```
    <Label Grid.Row="0" Grid.Column="0" Margin="3"
      VerticalAlignment="Center">Home:</Label>
    <TextBox Grid.Row="0" Grid.Column="1" Margin="3"
      Height="Auto" VerticalAlignment="Center"></TextBox>
    <Button Grid.Row="0" Grid.Column="2" Margin="3" Padding="2">
      Browse</Button>
    ...

  </Grid>
</ScrollViewer>
```

The result is shown in Figure 6-9.

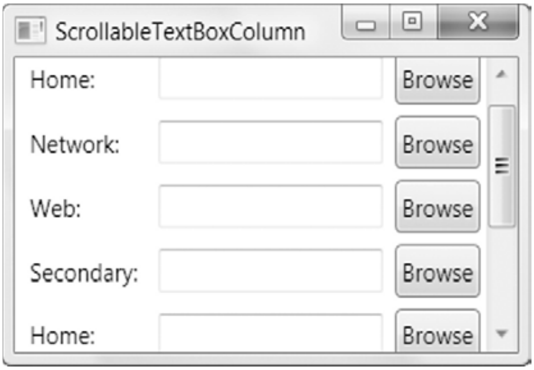


Figure 6-9. *A scrollable window*

If you resize the window in this example so that it's large enough to fit all its content, the scroll bar becomes disabled. However, the scroll bar will still be visible. You can control this behavior by setting the VerticalScrollBarVisibility property, which takes a value from the ScrollBarVisibility enumeration. The default value of Visible makes sure the vertical scroll bar is always present. Use Auto if you want the scroll bar to appear when it's needed and disappear when it's not. Or use Disabled if you don't want the scroll bar to appear at all.

■ **Note** You can also use Hidden, which is similar to Disabled but subtly different. First, content with a hidden scroll bar is still scrollable. (For example, you can scroll through the content using the arrow keys.) Second, the content in a ScrollViewer is laid out differently. When you use Disabled, you tell the content in the ScrollViewer that it has only as much space as the ScrollViewer itself. On the other hand, if you use Hidden, you tell the content that it has an infinite amount of space. That means it can overflow and stretch off into the scrollable region. Ordinarily, you'll use Hidden only if you plan to allow scrolling by another mechanism (such as the custom scrolling buttons described next). You'll use Disabled only if you want to temporarily prevent the ScrollViewer from doing anything at all.

The ScrollViewer also supports horizontal scrolling. However, the HorizontalScrollBarVisibility property is Hidden by default. To use horizontal scrolling, you need to change this value to Visible or Auto.

Programmatic Scrolling

To scroll through the window shown in Figure 6-9, you can click the scroll bar with the mouse, you can move over the grid and use a mouse scroll wheel, you can tab through the controls, or you can click somewhere on the blank surface of the grid and use the up and down arrow keys. If this still doesn't give you the flexibility you crave, you can use the methods of the ScrollViewer class to scroll your content programmatically:

- The most obvious are LineUp() and LineDown(), which are equivalent to clicking the arrow buttons on the vertical scroll bar to move up or down once.
- You can also use PageUp() and PageDown(), which scroll an entire screenful up or down and are equivalent to clicking the surface of the scroll bar, above or below the scroll bar thumb.
- Similar methods allow horizontal scrolling, including LineLeft(), LineRight(), PageLeft(), and PageRight().
- Finally, you can use the ScrollTo*Xxx*() methods to go somewhere specific. For vertical scrolling, they include ScrollToEnd() and ScrollToHome(), which take you to the top or bottom of the scrollable content, and ScrollToVerticalOffset(), which takes you to a specific position. There are horizontal versions of the same methods, including ScrollToLeftEnd(), ScrollToRightEnd(), and ScrollToHorizontalOffset().

Figure 6-10 shows an example where several custom buttons allow you to move through the ScrollViewer. Each button triggers a simple event handler that uses one of the methods in the previous list.

