The image inside the floater can then stretch itself as needed (based on the Image.Stretch property) to fill or center itself in the floater box.

Figure

The Figure element is similar to the Floater element, but it gives a bit more control over positioning. Usually, you'll use floaters and give WPF a little more control to arrange your content. But if you have a complex, rich document, you might prefer to use figures to make sure your floating boxes aren't bumped too far away as the window is resized, or to put boxes in specific positions.

So what does the Figure class offer that the Floater doesn't? Table 28-5 describes the properties you have to play with. However, there's one caveat: many of these properties (including HorizontalAnchor, VerticalOffset, and HorizontalOffset) aren't supported by the FlowDocumentScrollViewer that you've been using to display your flow document. Instead, they need one of the more sophisticated containers you'll learn about later in the "Using Read-Only Flow Document Containers" section. For now, replace the FlowDocumentScrollViewer tags with tags for the FlowDocumentReader if you want to use the figure placement properties.

Table 28-5. *Figure Properties*

Name	Description
Width	Sets the width of the figure. You can size a figure just as you size a floater, using device-independent pixels. However, you have the additional ability of sizing the figure proportionately, respective to the overall window or the current column. For example, in your XAML, you can supply the text *0.25 content* to create a box that takes 25 percent of the width of the window, or *2 Column* to create a box that's two columns wide.
Height	Sets the height of the figure. You can also set the exact height of a figure in device-independent units. (By comparison, a floater makes itself as tall as required to fit all its content in the specified width.) If your use of the Width and Height properties creates a floating box that's too small for all of its content, some content will be truncated.
HorizontalAnchor	Replaces the HorizontalAlignment property in the Floater class. However, along with three equivalent options (ContentLeft, ContentRight, and ContentCenter), it also includes options that allow you to orient the figure relative to the current page (such as PageCenter) or column (such as ColumnCenter).
VerticalAnchor	Allows you to align the image vertically with respect to the current line of text, the current column, or the current page.
HorizontalOffset and VerticalOffset	Set the figure alignment. These properties allow you to move the figure from its anchored position. For example, a negative VerticalOffset will shift the figure box up the number of units you specify. If you use this technique to move a figure away from the edge of the containing window, text will flow into the space you free up. (If you want to increase spacing on one side of a figure but you don't want text to enter that area, adjust the Figure.Padding property instead.)
WrapDirection	Determines whether text is allowed to wrap on one side or both sides (space permitting) of a figure.

Interacting with Elements Programmatically

So far, you've seen examples of how to create the markup required for flow documents. It should come as no surprise that flow documents can also be constructed programmatically. (After all, that's what the XAML parser does when it reads your flow document markup.)

Creating a flow document programmatically is fairly tedious because of a number of disparate elements that need to be created. As with all XAML elements, you must create each element and then set all its properties, as there are no constructors to help you out. You also need to create a Run element to wrap every piece of text, as it won't be generated automatically.

Here's a snippet of code that creates a document with a single paragraph and some bolded text. It then displays the document in an existing FlowDocumentScrollViewer named docViewer:

```
// Create the first part of the sentence.
Run runFirst = new Run();
runFirst.Text = "Hello world of ";

// Create bolded text.
Bold bold = new Bold();
Run runBold = new Run();
runBold.Text = "dynamically generated";
bold.Inlines.Add(runBold);

// Create last part of sentence.
Run runLast = new Run();
runLast.Text = " documents";

// Add three parts of sentence to a paragraph, in order.
Paragraph paragraph = new Paragraph();
paragraph.Inlines.Add(runFirst);
paragraph.Inlines.Add(bold);
paragraph.Inlines.Add(runLast);

// Create a document and add this paragraph.
FlowDocument document = new FlowDocument();
document.Blocks.Add(paragraph);

// Show the document.
docViewer.Document = document;
```

The result is the sentence "Hello world of **dynamically generated** documents."

Most of the time, you won't create flow documents programmatically. However, you might want to create an application that browses through portions of a flow document and modifies them dynamically. You can do this in the same way that you interact with any other WPF elements: by responding to element events, and by attaching a name to the elements that you want to change. However, because flow documents use deeply nested content with a free-flowing structure, you may need to dig through several layers to find the actual content you want to modify. (Remember, this content is always stored in a Run element, even if the run isn't declared explicitly.)

There are some properties that can help you navigate the structure of a flow document:

- To get the block elements in a flow document, use the FlowDocument.Blocks collection. Use FlowDocument.Blocks.FirstBlock or FlowDocument.Blocks.LastBlock to jump to the first or last block element.

- To move from one block element to the next (or previous) block, use the Block.NextBlock property (or Block.PreviousBlock). You can also use the Block.SiblingBlocks collection to browse all the block elements that are at the same level.
- Many block elements can contain other elements. For example, the List element provides a ListItem collection, the Section provides a Blocks collection, and the Paragraph provides an Inlines collection.

If you need to modify the text inside a flow document, the easiest way is to isolate exactly what you want to change (and no more) by using a Span element. For example, the following flow document highlights selected nouns, verbs, and adverbs in a block of text so they can be modified programmatically. The type of selection is indicated with an extra bit of information—a string that's stored in the Span.Tag property.

■ **Tip** Remember, the Tag property in any element is reserved for your use. It can store any value or object that you want to use later on.

```
<FlowDocument Name="document">
  <Paragraph FontSize="20" FontWeight="Bold">
    Release Notes
  </Paragraph>
  <Paragraph>
    These are the release <Span Tag="Plural Noun">notes</Span>
    for <Span Tag="Proper Noun">Linux</Span> version 1.2.13.
  </Paragraph>
  <Paragraph>
    Read them <Span Tag="Adverb">carefully</Span>, as they
    tell you what this is all about, how to <Span Tag="Verb">boot</Span>
    the <Span Tag="Noun">kernel</Span>, and what to do if
    something goes wrong.
  </Paragraph>
</FlowDocument>
```

This design allows you to create the straightforward Mad Libs game shown in Figure 28-10. In this game, the user gets the chance to supply values for all the span tags before seeing the source document. These user-supplied values are then substituted for the original values to humorous effect.

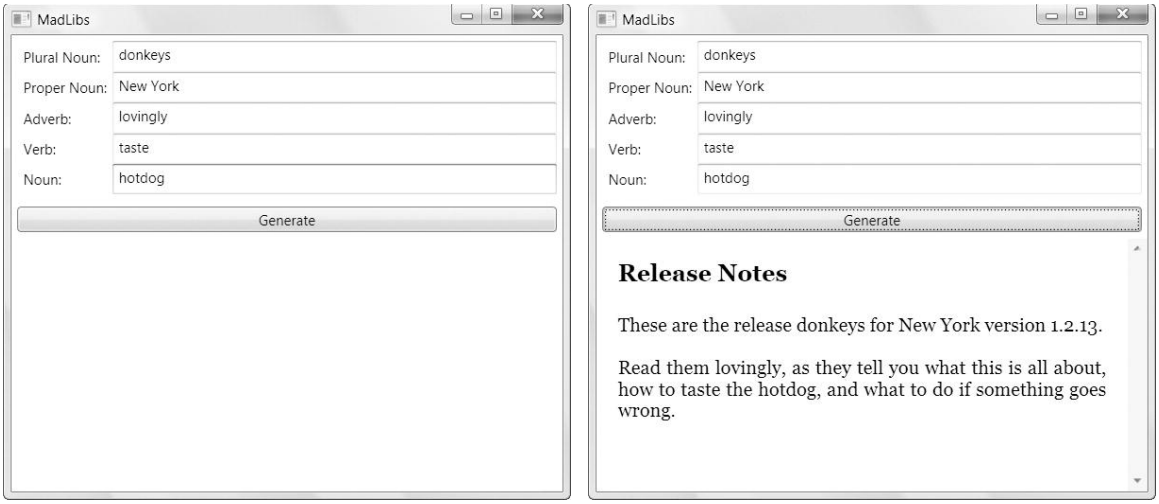

Figure 28-10. *Dynamically modifying a flow document*

To make this example as generic as possible, the code doesn't have any specific knowledge about the document that you're using. Instead, it's written generically so that it can pull the named Span elements out of all the top-level paragraphs in any document. It simply walks through the Blocks collection looking for paragraphs and then walks through the Inlines collection of each paragraph looking for spans. Each time it finds a Span object, it creates the text box that the user can use to supply a new value and adds it to a grid above the document (along with a descriptive label). And to make the substitution process easier, each text box stores a reference (through the TextBox.Tag property) to the Run element with the text inside the corresponding Span element:

```
private void WindowLoaded(Object sender, RoutedEventArgs e)
{
    // Clear grid of text entry controls.
    gridWords.Children.Clear();

    // Look at paragraphs.
    foreach (Block block in document.Blocks)
    {
        Paragraph paragraph = block as Paragraph;

        // Look for spans.
        foreach (Inline inline in paragraph.Inlines)
        {
            Span span = inline as Span;
            if (span != null)
            {
                // Create a slot in the row for this term.
                RowDefinition row = new RowDefinition();
                gridWords.RowDefinitions.Add(row);

                // Add the descriptive label for this term.
                Label lbl = new Label();
```

```
                lbl.Content = inline.Tag.ToString() + ":";
                Grid.SetColumn(lbl, 0);
                Grid.SetRow(lbl, gridWords.RowDefinitions.Count - 1);
                gridWords.Children.Add(lbl);

                // Add the text box where the user can supply a value for this term.
                TextBox txt = new TextBox();
                Grid.SetColumn(txt, 1);
                Grid.SetRow(txt, gridWords.RowDefinitions.Count - 1);
                gridWords.Children.Add(txt);

                // Link the text box to the run where the text should appear.
                txt.Tag = span.Inlines.FirstInline;
            }
        }
    }
}
```

When the user clicks the Generate button, the code walks through all the text boxes that were added dynamically in the previous step. It then copies the text from the text box to the related Run in the flow document:

```
private void cmdGenerate_Click(Object sender, RoutedEventArgs e)
{
    foreach (UIElement child in gridWords.Children)
    {
        if (Grid.GetColumn(child) == 1)
        {
            TextBox txt = (TextBox)child;
            if (txt.Text != "") ((Run)txt.Tag).Text = txt.Text;
        }
    }
    docViewer.Visibility = Visibility.Visible;
}
```

It might occur to you to do the reverse—in other words, walk through the document again, inserting the matching text each time you find a Span. However, this approach is more problematic because you can't enumerate through the collections of inline elements in a paragraph at the same time that you're modifying its content.

