

- **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 different 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 non-content 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 non-content 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 in 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.
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 “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 only apply to 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 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 using the `FlowDocumentScrollView` 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" >

    <FlowDocumentScrollView>
        <FlowDocument>
            ...
        </FlowDocument>
    </FlowDocumentScrollView>

</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 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`:

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

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.

The scroll bar 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`.

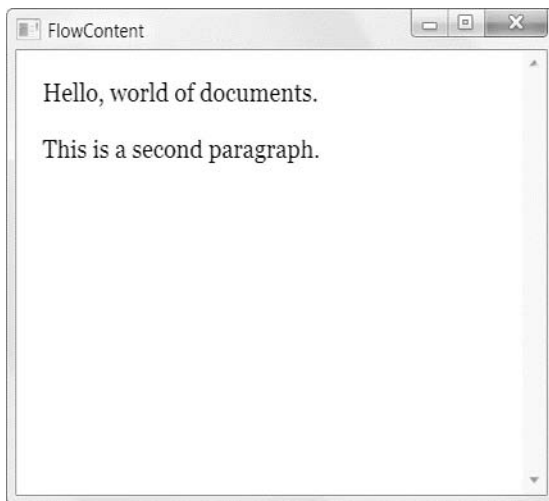


Figure 28-4. A bare-bones flow document

Block Elements

Creating a basic document is easy, but to get the result you really want you need to master a range of different 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.

There are two consequences of this fact. First, it means 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 “Inline Elements” section.

■ **Note** WPF 4 brings one minor refinement to the Run element. In previous versions, Run.Text was an ordinary property, and so didn’t support data binding. In WPF 4, Run.Text is a dependency property, and so you can set it using a data binding expression.

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 “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 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.

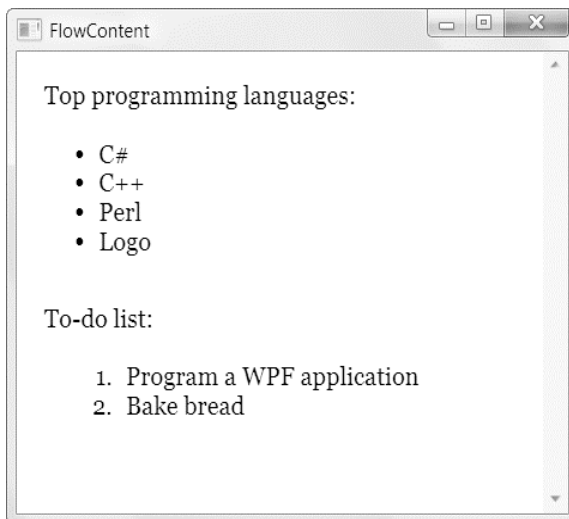


Figure 28-5. Two lists

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>
```



Largest Cities in the Year 100		
Rank	Name	Population
1	Rome	450,000
2	Luoyang (Honan), China	420,000
3	Seleucia (on the Tigris), Iraq	250,000
4	Alexandria, Egypt	250,000
5	Antioch, Turkey	150,000
6	Anuradhapura, Sri Lanka	130,000
7	Peshawar, Pakistan	120,000
8	Carthage, Tunisia	100,000
9	Suzhou, China	n/a
10	Smyrna, Turkey	90,000

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's used to pad in between cells. You can also apply individual formatting (such as different 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 only allow you to draw a border around the entire table. If you're hoping for a more sophisticated effect (for example, adding lines in between columns), you're out of luck.

Another limitation is the fact 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 non-content 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 using a style:

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

The `Section` element is analogous to the `<div>` 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 non-content 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 there are many types of documents that 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 non-content 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 using common controls.

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 non-content 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, it's 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 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. Incidentally, Visual Studio 2005 incorrectly ignores the space in both examples when you're viewing flow document content in a design window. However, when you run your application you'll get the correct behavior.

In some situations, you might want to add space where it would ordinarily be ignored or include a series of spaces. In this situation, you need to use 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.

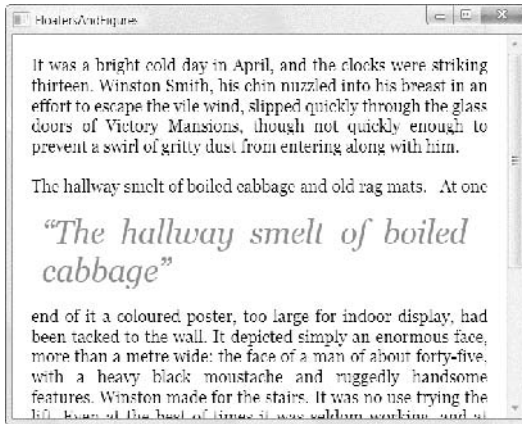


Figure 28-7. A floating pull quote

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 ellipses indicate 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 using the `Width` property. You can also use the `HorizontalAlignment` property to choose 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 only available 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 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 "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% 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 <code>Width</code> and <code>Height</code> properties creates a floating box that's too small for all of its content, some content will be truncated.
HorizontalAnchor	Replaces the <code>HorizontalAlignment</code> property in the <code>Floater</code> class. However, along with three equivalent options (<code>ContentLeft</code> , <code>ContentRight</code> , and <code>ContentCenter</code>), it also includes options that allow you to orient the figure relative to the current page (such as <code>PageCenter</code>) or column (such as <code>ColumnCenter</code>).
VerticalAnchor	Allows you to align the image vertically with respect to the current line of text, the current column, or the current page.

Name	Description
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 `List<Item>` 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) 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.

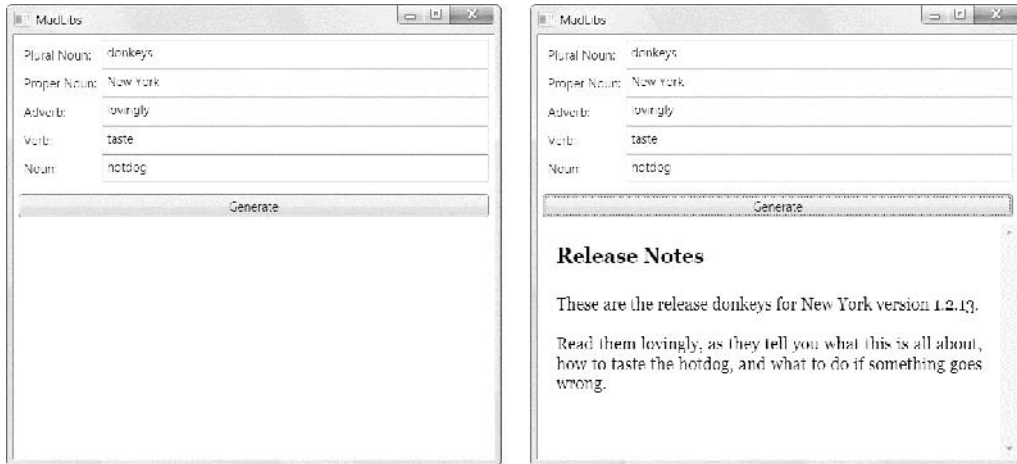


Figure 28-10. Dynamically modifying a flow document

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 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).