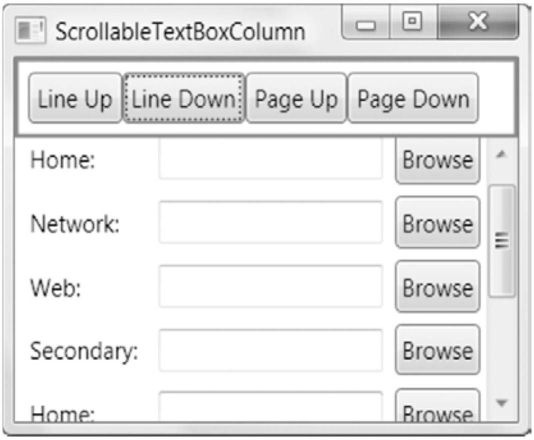


Figure 6-10. *Programmatic scrolling*

Custom Scrolling

The built-in scrolling in the ScrollViewer is quite useful. It allows you to scroll slowly through any content, from a complex vector drawing to a grid of elements. However, one of the most intriguing features of the ScrollViewer is its ability to let its content participate in the scrolling process. Here's how it works:

- You place a scrollable element inside the ScrollViewer. This is any element that implements IScrollInfo.
- You tell the ScrollViewer that the content knows how to scroll itself by setting the ScrollViewer.CanContentScroll property to true.
- When you interact with the ScrollViewer (by using the scroll bar, the mouse wheel, the scrolling methods, and so on), the ScrollViewer calls the appropriate methods on your element using the IScrollInfo interface. The element then performs its own custom scrolling.

■ **Note** The IScrollInfo interface defines a set of methods that react to different scrolling actions. For example, it includes many of the scrolling methods exposed by the ScrollViewer, such as LineUp(), LineDown(), PageUp(), and PageDown(). It also defines methods that handle the mouse wheel.

Very few elements implement IScrollInfo. One element that does is the StackPanel container. Its IScrollInfo implementation uses *logical scrolling*, which is scrolling that moves from element to element, rather than from line to line.

If you place a StackPanel in a ScrollViewer and you don't set the CanContentScroll property, you get the ordinary behavior. Scrolling up and down moves you a few pixels at a time. However, if you set CanContentScroll to true, each time you click down, you scroll to the beginning of the next element:

```
<ScrollViewer CanContentScroll="True">
  <StackPanel>
    <Button Height="100">1</Button>
    <Button Height="100">2</Button>
    <Button Height="100">3</Button>
    <Button Height="100">4</Button>
  </StackPanel>
</ScrollViewer>
```

You may or may not find that the StackPanel's logical scrolling system is useful in your application. However, it's indispensable if you want to create a custom panel with specialized scrolling behavior.

Headered Content Controls

One of the classes that derive from ContentControl is HeaderedContentControl. Its role is simple—it represents a container that has both single-element content (as stored in the Content property) and a single-element header (as stored in the Header property). The addition of the header is what distinguishes the HeaderedContentControl from the content controls you've seen so far.

Three classes derive from HeaderedContentControl: GroupBox, TabItem, and Expander. You'll explore them in the following sections.

The GroupBox

The GroupBox is the simplest of the three controls that derives from HeaderedContentControl. It's displayed as a box with rounded corners and a title. Here's an example (shown in Figure 6-11):

```
<GroupBox Header="A GroupBox Test" Padding="5"
  Margin="5" VerticalAlignment="Top">
  <StackPanel>
    <RadioButton Margin="3">One</RadioButton>
    <RadioButton Margin="3">Two</RadioButton>
    <RadioButton Margin="3">Three</RadioButton>
    <Button Margin="3">Save</Button>
  </StackPanel>
</GroupBox>
```

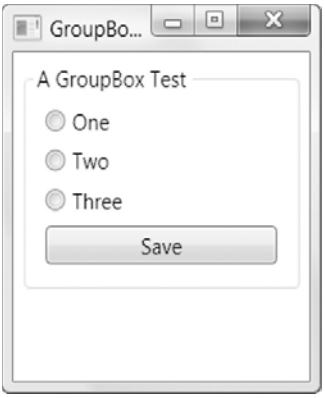


Figure 6-11. A basic group box

Notice that the GroupBox still requires a layout container (such as a StackPanel) to arrange its contents. The GroupBox is often used to group small sets of related controls, such as radio buttons. However, the GroupBox has no built-in functionality, so you can use it however you want. (RadioButton objects are grouped by placing them into any panel. A GroupBox is not required, unless you want the rounded, titled border.)