Text Justification

You may have already noticed that text content in a flow document is, by default, justified so that every line stretches from the left to the right margin. You can change this behavior by using the TextAlignment property, but most flow documents in WPF are justified.

To improve the readability of justified text, you can use a WPF feature called *optimal paragraph layout* that ensures that whitespace is distributed as evenly as possible. This avoids the distracting rivers of whitespace and oddly spaced-out words that can occur with more-primitive line-justification algorithms (such as those provided by web browsers).

■ **Note** Basic line-justification algorithms work on one line at a time. WPF's optimal paragraph justification uses a total-fit algorithm that looks ahead at the lines to come. It then chooses line breaks that balance the word spacing throughout the entire paragraph and result in the minimal cost over all lines.

Ordinarily, WPF's optimal paragraph feature isn't enabled. Presumably, this is because of the additional overhead in the total-fit algorithm. However, in most cases you'll find that the responsiveness of your application (how it "feels" as you resize the window) is the same with optimal paragraphs enabled.

To enable optimal paragraphs, set the FlowDocument.IsOptimalParagraphEnabled property to true. Figure 28-11 compares the difference by placing a flow document that uses normal paragraphs on top, and one that uses the total-fit algorithm below.

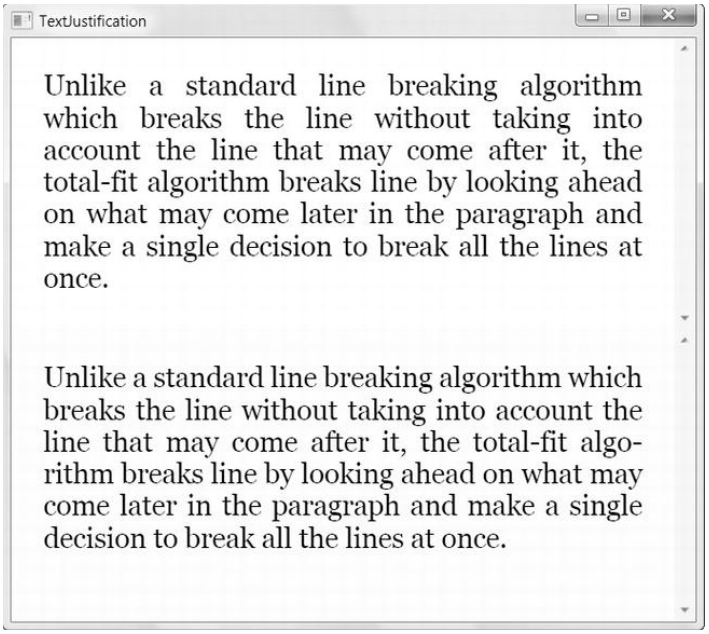

Figure 28-11. *Comparing ordinary justification (top) with optimal paragraphs (bottom)*

To further improve text justification, particularly in narrow windows, set the FlowDocument.IsHyphenationEnabled property to true. This way, WPF will break long words where necessary to keep the space between words small. Hyphenation works well with the optimal paragraph feature, and it's particularly important when using multicolumn displays. WPF uses a hyphenating dictionary to make sure that hyphens fall in the appropriate places (between syllables, as in *algo-rithm* rather than *algori-thm*).

Read-Only Flow Document Containers

WPF provides three read-only containers that you can use to display flow documents:

- FlowDocumentScrollViewer shows the entire document with a scrollbar to let you move through it if the document exceeds the size of the FlowDocumentScrollViewer. The FlowDocumentScrollViewer doesn't support pagination or multicolumn displays (although it does support printing and zooming, as all containers do). All of the examples you've seen up to this point have used the FlowDocumentScrollViewer.
- FlowDocumentPageViewer splits a flow document into multiple pages. Each page is as large as the available space, and the user can step from one page to the next. The FlowDocumentPageViewer has more overhead than the FlowDocumentScrollViewer (because of the additional calculations required for breaking content into pages).
- FlowDocumentReader combines the features of the FlowDocumentScrollViewer and FlowDocumentPageViewer. It lets the user choose whether to read content in a scrollable or paginated display. It also includes searching functionality. The FlowDocumentReader has the most overhead of any flow document container.

Switching from one container to another is simply a matter of modifying the containing tag. For example, here's a flow document in a FlowDocumentPageViewer:

```
<FlowDocumentPageViewer>
  <FlowDocument>
    <Paragraph>Hello, world of documents.</Paragraph>
  </FlowDocument>
</FlowDocumentPageViewer>
```

Each of these containers provides additional features, such as zooming, pagination, and printing. You'll learn about them in the following sections.

THE TEXTBLOCK

You can display small amounts of flow content by using the familiar TextBlock, a text display element that you've seen extensively over the past chapters. Although the TextBlock is often used to hold ordinary text (in which case the TextBlock creates a Run object to wrap that text), you can place any combination of inline elements inside. They'll all be added to the TextBlock.Inlines collection.

The TextBlock provides text wrapping (through the TextWrapping property), and a TextTrimming property that allows you to control how text is treated when it can't fit in the bounds of the TextBlock. When this occurs, the extra text is trimmed off, but you can choose whether an ellipsis is used to indicate that trimming has taken place. Your options are the following:

The TextBlock can't match the scrolling and paging features of the more sophisticated FlowDocument containers. For that reason, the TextBlock is best for displaying small amounts of flow content, such as control labels and hyperlinks. The TextBlock can't accommodate block elements at all.

Zooming

All three document containers support *zooming*: the ability for you to shrink or magnify the displayed content. The Zoom property of the container (for example, FlowDocumentScrollViewer.Zoom) sets the size of the content as a percentage value. Ordinarily, the Zoom value begins at 100, and the FontSize values correspond to any other elements in your window. If you increase the Zoom value to 200, the text size is doubled. Similarly, if you reduce it to 50, the text size is halved (although you can use any value in between).

Obviously, you can set the zoom percentage by hand. You can also change the zoom programmatically by using the IncreaseZoom() and DecreaseZoom() methods, which change the Zoom value by the amount specified by the ZoomIncrement property. You can also wire up other controls to these features by using commands (Chapter 9). But there's no need to go to any of this trouble. The simplest approach is to let users set the zoom percentage to match their preferences. The FlowDocumentScrollViewer includes a toolbar with a zoom slider bar for just this purpose. To make it visible, set IsToolbarVisible to true, as shown here:

```
<FlowDocumentScrollViewer MinZoom="50" MaxZoom="1000"
  Zoom="100" ZoomIncrement="5" IsToolbarVisible="True">
```

Figure 28-12 shows a flow document with a zoom slider bar at the bottom.

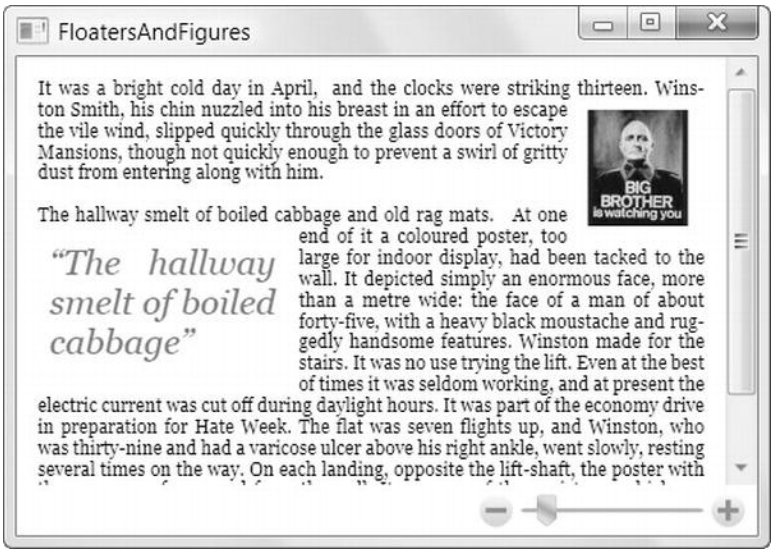

Figure 28-12. *Scaling down a document*

If you're using the FlowDocumentPageViewer or FlowDocumentReader, the zoom slider is always visible (although you can still configure the zoom increment and the minimum and maximum allowed zoom values).

