■ **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.

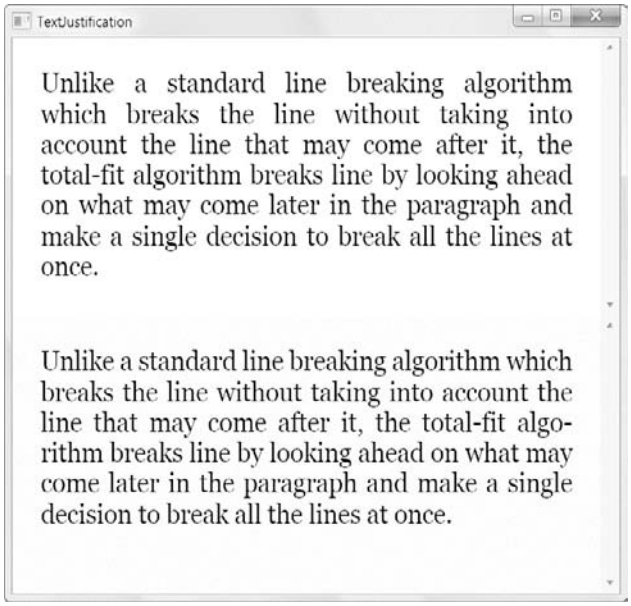

Figure 28-11. *Comparing ordinary justification (top) with optimal paragraphs (bottom)*

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 scroll bar 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 (due to 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 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 actually 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 ellipse is used to indicate that trimming has taken place. Your options are the following:

* **None.** The text is trimmed with no ellipse. "This text is too big" might become "This text is to".
* **WordEllipse.** The ellipse is inserted after the last word that fits (as in "This text is . . .").

* **CharacterEllipse.** The ellipse is inserted after the last character that fits (as in "This text is t . . .").

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 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 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.

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.

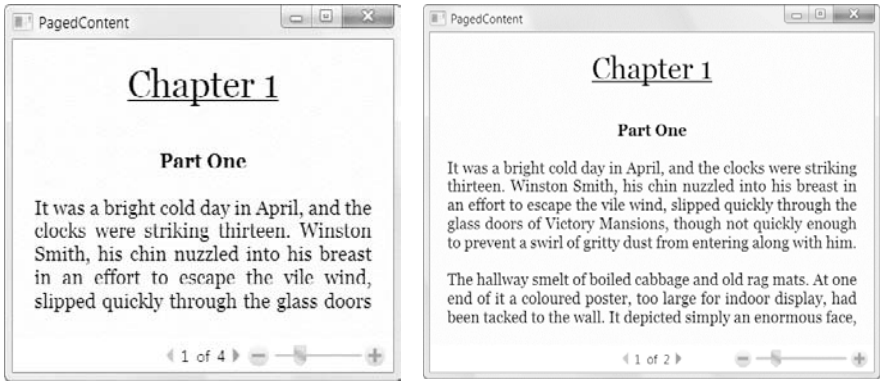

Figure 28-13. *Dynamically repaginated content*

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.

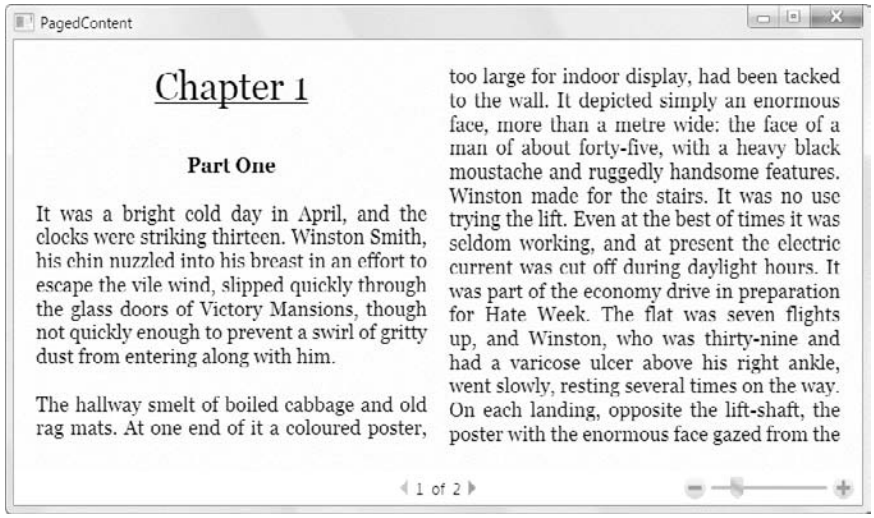

Figure 28-14. *Automatic columns*

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 in between columns.
ColumnRuleWidth and ColumnRuleBrush	Allow you to draw a vertical line in 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, there are situations when 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 once 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 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 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 than 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.

Once the TextRange has been created, you can fill it with data 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 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 only works 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 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 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) 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 using 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 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 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.)