The TabItem

The TabItem represents a page in a TabControl. The only significant member that the TabItem class adds is the IsSelected property, which indicates whether the tab is currently being shown in the TabControl. Here's the markup that's required to create the simple example shown in Figure 6-12:

```
<TabControl Margin="5">
  <TabItem Header="Tab One">
    <StackPanel Margin="3">
      <CheckBox Margin="3">Setting One</CheckBox>
      <CheckBox Margin="3">Setting Two</CheckBox>
      <CheckBox Margin="3">Setting Three</CheckBox>
    </StackPanel>
  </TabItem>
  <TabItem Header="Tab Two">
    ...
  </TabItem>
</TabControl>
```

■ **Tip** You can use the TabStripPlacement property to make the tabs appear on the side of the tab control, rather than in their normal location at the top.

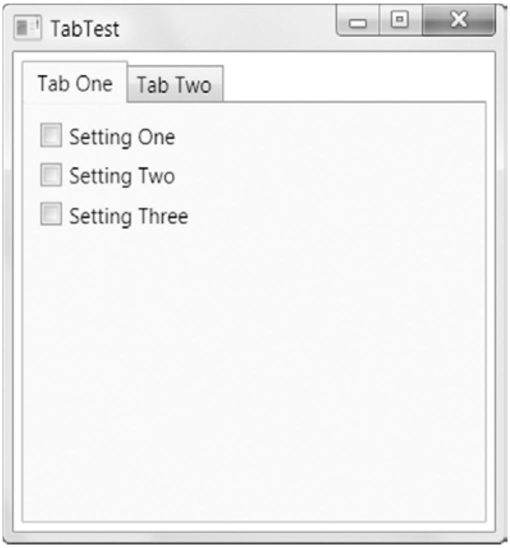


Figure 6-12. A set of tabs

As with the Content property, the Header property can accept any type of object. It displays UIElement-derived classes by rendering them and uses the ToString() method for inline text and all other objects. That means you can create a group box or a tab with graphical content or arbitrary elements in its title. Here's an example:

```
<TabControl Margin="5">
  <TabItem>
    <TabItem.Header>
      <StackPanel>
        <TextBlock Margin="3" >Image and Text Tab Title</TextBlock>
        <Image Source="happyface.jpg" Stretch="None" />
      </StackPanel>
    </TabItem.Header>

    <StackPanel Margin="3">
      <CheckBox Margin="3">Setting One</CheckBox>
      <CheckBox Margin="3">Setting Two</CheckBox>
      <CheckBox Margin="3">Setting Three</CheckBox>
    </StackPanel>
  </TabItem>

  <TabItem Header="Tab Two"></TabItem>
</TabControl>
```

Figure 6-13 shows the somewhat garish result.