■ **Tip** Zooming affects the size of anything that's set in device-independent units (not just font sizes). For example, if your flow document uses floater or figure boxes with explicit widths, these widths are also sized proportionately.

Creating Pages and Columns

The FlowDocumentPageViewer can split a long document into separate pages. This makes it easier to read long content. (When scrolling, readers are constantly forced to stop reading, scroll down, and then find the point where they left off. But when readers browse through a series of pages, they know exactly where to start reading—at the top of each page.)

The number of pages depends on the size of the window. For example, if you allow a FlowDocumentPageViewer to take the full size of a window, you'll notice that the number of pages changes as you resize the window, as shown in Figure 28-13.

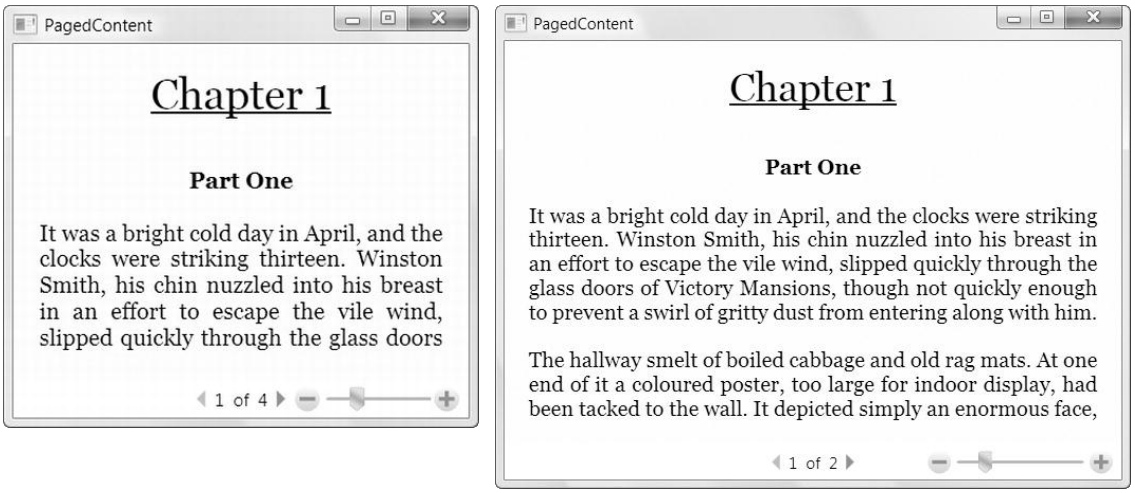

Figure 28-13. *Dynamically repaginated content*

If you make the window wide enough, the FlowDocumentPageViewer splits the text into multiple columns to make it easier to read (Figure 28-14). Figure 28-13 and Figure 28-14 show the same window. This window simply adjusts itself to make the best use of the available space.

■ **Note** Remember, Floater elements like to make themselves as wide as a single column. You can make them smaller by setting an explicit width, but not wider. On the other hand, Figure elements can easily span multiple columns.

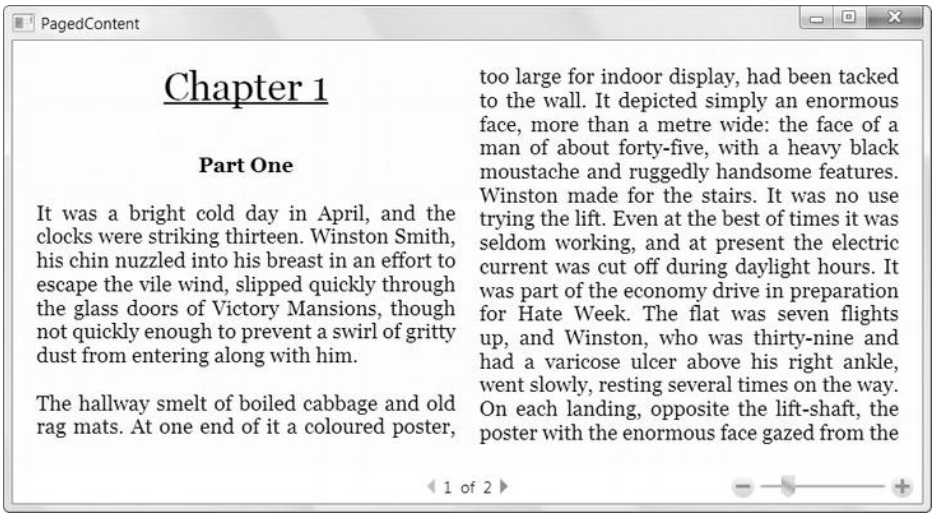

Figure 28-14. *Automatic columns*

Although the standard settings give good page breaking and column breaking, you can tweak them in a number of ways to get exactly the result you want. There are two key extensibility points that you can use: the FlowDocument class that contains the content (which provides the properties listed in Table 28-6) and individual Paragraph elements in the document (which provide the properties listed in Table 28-7).

Table 28-6. *FlowDocument Properties for Controlling Columns*

Name	Description
ColumnWidth	Specifies the preferred size of text columns. This acts as a minimum size, and the FlowDocumentPageViewer adjusts the width to make sure all the space is used on the page.
IsColumnWidthFlexible	Determines whether the document container can adjust the column size. If false, the exact column width specified by the ColumnWidth property is used. The FlowDocumentPageViewer will not create partial columns, so this may leave some blank space at the right edge of the page (or on either side if FlowDocumentMaxPageWidth is less than the width of the document window). If true (the default), the FlowDocumentPageViewer splits the space evenly to create columns, respecting the ColumnWidth property as a minimum.
ColumnGap	Sets the blank space between columns.
ColumnRuleWidth and ColumnRuleBrush	Allow you to draw a vertical line between columns. You can choose the width and fill of that line.

Table 28-7. *Paragraph Properties for Controlling Columns*

Name	Description
KeepTogether	Determines whether a paragraph can be split over a page break. If true, this paragraph will not be split over a page break. Usually, it will all be bumped to the next page. (This setting makes sense for small amounts of text that need to be read in one piece.)
KeepWithNext	Determines whether a pair of paragraphs can be separated by a page break. If true, this paragraph will not be divided from the following paragraph over a page break. (This setting makes sense for headings.)
MinOrphanLines	Controls how a paragraph can be split over a page break. When this paragraph is split over a page break, this is the minimum number of lines that needs to appear on the first page. If there isn't enough space for this number of lines, the entire paragraph will be bumped to the next page.
MinWindowLines	Controls how a paragraph can be split over a page break. When this paragraph is split over a page break, this is the minimum number of lines that needs to appear on the second page. The FlowDocumentPageViewer will move lines from the first page to the second to meet this criteria.

■ **Note** Obviously, in some situations the column-break properties of the Paragraph element can't be met. For example, if a paragraph is too large to fit on a single page, it doesn't matter whether you set KeepTogether to true, as the paragraph must be broken.

The FlowDocumentPageViewer isn't the only container that supports pagination. The FlowDocumentReader allows the user to choose between a scroll mode (which works exactly like the FlowDocumentScrollViewer) and two page modes. You can choose to see one page at a time (which works exactly like the FlowDocumentPageViewer), or two pages side by side. To switch between viewing modes, you simply click one of the icons in the bottom-right corner of the FlowDocumentReader toolbar.

Loading Documents from a File

So far, the examples you've seen declare the FlowDocument inside its container. However, it's no stretch to imagine that after you've created the perfect document viewer, you might want to reuse it to show different document content. (For example, you might show different topics in a help window.) To make this possible, you need to dynamically load content into the container by using the XamlReader class in the System.Windows.Markup namespace.

Fortunately, it's a fairly easy task. Here's the code you need (without the obligatory error-handling you'd use to catch file-access problems):

```
using (FileStream fs = File.Open(documentFile, FileMode.Open))
{
    FlowDocument document = XamlReader.Load(fs) as FlowDocument;

    if (document == null)
    {
        MessageBox.Show("Problem loading document.");
    }
```

```
    else
    {
        flowContainer.Document = document;
    }
}
```

It's just as easy to take the current content of a FlowDocument and save it to a XAML file by using the XamlWriter class. This functionality is less useful (after all, the containers you've seen so far don't allow the user to make changes). However, it's a worthwhile technique if you need to make programmatic changes to a document based on user actions (for example, you want to save the text from the completed Mad Libs game shown earlier), or you want to construct a FlowDocument programmatically and save it directly to disk.

Here's the code that serializes a FlowDocument object to XAML:

```
using (FileStream fs = File.Open(documentFile, FileMode.Create))
{
    XamlWriter.Save(flowContainer.Document, fs);
}
```

Printing

If you want to print a flow document, it's easy. Just use the Print() method of the container. (All flow document containers support printing.) The Print() method shows the Windows Print dialog box, where the user can choose the printer and other printing preferences, such as the number of copies, before choosing to cancel the operation or to go ahead and send the job to the printer.

Printing, like many of the features in the flow document containers, works through commands. As a result, if you want to wire a control up to this functionality, you don't need to write code that calls the Print() method. Instead, you can simply use the appropriate command, as shown here:

```
<Button Command="ApplicationCommands.Print" CommandTarget="docViewer">Print</Button>
```

Along with printing, the flow document containers also support commands for searching, zooming, and page navigation.

Commands may also have key bindings. For example, the Print command has a default key binding that maps the Ctrl+P keystroke. As a result, even if you don't include a button or code to call the Print() method, the user can still hit Ctrl+P to trigger it and show the Print window. If you don't want this behavior, you need to remove the key binding from the command.