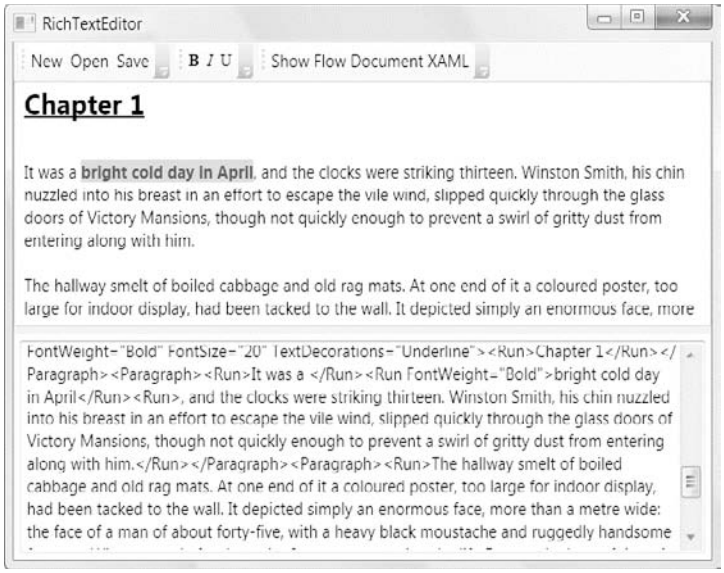

Figure 28-15. *Editing text*

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 the ApplyPropertyValue() to change a dependency property in the selection. For example, you could apply bold formatting to any text elements in the selection 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 Visual Studio 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 popup 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 non-content 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 in 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 Windows Vista and Windows 7. Both versions of Windows include 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 2007 and Microsoft Office 2010 allow 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 its text out to fit the available space. As with flow documents, you can get straightforward support for printing a fixed document 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.

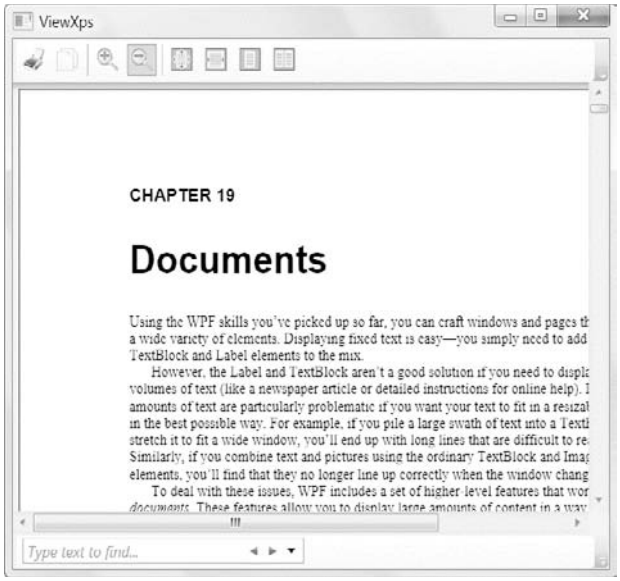

Figure 28-16. *A fixed document*

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.

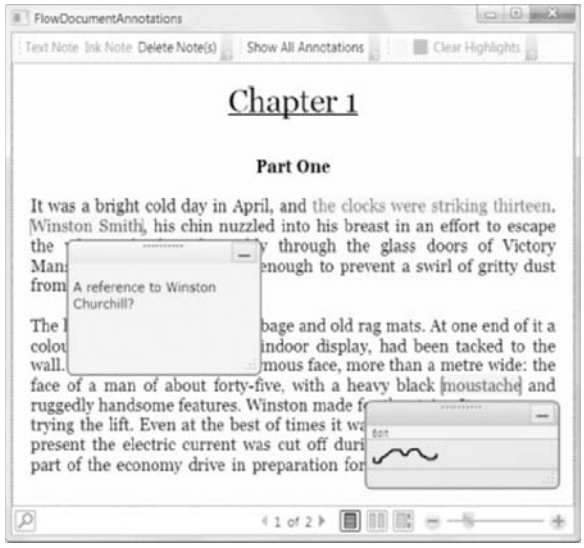

Figure 28-17. *Annotating a flow document*

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 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.

The Annotation Classes

WPF's annotation system relies on several classes from the System.Windows.Annotations and System.Windows.Annotations.Storage namespace. 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 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 of 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, so if you drag one somewhere specific in your document, close it and then reopen it; it will reappear in its previous place.

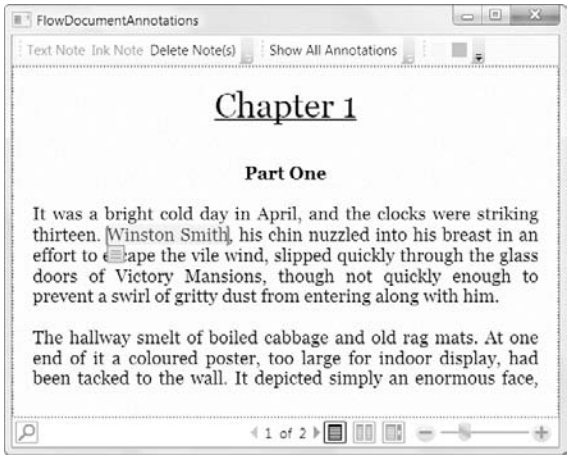

Figure 28-18. *A "hidden" annotation*