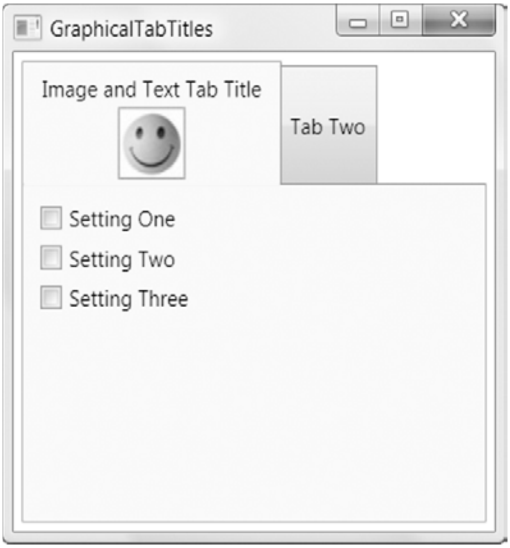
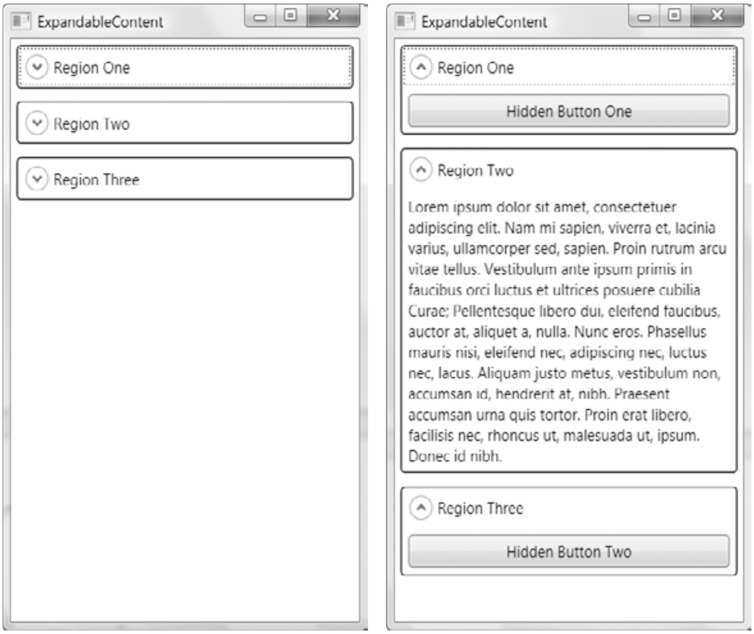


Figure 6-13. *An exotic tab title*

The Expander

The most exotic headered content control is the Expander. It wraps a region of content that the user can show or hide by clicking a small arrow button. This technique is used frequently in online help and on web pages, to allow them to include large amounts of content without overwhelming users with information they don't want to see.

Figure 6-14 shows two views of a window with three expanders. In the version on the left, all three expanders are collapsed. In the version on the right, all the regions are expanded. (Of course, users are free to expand or collapse any combination of expanders individually.)



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien. Proin rutrum arcu vitae tellus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque libero dui, eleifend faucibus, auctor at, aliquet a, nulla. Nunc eros. Phasellus mauris nisi, eleifend nec, adipiscing nec, luctus nec, lacus. Aliquam justo metus, vestibulum non, accumsan id, hendrerit at, nibh. Praesent accumsan urna quis tortor. Proin erat libero, facilisis nec, rhoncus ut, malesuada ut, ipsum. Donec id nibh.

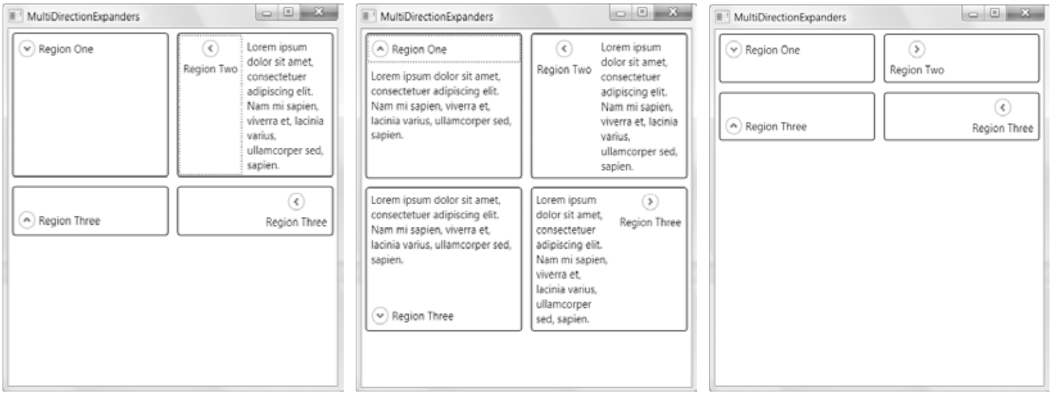
Figure 6-14. Hiding content with expandable regions