■ **Note** It's possible to customize the printout of a flow document. You'll learn how to do this, and how to print other types of content, in Chapter 29.

Editing a Flow Document

All the flow document containers you've seen so far are read-only. They're ideal for displaying document content, but they don't allow the user to make changes. Fortunately, there's another WPF element that fills the gap: the RichTextBox control.

Programming toolkits have included rich text controls, in some form or another, for more than a decade. However, the RichTextBox control that WPF includes is significantly different from its

predecessors. It's no longer bound to the dated RTF standard that's found in word processing programs. Instead, it now stores its content as a FlowDocument object.

The consequences of this change are significant. Although you can still load RTF content into a RichTextBox control, internally the RichTextBox uses the much more straightforward flow content model that you've studied in this chapter. That makes it far easier to manipulate document content programmatically.

The RichTextBox control also exposes a rich programming model that provides plenty of extensibility points so you can plug in your own logic, which allows you to use the RichTextBox as a building block for your own customized text editor. The one drawback is speed. The WPF RichTextBox, like most of the rich text controls that have preceded it, can be a bit sluggish. If you need to hold huge amounts of data, use intricate logic to handle key presses, or add effects such as automatic formatting (for example, Visual Studio's syntax highlighting or Word's spelling-checker underlining), the WPF RichTextBox probably won't provide the performance you need.

■ **Note** The RichTextBox doesn't support all the features that read-only flow document containers do. Zooming, pagination, multicolumn displays, and search are all features that the RichTextBox doesn't provide.

Loading a File

To try out the RichTextBox, you can declare one of the flow documents you've already seen inside a RichTextBox element, as shown here:

```
<RichTextBox>
  <FlowDocument>
    <Paragraph>Hello, world of editable documents.</Paragraph>
  </FlowDocument>
</RichTextBox>
```

More practically, you may choose to retrieve a document from a file and then insert it in the RichTextBox. To do this, you can use the same approach that you used to load and save the content of a FlowDocument before displaying it in a read-only container—namely, the static XamlReader.Load() method. However, you might want the additional ability to load and save files in other formats (namely, .rtf files). To do this, you need to use the System.Windows.Documents.TextRange class, which wraps a chunk of text. The TextRange is a miraculously useful container that allows you to convert text from one format to another and apply formatting (as described in the next section).

Here's a simple code snippet that translates an .rtf document into a selection of text in a TextRange and then inserts it into a RichTextBox:

```
OpenFileDialog openFile = new OpenFileDialog();
openFile.Filter = "RichText Files (*.rtf)|*.rtf|All Files (*.*)|*.*";

if (openFile.ShowDialog() == true)
{
    TextRange documentTextRange = new TextRange(
      richTextBox.Document.ContentStart, richTextBox.Document.ContentEnd);

    using (FileStream fs = File.Open(openFile.FileName, FileMode.Open))
    {
```

```
            documentTextRange.Load(fs, DataFormats.Rtf);
        }
    }
```

Notice that before you can do anything, you need to create a TextRange that wraps the portion of the document you want to change. Even though there's currently no document content, you still need to specify the starting point and ending point of the selection. To select the whole document, you can use the FlowDocument.ContentStart and FlowDocument.ContentEnd properties, which provide the TextPointer objects the TextRange requires.

After the TextRange has been created, you can fill it with data by using the Load() method. However, you need to supply a string that identifies the type of data format you're attempting to convert. You can use one of the following:

- DataFormat.Xaml for XAML flow content
- DataFormats.Rtf for rich text (as in the previous example)
- DataFormats.XamlPackage for XAML flow content with embedded images
- DataFormats.Text for plain text

■ **Note** The DataFormats.XamlPackage format is essentially the same as DataFormats.Xaml. The only difference is that DataFormats.XamlPackage stores the binary data for any embedded images (which is left out if you use the ordinary DataFormats.Xaml serialization). The XAML package format is not a true standard—it's just a feature that WPF provides to make it easier to serialize document content and support other features you might want to implement, such as cut-and-paste or drag-and-drop.

Although the DataFormats class provides many additional fields, the rest aren't supported. For example, you won't have any luck attempting to convert an HTML document to flow content by using DataFormats.Html. Both the XAML package format and RTF require unmanaged code permission, which means you can't use them in a limited-trust scenario (such as a browser-based application).

The TextRange.Load() method works only if you specify the correct file format. However, it's quite possible that you might want to create a text editor that supports both XAML (for best fidelity) and RTF (for compatibility with other programs, such as word processors). In this situation, the standard approach is to let the user specify the file format or make an assumption about the format based on the file extension, as shown here:

```
using (FileStream fs = File.Open(openFile.FileName, FileMode.Open))
{
    if (Path.GetExtension(openFile.FileName).ToLower() == ".rtf")
    {
        documentTextRange.Load(fs, DataFormats.Rtf);
    }
    else
    {
        documentTextRange.Load(fs, DataFormats.Xaml);
    }
}
```

This code will encounter an exception if the file isn't found, can't be accessed, or can't be loaded using the format you specify. For all these reasons, you should wrap this code in an exception handler.

Remember, no matter how you load your document content, it's converted to a FlowDocument in order to be displayed by the RichTextBox. To study exactly what's taking place, you can write a simple routine that grabs the content from the FlowDocument and converts it to a string text by using the XamlWriter or a TextRange. Here's an example that displays the markup for the current flow document in another text box:

```
// Copy the document content to a MemoryStream.
using (MemoryStream stream = new MemoryStream())
{
    TextRange range = new TextRange(richTextBox.Document.ContentStart,
      richTextBox.Document.ContentEnd);
    range.Save(stream, DataFormats.Xaml);
    stream.Position = 0;

    // Read the content from the stream and display it in a text box.
    using (StreamReader r = new StreamReader(stream))
    {
        txtFlowDocumentMarkup.Text = r.ReadToEnd();
    }
}
```

This trick is extremely useful as a debugging tool for investigating how the markup for a document changes after it's been edited.

Saving a File

You can also save your document by using a TextRange object. You need to supply two TextPointer objects—one that identifies the start of the content, and one that demarcates the end. You can then call the TextRange.Save() method and specify the desired export format (text, XAML, XAML package, or RTF) by using a field from the DataFormats class. Once again, the XAML package and RTF formats require unmanaged code permission.

The following block of code saves the document in the XAML format unless the file name has an .rtf extension. (Another, more explicit approach is to give the user the choice of using a save feature that uses XAML and an export feature that uses RTF.)

```
SaveFileDialog saveFile = new SaveFileDialog();
saveFile.Filter =
  "XAML Files (*.xaml)|*.xaml|RichText Files (*.rtf)|*.rtf|All Files (*.*)|*.*";

if (saveFile.ShowDialog() == true)
{
    // Create a TextRange around the entire document.
    TextRange documentTextRange = new TextRange(
      richTextBox.Document.ContentStart, richTextBox.Document.ContentEnd);

    // If this file exists, it's overwritten.
    using (FileStream fs = File.Create(saveFile.FileName))
    {
        if (Path.GetExtension(saveFile.FileName).ToLower() == ".rtf")
        {
            documentTextRange.Save(fs, DataFormats.Rtf);
```

```
        }
        else
        {
            documentTextRange.Save(fs, DataFormats.Xaml);
        }
    }
}
```

When you use the XAML format to save a document, you probably assume that the document is stored as an ordinary XAML file with a top-level FlowDocument element. This is close, but not quite right. Instead, the top-level element must be a Section element.

As you learned earlier in this chapter, the Section is an all-purpose container that wraps other block elements. This makes sense—after all, the TextRange object represents a section of selected content. However, make sure that you don't try to use the TextRange.Load() method with other XAML files, including those that have a top-level FlowDocument, Page, or Window element, as none of these files will be parsed successfully. (Similarly, the document file can't link to a code-behind file or attach any event handlers.) If you have a XAML file that has a top-level FlowDocument element, you can create a corresponding FlowDocument object by using the XamlReader.Load() method, as you did with the other FlowDocument containers.

Formatting Selected Text

You can learn a fair bit about the RichTextBox control by building a simple rich text editor, like the one shown in Figure 28-15. Here, toolbar buttons allow the user to quickly apply bold formatting, italic formatting, and underlining. But the most interesting part of this example is the ordinary TextBox control underneath, which shows the XAML markup for the FlowDocument object that's currently displayed in the RichTextBox. This allows you to study how the RichTextBox modifies the FlowDocument object as you make edits.

■ **Note** Technically, you don't need to code the logic for bolding, italicizing, and underlining selected text. That's because the RichTextBox supports the ToggleBold, ToggleItalic, and ToggleUnderline commands from the EditingCommands class. You can wire your buttons up to these commands directly. However, it's still worth considering this example to learn more about how the RichTextBox works. The knowledge you gain is indispensable if you need to process text in another way. (The downloadable code for this chapter demonstrates both the code-based approach and the command-based approach.)

Figure 28-15. *Editing text*

All of the buttons work in a similar way. They use the RichTextBox.Selection property, which provides a TextSelection object that wraps the currently selected text. (TextSelection is a slightly more advanced class that derives from the TextRange class you saw in the previous section.)

Making changes with the TextSelection object is easy enough, but not obvious. The simplest approach is to use ApplyPropertyValue() to change a dependency property in the selection. For example, you could apply bold formatting to any text elements in the selection by using this code:

```
richTextBox.Selection.ApplyPropertyValue(
  TextElement.FontWeightProperty, FontWeights.Bold);
```

There's more happening here than meets the eye. For example, if you try this out on a small piece of text inside a larger paragraph, you'll find that this code automatically creates an inline Run element to wrap the selection and then applies the bold formatting to just that run. This way, you can use the same line of code to format individual words, entire paragraphs, and irregular selections that involve more than one paragraph (in which case you'll end up with a separate run being created in each affected paragraph).

Of course, this code as written isn't a complete solution. If you want to toggle the bold formatting, you'll also need to use the TextSelection.GetPropertyValue() to check whether bold formatting is already applied:

```
Object obj = richTextBox.Selection.GetPropertyValue(
  TextElement.FontWeightProperty);
```

This method is a little trickier. If your selection encloses text that is all unambiguously bold or unambiguously normal, you'll receive the FontWeights.Bold or FontWeights.Normal property. However, if your selection contains some bold text and some normal text, you'll get a DependencyProperty.UnsetValue instead.

It's up to you how you want to handle a mixed selection. You might want to do nothing, always apply the formatting, or decide based on the first character (which is what the EditingCommands.ToggleBold command does). To do this, you'd need to create a new TextRange that wraps just the starting point of the selection. Here's the code that implements the latter approach and checks the first letter in ambiguous cases:

```
Object obj = richTextBox.Selection.GetPropertyValue(
  TextElement.FontWeightProperty);

if (obj == DependencyProperty.UnsetValue)
{
    TextRange range = new TextRange(richTextBox.Selection.Start,
      richTextBox.Selection.Start);

    obj = range.GetPropertyValue(TextElement.FontWeightProperty);
}

FontWeight fontWeight = (FontWeight)obj;

if (fontWeight == FontWeights.Bold)
  fontWeight = FontWeights.Normal;
else
  fontWeight = FontWeights.Bold;

richTextBox.Selection.ApplyPropertyValue(
  TextElement.FontWeightProperty, fontWeight);
```

In some cases, a user might trigger the bold command without any selected text at all. Just for fun, here's a code routine that checks for this condition and then checks the formatting that's applied to the entire paragraph that contains this text. The font weight of that paragraph is then flipped from bold to normal or from normal to bold:

```
if (richTextBox.Selection.Text == "")
{
    FontWeight fontWeight = richTextBox.Selection.Start.Paragraph.FontWeight;
    if (fontWeight == FontWeights.Bold)
      fontWeight = FontWeights.Normal;
    else
      fontWeight = FontWeights.Bold;

    richTextBox.Selection.Start.Paragraph.FontWeight = fontWeight;
}
```

■ **Tip** To get the plain, unformatted text in a selection, use the TextRange.Text property.

There are many more methods for manipulating text in a RichTextBox. For example, the TextRange class and RichTextBox class both include a range of properties that let you get character offsets, count lines, and navigate through the flow elements in a portion of a document. To get more information, consult the MSDN help.

Getting Individual Words

One frill that the RichTextBox lacks is the ability to isolate specific words in a document. Although it's easy enough to find the flow document element that exists in a given position (as you saw in the previous section), the only way to grab the nearest word is to move character by character, checking for whitespace. This type of code is tedious and extremely difficult to write without error.

Prajakta Joshi of the WPF editing team has posted a reasonably complete solution at `http://tinyurl.com/ylbla4v` that detects word breaks. Using this code, you can quickly create a host of interesting effects, such as the following routine that grabs a word when the user right-clicks, and then displays that word in a separate text box. Another option might be to show a pop-up with a dictionary definition, launch an e-mail program or a web browser to follow a link, and so on:

```
private void richTextBox_MouseDown(object sender, MouseEventArgs e)
{
    if (e.RightButton == MouseButtonState.Pressed)
    {
        // Get the nearest TextPointer to the mouse position.
        TextPointer location = richTextBox.GetPositionFromPoint(
           Mouse.GetPosition(richTextBox), true);

        // Get the nearest word using this TextPointer.
        TextRange word = WordBreaker.GetWordRange(location);

        // Display the word.
        txtSelectedWord.Text = word.Text;
    }
}
```

■ **Note** This code doesn't actually connect to the MouseDown event, because the RichTextBox intercepts and suppresses MouseUp and MouseDown. Instead, this event handler is attached to the PreviewMouseDown event, which occurs just before MouseDown.

PLACING UIELEMENT OBJECTS IN A RICHTEXTBOX

As you learned earlier in this chapter, you can use the BlockUIContainer and InlineUIContainer classes to place noncontent elements (classes that derive from UIElement) inside a flow document. However, if you use this technique to add interactive controls (such as text boxes, buttons, check boxes, hyperlinks, and so on) to a RichTextBox, they'll be disabled automatically and will appear grayed out.

You can opt out of this behavior and force the RichTextBox to enable embedded controls, much like the read-only FlowDocument containers do. To do so, simply set the RichTextBox.IsDocumentEnabled property to true.

Although it's easy, you may want to think twice before you set IsDocumentEnabled to true. Including element content inside a RichTextBox introduces all sorts of odd usability quirks. For example, controls can be deleted and undeleted (using Ctrl+Z or the Undo command), but undeleting them loses their event handlers. Furthermore, text can be inserted between adjacent containers, but if you attempt to cut and paste a block of content that includes UIElement objects, they'll be discarded. For reasons like these, it's probably not worth the trouble to use embedded controls inside a RichTextBox.

Fixed Documents

Flow documents allow you to dynamically lay out complex, text-heavy content in a way that's naturally suited to onscreen reading. Fixed documents—those that use XPS (the XML Paper Specification)—are much less flexible. They serve as print-ready documents that can be distributed and printed on any output device with full fidelity to the original source. Toward that end, they use a precise, fixed layout, have support for font embedding, and can't be casually rearranged.

XPS isn't just a part of WPF. It's a standard that's tightly integrated into the Windows operating system. Windows includes a print driver that can create XPS documents (in any application) and a viewer that allows you to display them. These two pieces work similarly to Adobe Acrobat, allowing users to create, review, and annotate print-ready electronic documents. Additionally, Microsoft Office allows you to save your documents as XPS or PDF files.

■ **Note** Under the hood, XPS files are actually ZIP files that contain a library of compressed files, including fonts, images, and text content for individual pages (using a XAML-like XML markup). To browse the inner contents of an XPS file, just rename the extension to .zip and open it. You can also refer to `http://tinyurl.com/yg7jqjb` for an overview of the XPS file format.

You can display an XPS document just as easily as you display a flow document. The only difference is the viewer. Instead of using one of the FlowDocument containers (FlowDocumentReader, FlowDocumentScrollViewer, or FlowDocumentPageViewer), you use the simply named DocumentViewer. It includes controls for searching and zooming (Figure 28-16). It also provides a similar set of properties, methods, and commands as the FlowDocument containers.

Here's the code you might use to load an XPS file into memory and show it in a DocumentViewer:

```
XpsDocument doc = new XpsDocument("filename.xps", FileAccess.Read);
docViewer.Document = doc.GetFixedDocumentSequence();
doc.Close();
```

The XpsDocument class isn't terribly exciting. It provides the GetFixedDocument-Sequence() method used previously, which returns a reference to the document root with all its content. It also includes an AddFixedDocument() method for creating the document sequence in a new document, and two methods for managing digital signatures (SignDigitally() and RemoveSignature()).

XPS documents are closely associated with the concept of printing. A single XPS document is fixed at a particular page size and lays out its text to fit the available space. As with flow documents, you can get straightforward support for printing a fixed document by using the ApplicationCommands.Print command. In Chapter 29, you'll learn how to get fine-grained control of printing, and you'll see how the XPS model allows you to create a straightforward print preview feature.

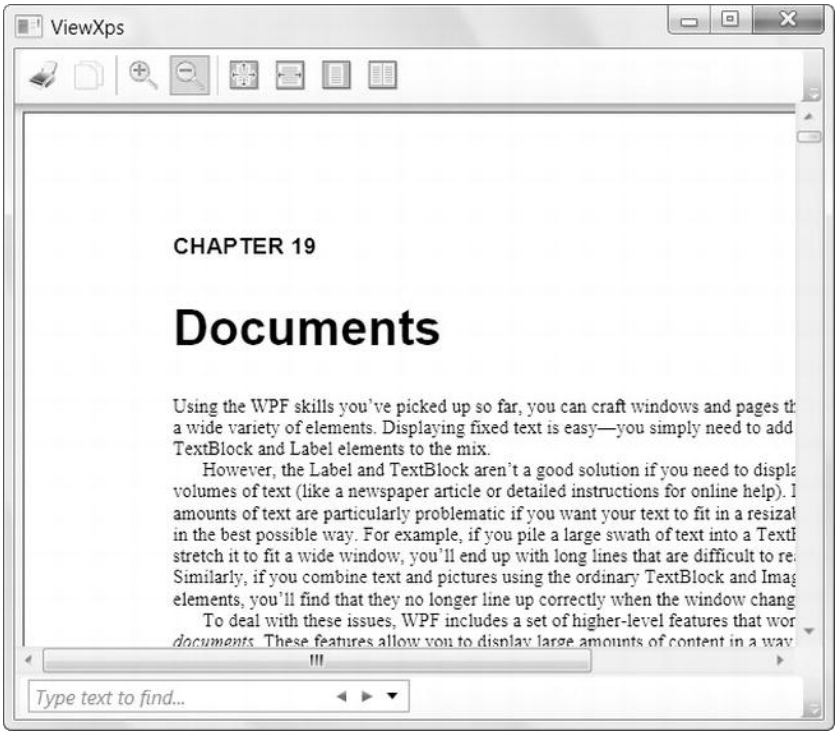

Figure 28-16. *A fixed document*

Annotations

WPF provides an annotation feature that allows you to add comments and highlights to flow documents and fixed documents. These annotations can be used to suggest revisions, highlight errors, or flag important pieces of information.

> Many products provide a wide range of annotation types. For example, Adobe Acrobat allows you to draw revision marks and shapes on a document. WPF isn't quite as flexible. It allows you to use two types of annotations: *Highlighting*: You can select some text and give it a colored background of your choice. (Technically, WPF highlighting applies a partially transparent color *over* your text, but the effect makes it seem as if you were changing the background.) *Sticky notes*: You can select some text and attach a floating box that contains additional text information or ink content.

Figure 28-17 shows the sample you'll learn how to build in this section. It shows a flow document with a highlighted text region and two sticky notes, one with ink content and one with text content.

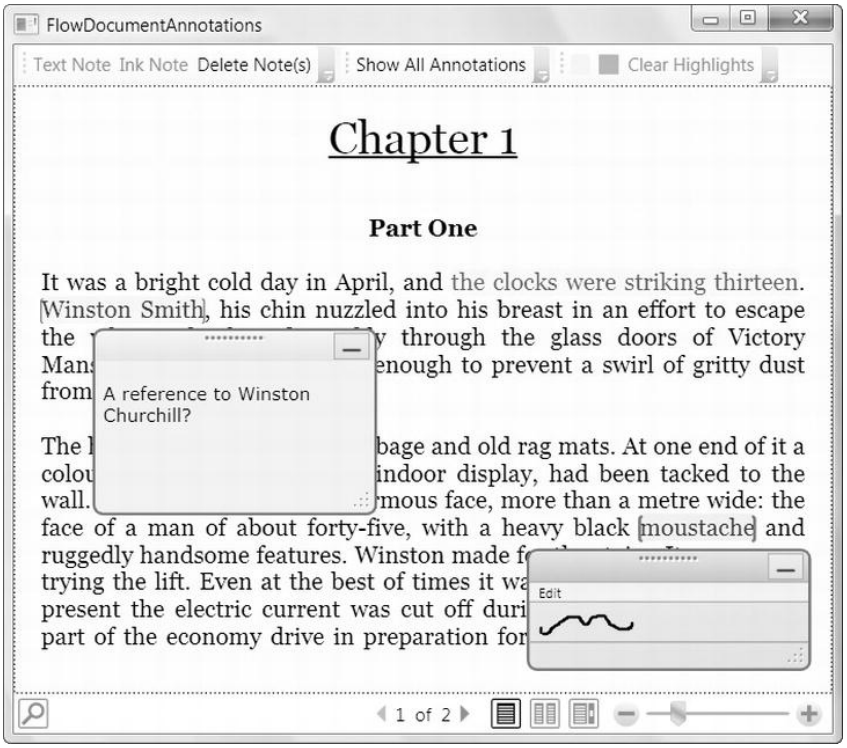

Figure 28-17. *Annotating a flow document*

All four of the WPF document containers—FlowDocumentReader, FlowDocumentScrollViewer, FlowDocumentPageViewer, and DocumentViewer—support annotations. But in order to use annotations, you need to take two steps. First, you need to manually enable the annotation service by using a bit of initialization code. Second, you need to add controls (such as toolbar buttons) that allow users to add the types of annotations you want to support.

Understanding the Annotation Classes

WPF's annotation system relies on several classes from the System.Windows.Annotations and System.Windows.Annotations.Storage namespaces. Here are the key players:

AnnotationService: This class manages the annotations feature. In order to use annotations, it's up to you to create this object.

AnnotationStore: This class manages the storage of your annotations. It defines several methods that you can use to create and delete individual annotations. It also includes events that you can use to react to annotations being created or changed. AnnotationStore is an abstract class, and there's currently just one class that derives from it: XmlStreamStore. XmlStreamStore serializes annotations to an XML-based format and allows you to store your annotation XML in any stream.

AnnotationHelper: This class provides a small set of static methods for dealing with annotations. These methods bridge the gap between the stored annotations and the document container. Most of the AnnotationHelper methods work with the currently selected text in the document container (allowing you to highlight it, annotate it, or remove its existing annotations). The AnnotationHelper also allows you to find where a specific annotation is placed in a document.

In the following sections, you'll use all three of these key ingredients.

■ **Tip** Both the AnnotationStore and the AnnotationHelper provide methods for creating and deleting annotations. However, the methods in the AnnotationStore class work with the currently selected text in a document container. For that reason, the AnnotationStore methods are best for programmatically manipulating annotations without user interaction, while the AnnotationHelper methods are best for implementing user-initiated annotation changes (for example, adding an annotation when the user selects some text and clicks a button).

Enabling the Annotation Service

Before you can do anything with annotations, you need to enable the annotation service with the help of an AnnotationService and AnnotationStream object.

In the example shown in Figure 28-17, it makes sense to create the AnnotationService when the window first loads. Creating the service is simple enough—you just need to create an AnnotationService object for the document reader and call AnnotationService.Enable(). However, when you call Enable(), you need to pass in an AnnotationStore object. The AnnotationService manages the information for your annotations, while the AnnotationStore manages the storage of these annotations.

Here's the code that creates and enables annotations:

```
// A stream for storing annotation.
private MemoryStream annotationStream;

// The service that manages annotations.
private AnnotationService service;

protected void window_Loaded(object sender, RoutedEventArgs e)
{
    // Create the AnnotationService for your document container.
    service = new AnnotationService(docReader);

    // Create the annotation storage.
    annotationStream = new MemoryStream();
    AnnotationStore store = new XmlStreamStore(annotationStream);

    // Enable annotations.
    service.Enable(store);
}
```

Notice that in this example, annotations are stored in a MemoryStream. As a result, they'll be discarded as soon as the MemoryStream is garbage collected. If you want to store annotations so they can be reapplied to the original document, you have two choices. You can create a FileStream instead of a

MemoryStream, which ensures that the annotation data is written as the user applies it. Or you can copy the data in the MemoryStream to another location (such as a file or a database record) after the document is closed.

■ **Tip** If you aren't sure whether annotations have been enabled for your document container, you can use the static AnnotionService.GetService() method and pass in a reference to the document container. This method returns a null reference if annotations haven't been enabled yet.

At some point, you'll also need to close your annotation stream and switch off the AnnotationService. In this example, these tasks are performed when the user closes the window:

```
protected void window_Unloaded(object sender, RoutedEventArgs e)
{
    if (service != null && service.IsEnabled)
    {
        // Flush annotations to stream.
        service.Store.Flush();

        // Disable annotations.
        service.Disable();
        annotationStream.Close();
    }
}
```

This is all you need to enable annotations in a document. If there are any annotations defined in the stream object when you call AnnotationService.Enable(), these annotations will appear immediately. However, you still need to add the controls that will allow the user to add or remove annotations. That's the topic of the next section.

■ **Tip** Every document container can have one instance of the AnnotationService. Every document should have its own instance of the AnnotationStore. When you open a new document, you should disable the AnnotationService, save and close the current annotation stream, create a new AnnotationStore, and then reenable the AnnotationService.

Creating Annotations

There are two ways to manipulate annotations. You can use one of the methods of the AnnotationHelper class that allows you to create annotations (CreateTextStickyNoteForSelection() and CreateInkStickyNoteForSelection()), delete them (DeleteTextStickyNotesForSelection() and DeleteInkStickyNotesForSelection()), and apply highlighting (CreateHighlightsForSelection() and ClearHighlightsForSelection()). The *ForSelection* part of the method name indicates that these methods apply the annotation to whatever text is currently selected.

Although the AnnotationHelper methods work perfectly well, it's far easier to use the corresponding commands that are exposed by the AnnotationService class. You can wire these commands directly to the buttons in your user interface. That's the approach we'll take in this example.

Before you can use the AnnotationService class in XAML, you need to map the System.Windows.Annotations namespace to an XML namespace, as it isn't one of the core WPF namespaces. You can add a mapping like this:

```
<Window x:Class="XpsAnnotations.FlowDocumentAnnotations"
  xmlns:annot=
  "clr-namespace:System.Windows.Annotations;assembly=PresentationFramework" ... >
```

Now you can create a button like this, which creates a text note for the currently selected portion of the document:

```
<Button Command="annot:AnnotationService.CreateTextStickyNoteCommand">
  Text Note
</Button>
```

Now when the user clicks this button, a green note window will appear. The user can type text inside this note. (If you create an ink sticky note with the CreateInkStickyNoteCommand, the user can draw inside the note window instead.)

■ **Note** This Button element doesn't set the CommandTarget property. That's because the button is placed in a toolbar. As you learned in Chapter 9, the Toolbar class is intelligent enough to automatically set the CommandTarget to the element that has focus. Of course, if you use the same command in a button outside a toolbar, you'll need to set the CommandTarget to point to your document viewer.

Sticky notes don't need to remain visible at all times. If you click the minimize button in the top-right corner of the note window, it will disappear. All you'll see is the highlighted portion of the document where the note is set. If you hover over this highlighted region with the mouse, a note icon appears (see Figure 28-18)—click this to restore the sticky note window. The AnnotationService stores the position of each note window. Therefore, if you drag one somewhere specific in your document, close it and then reopen it, the note window will reappear in its previous place.

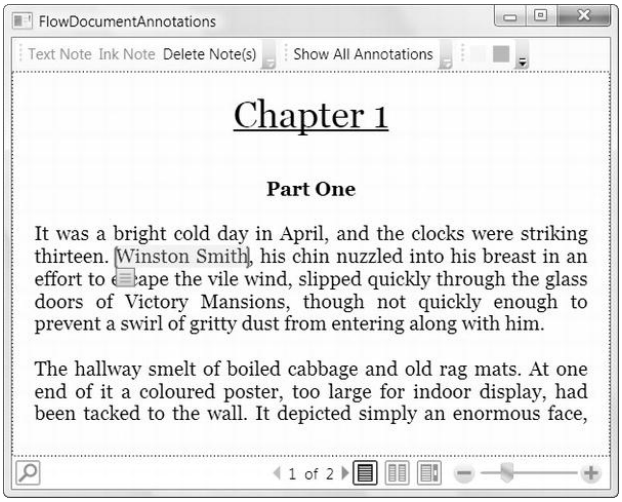

Figure 28-18. *A "hidden" annotation*

In the previous example, the annotation is created without any author information. If you plan to have multiple users annotating the same document, you'll almost certainly want to store some identifying information. Just pass a string that identifies the author as a parameter to the command, as shown here:

```
<Button Command="annot:AnnotationService.CreateTextStickyNoteCommand"
 CommandParameter="{StaticResource AuthorName}">
  Text Note
</Button>
```

This markup assumes that the author name is set as a resource:

```
<sys:String x:Key="AuthorName">[Anonymous]</sys:String>
```

This allows you to set the author name when the window first loads, at the same time as you initialize the annotation service. You can use a name that the user supplies, which you'll probably want to store in a user-specific .config file as an application setting. Alternatively, you can use the following code to grab the current user's Windows user account name with the help of the System.Security.Principal.WindowsIdentity class:

```
WindowsIdentity identity = WindowsIdentity.GetCurrent();
this.Resources["AuthorName"] = identity.Name;
```

To create the window shown in Figure 28-17, you'll also want to create buttons that use the CreateInkStickyNoteCommand (to create a note window that accepts hand-drawn ink content) and DeleteStickyNotesCommand (to remove previously created sticky notes):

```
<Button Command="annot:AnnotationService.CreateInkStickyNoteCommand"
 CommandParameter="{StaticResource AuthorName}">
  Ink Note
</Button>
<Button Command="annot:AnnotationService.DeleteStickyNotesCommand">
  Delete Note(s)
</Button>
```

The DeleteStickyNotesCommand removes all the sticky notes in the currently selected text. Even if you don't provide this command, the user can still remove annotations by using the Edit menu in the note window (unless you've given the note window a different control template that doesn't include this feature).

The final detail is to create the buttons that allow you to apply highlighting. To add a highlight, you use the CreateHighlightCommand and you pass the Brush object that you want to use as the CommandParameter. However, it's important to make sure you use a brush that has a partially transparent color. Otherwise, your highlighted content will be completely obscured, as shown in Figure 28-19.

For example, if you want to use the solid color #FF32CD32 (for lime green) to highlight your text, you should reduce the alpha value, which is stored as a hexadecimal number in the first two characters. (The alpha value ranges from 0 to 255, where 0 is fully transparent and 255 is fully opaque.) For example, the color #**54**FF32CD32 gives you a semitransparent version of the lime green color, with an alpha value of 84 (or 54 in hexadecimal notation).

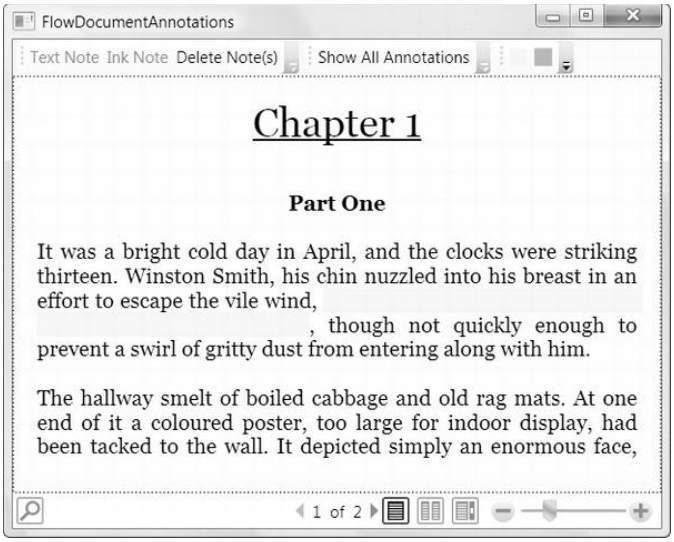

Figure 28-19. *Highlighting content with a nontransparent color*

The following markup defines two highlighting buttons, one for applying yellow highlights and one for green highlights. The button itself doesn't include any text. It simply shows a 15-by-15 square of the appropriate color. The CommandParameter defines a SolidColorBrush that uses the same color but with reduced opacity so the text is still visible:

```
<Button Background="Yellow" Width="15" Height="15" Margin="2,0"
 Command="annot:AnnotationService.CreateHighlightCommand">
  <Button.CommandParameter>
    <SolidColorBrush Color="#54FFFF00"></SolidColorBrush>
  </Button.CommandParameter>
</Button>

<Button Background="LimeGreen" Width="15" Height="15" Margin="2,0"
 Command="annot:AnnotationService.CreateHighlightCommand">
  <Button.CommandParameter>
    <SolidColorBrush Color="#5432CD32"></SolidColorBrush>
  </Button.CommandParameter>
</Button>
```

You can add a final button to remove highlighting in the selected region:

```
<Button Command="annot:AnnotationService.ClearHighlightsCommand">
  Clear Highlights
</Button>
```

■ **Note** When you print a document that includes annotations by using the ApplicationCommands.Print command, the annotations are printed just as they appear. In other words, minimized annotations will appear minimized, visible annotations will appear on top of content (and may obscure other parts of the document), and so on. If you want to

create a printout that doesn't include annotations, simply disable the annotation service before you begin your printout.

Examining Annotations

At some point, you may want to examine all the annotations that are attached to a document. There are many possible reasons—you may want to display a summary report about your annotations, print an annotation list, export annotation text to a file, and so on.

The AnnotationStore makes it relatively easy to get a list of all the annotations it contains by using the GetAnnotations() method. You can then examine each annotation as an Annotation object:

```
IList<Annotation> annotations = service.Store.GetAnnotations();
foreach (Annotation annotation in annotations)
{
    ...
}
```

In theory, you can find annotations in a specific portion of a document by using the overloaded version of the GetAnnotations() method that takes a ContentLocator object. In practice, however, this is tricky, because the ContentLocator object is difficult to use correctly and you need to match the starting position of the annotation precisely.

After you've retrieved an Annotation object, you'll find that it provides the properties listed in Table 28-8.

Table 28-8. *Annotation Properties*

Name	Description
Id	A global identifier (GUID) that uniquely identifies this annotation. If you know the GUID for an annotation, you can retrieve the corresponding Annotation object by using the AnnotationStore.GetAnnotation() method. (Of course, there's no reason you'd know the GUID of an existing annotation unless you had previously retrieved it by calling GetAnnotations(), or you had reacted to an AnnotationStore event when the annotation was created or changed.)
AnnotationType	The XML element name that identifies this type of annotation, in the format *namespace*:*localname*.
Anchors	A collection of zero, one, or more AnnotationResource objects that identify what text is being annotated.
Cargos	A collection of zero, one, or more AnnotationResource objects that contain the user data for the annotation. This includes the text of a text note, or the ink strokes for an ink note.
Authors	A collection of zero, one, or more strings that identify who created the annotation.
CreationTime	The date and time when the annotation was created.
LastModificationTime	The date and time the annotation was last updated.

The Annotation object is really just a thin wrapper over the XML data that's stored for the annotation. One consequence of this design is that it's difficult to pull information out of the Anchors and Cargos properties. For example, if you want to get the actual text of an annotation, you need to look at the second

item in the Cargos selection. This contains the text, but it's stored as a Base64-encoded string (which avoids problems if the note contains characters that wouldn't otherwise be allowed in XML element content). If you want to actually view this text, it's up to you to write tedious code like this to crack it open:

```
// Check for text information.
if (annotation.Cargos.Count > 1)
{
    // Decode the note text.
    string base64Text = annotation.Cargos[1].Contents[0].InnerText;
    byte[] decoded = Convert.FromBase64String(base64Text);

    // Write the decoded text to a stream.
    MemoryStream m = new MemoryStream(decoded);

    // Using the StreamReader, convert the text bytes into a more
    // useful string.
    StreamReader r = new StreamReader(m);
    string annotationXaml = r.ReadToEnd();
    r.Close();

    // Show the annotation content.
    MessageBox.Show(annotationXaml);
}
```

This code gets the text of the annotation, wrapped in a XAML <Section> element. The opening <Section> tag includes attributes that specify a wide range of typography details. Inside the <Section> element are more <Paragraph> and <Run> elements.

■ **Note** Like a text annotation, an ink annotation will also have a Cargos collection with more than one item. However, in this case the Cargos collection will contain the ink data but no decodable text. If you use the previous code on an ink annotation, you'll get an empty message box. Thus, if your document contains both text and ink annotations, you should check the Annotation.AnnotationType property to make sure you're dealing with a text annotation before you use this code.

If you just want to get the text without the surrounding XML, you can use the XamlReader to deserialize it (and avoid using the StreamReader). The XML can be deserialized into a Section object, using code like this:

```
if (annotation.Cargos.Count > 1)
{
    // Decode the note text.
    string base64Text = annotation.Cargos[1].Contents[0].InnerText;
    byte[] decoded = Convert.FromBase64String(base64Text);

    // Write the decoded text to a stream.
    MemoryStream m = new MemoryStream(decoded);

    // Deserialize the XML into a Section object.
    Section section = XamlReader.Load(m) as Section;
```

```
    m.Close();

    // Get the text inside the Section.
    TextRange range = new TextRange(section.ContentStart, section.ContentEnd);

    // Show the annotation content.
    MessageBox.Show(range.Text);
}
```

As Table 28-8 shows, text isn't the only detail you can recover from an annotation. It's easy to get the annotation author, the time it was created, and the time it was last modified.

You can also retrieve information about where an annotation is anchored in your document. The Anchors collection isn't much help for this task, because it provides a low-level collection of AnnotationResource objects that wrap additional XML data. Instead, you need to use the GetAnchorInfo() method of the AnnotationHelper class. This method takes an annotation and returns an object that implements IAnchorInfo.

```
IAnchorInfo anchorInfo = AnnotationHelper.GetAnchorInfo(service, annotation);
```

IAnchorInfo combines the AnnotationResource (the Anchor property), the annotation (Annotation), and an object that represents the location of the annotation in the document tree (ResolvedAnchor), which is the most useful detail. Although the ResolvedAnchor property is typed as an object, text annotations and highlights always return a TextAnchor object. The TextAnchor describes the starting point of the anchored text (BoundingStart) and the ending point (BoundingEnd).

Here's how you could determine the highlighted text for an annotation by using the IAnchorInfo:

```
IAnchorInfo anchorInfo = AnnotationHelper.GetAnchorInfo(service, annotation);
TextAnchor resolvedAnchor = anchorInfo.ResolvedAnchor as TextAnchor;
if (resolvedAnchor != null)
{
    TextPointer startPointer = (TextPointer)resolvedAnchor.BoundingStart;
    TextPointer endPointer = (TextPointer)resolvedAnchor.BoundingEnd;

    TextRange range = new TextRange(startPointer, endPointer);
    MessageBox.Show(range.Text);
}
```

You can also use the TextAnchor objects as a jumping-off point to get to the rest of the document tree, as shown here:

```
// Scroll the document so the paragraph with the annotated text is displayed.
TextPointer textPointer = (TextPointer)resolvedAnchor.BoundingStart;
textPointer.Paragraph.BringIntoView();
```

The samples for this chapter include an example that uses this technique to create an annotation list. When an annotation is selected in the list, the annotated portion of the document is shown automatically.

In both cases, the AnnotationHelper.GetAnchorInfo() method allows you to travel from the annotation to the annotated text, much as the AnnotationStore.GetAnnotations() method allows you to travel from the document content to the annotations.

Although it's relatively easy to examine existing annotations, the WPF annotation feature isn't as strong when it comes to manipulating these annotations. It's easy enough for the user to open a sticky note, drag it to a new position, change the text, and so on, but it's not easy for you to perform these tasks programmatically. In fact, all the properties of the Annotation object are read-only. There are no readily available methods to modify an annotation, so annotation editing involves deleting and re-creating the

annotation. You can do this by using the methods of the AnnotationStore or the AnnotationHelper (if the annotation is attached to the currently selected text). However, both approaches require a fair bit of grunt work. If you use the AnnotationStore, you need to construct an Annotation object by hand. If you use the AnnotationHelper, you need to explicitly set the text selection to include the right text before you create the annotation. Both approaches are tedious and unnecessarily error-prone.

Reacting to Annotation Changes

You've already learned how the AnnotationStore allows you to retrieve the annotations in a document (with GetAnnotations()) and manipulate them (with DeleteAnnotation() and AddAnnotation()). The AnnotationStore provides one additional feature—it raises events that inform you when annotations are changed.

The AnnotationStore provides four events: AnchorChanged (which fires when an annotation is moved), AuthorChanged (which fires when the author information of an annotation changes), CargoChanged (which fires when annotation data, including text, is modified), and StoreContentChanged (which fires when an annotation is created, deleted, or modified in any way).

The online samples for this chapter include an annotation-tracking example. An event handler for the StoreContentChanged event reacts when annotation changes are made. It retrieves all the annotation information (using the GetAnnotations() method) and then displays the annotation text in a list.

■ **Note** The annotation events occur after the change has been made. That means there's no way to plug in custom logic that extends an annotation action. For example, you can't add just-in-time information to an annotation or selectively cancel a user's attempt to edit or delete an annotation.

Storing Annotations in a Fixed Document

The previous examples used annotations on a flow document. In this scenario, annotations can be stored for future use, but they must be stored separately—for example, in a distinct XML file.

When using a fixed document, you can use the same approach, but you have an additional option—you can store annotations directly in the XPS document file. In fact, you could even store multiple sets of distinct annotations, all in the same document. You simply need to use the package support in the System.IO.Packaging namespace.

As you learned earlier, every XPS document is actually a ZIP archive that includes several files. When you store annotations in an XPS document, you are actually creating another file inside the ZIP archive.

The first step is to choose a URI to identify your annotations. Here's an example that uses the name AnnotationStream:

```
Uri annotationUri = PackUriHelper.CreatePartUri(
  new Uri("AnnotationStream", UriKind.Relative));
```

Now you need to get the Package for your XPS document by using the static PackageStore.GetPackage() method:

```
Package package = PackageStore.GetPackage(doc.Uri);
```

You can then create the package part that will store your annotations inside the XPS document. However, you need to check whether the annotation package part already exists (in case you've loaded the document before and already added annotations). If it doesn't exist, you can create it now:

```
PackagePart annotationPart = null;
if (package.PartExists(annotationUri))
{
    annotationPart = package.GetPart(annotationUri);
}
else
{
    annotationPart = package.CreatePart(annotationUri, "Annotations/Stream");
}
```

The last step is to create an AnnotationStore that wraps the annotation package part, and then enable the AnnotationService in the usual way:

```
AnnotationStore store = new XmlStreamStore(annotationPart.GetStream());
service = new AnnotationService(docViewer);
service.Enable(store);
```

In order for this technique to work, you must open the XPS file by using FileMode.ReadWrite mode rather than FileMode.Read, so the annotations can be written to the XPS file. For the same reason, you need to keep the XPS document open while the annotation service is at work. You can close the XPS document when the window is closed (or you choose to open a new document).

Customizing the Appearance of Sticky Notes

The note windows that appear when you create a text note or ink note are instances of the StickyNoteControl class, which is found in the System.Windows.Controls namespace. Like all WPF controls, you can customize the visual appearance of the StickyNoteControl by using style setters or applying a new control template.

For example, you can easily create a style that applies to all StickyNoteControl instances by using the Style.TargetType property. Here's an example that gives every StickyNoteControl a new background color:

```
<Style TargetType="{x:Type StickyNoteControl}">
  <Setter Property="Background" Value="LightGoldenrodYellow"/>
</Style>
```

To make a more dynamic version of the StickyNoteControl, you can write a style trigger that responds to the StickyNoteControl.IsActive property, which is true when the sticky note has focus.

For more control, you can use a completely different control template for your StickyNoteControl. The only trick is that the StickyNoteControl template varies depending on whether it's used to hold an ink note or a text note. If you allow the user to create both types of notes, you need a trigger that can choose between two templates. Ink notes must include an InkCanvas, and text notes must contain a RichTextBox. In both cases, this element should be named PART_ContentControl.

Here's a style that applies the bare minimum control template for both ink and text sticky notes. It sets the dimensions of the note window and chooses the appropriate template based on the type of note content:

```
<Style x:Key="MinimumStyle" TargetType="{x:Type StickyNoteControl}">
  <Setter Property="OverridesDefaultStyle" Value="true" />
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value ="100" />
  <Style.Triggers>
    <Trigger Property="StickyNoteControl.StickyNoteType"
     Value="{x:Static StickyNoteType.Ink}">
```

```
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate>
            <InkCanvas Name="PART_ContentControl" Background="LightYellow" />
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Trigger>
    <Trigger Property="StickyNoteControl.StickyNoteType"
     Value="{x:Static StickyNoteType.Text}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate>
              <RichTextBox Name="PART_ContentControl" Background="LightYellow"/>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Trigger>
  </Style.Triggers>
</Style>
```

The Last Word

Most developers already know that WPF offers its own, specialized model for drawing, layout, and animation. However, WPF's rich document features are often overlooked.

In this chapter, you've seen how to create flow documents, lay out text inside them in a variety of ways, and control how that text is displayed in different containers. You also learned how to use the FlowDocument object model to change portions of the document dynamically, and you considered the RichTextBox, which provides a solid base for advanced text-editing features.

Finally, you took a quick look at fixed documents and the XpsDocument class. The XPS model provides the plumbing for WPF's printing feature, which is the subject of the next chapter.