Using an Expander is extremely simple—you just need to wrap the content you want to make collapsible inside. Ordinarily, each Expander begins collapsed, but you can change this in your markup (or in your code) by setting the IsExpanded property. Here's the markup that creates the example shown in Figure 6-14:

```
<StackPanel>
  <Expander Margin="5" Padding="5" Header="Region One">
    <Button Padding="3">Hidden Button One</Button>
  </Expander>
  <Expander Margin="5" Padding="5" Header="Region Two" >
    <TextBlock TextWrapping="Wrap">
      Lorem ipsum dolor sit amet, consectetuer adipiscing elit ...
    </TextBlock>
  </Expander>
  <Expander Margin="5" Padding="5" Header="Region Three">
    <Button Padding="3">Hidden Button Two</Button>
  </Expander>
</StackPanel>
```

You can also choose in which direction the expander expands. In Figure 6-14, the standard value (Down) is used, but you can also set the ExpandDirection property to Up, Left, or Right. When the Expander is collapsed, the arrow always points in the direction where it will expand.

Life gets a little interesting when using different ExpandDirection values, because the effect on the rest of your user interface depends on the type of container. Some containers, such as the WrapPanel, simply bump other elements out of the way. Others, such as Grid, have the option of using proportional or automatic sizing. Figure 6-15 shows an example with a four-cell grid in various degrees of expansion. In each cell is an Expander with a different ExpandDirection. The columns are sized proportionately, which forces the text in the Expander to wrap. (An autosized column would simply stretch to fit the text, making it larger than the window.) The rows are set to automatic sizing, so they expand to fit the extra content.



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam mi sapien, viverra et, lacinia varius, ullamcorper sed, sapien.

Figure 6-15. Expanding in different directions

The Expander is a particularly nice fit in WPF because WPF encourages you to use a flowing layout model that can easily handle content areas that grow or shrink dynamically.

If you need to synchronize other controls with an Expander, you can handle the Expanded and Collapsed events. Contrary to what the naming of these events implies, they fire just *before* the content appears or disappears. This gives you a useful way to implement a lazy load. For example, if the content in an Expander is expensive to create, you might wait until it's shown to retrieve it. Or perhaps you want to update the content just before it's shown. Either way, you can react to the Expanded event to perform your work.

■ **Note** If you like the functionality of the Expander but aren't impressed with the built-in appearance, don't worry. Using the template system in WPF, you can completely customize the expand and collapse arrows so they match the style of the rest of your application. You'll learn how in Chapter 17.

Ordinarily, when you expand an Expander, it grows to fit its content. This may create a problem if your window isn't large enough to fit all the content when everything is expanded. You can use several strategies to handle this problem:

- Set a minimum size for the window (using MinWidth and MinHeight) to make sure it will fit everything even at its smallest.
- Set the SizeToContent property of the window so that it expands automatically to fit the exact dimensions you need when you open or close an Expander. Ordinarily, SizeToContent is set to Manual, but you can use Width or Height to make it expand or contract in either dimension to accommodate its content.
- Limit the size of the Expander by hard-coding its Height and Width. Unfortunately, this is likely to truncate the content that's inside if it's too large.
- Create a scrollable expandable region using the ScrollViewer.

For the most part, these techniques are quite straightforward. The only one that requires any further exploration is the combination of an Expander and a ScrollViewer. In order for this approach to work, you need to hard-code the size for the ScrollViewer. Otherwise, it will simply expand to fit its content. Here's an example:

```
<Expander Margin="5" Padding="5" Header="Region Two">
  <ScrollViewer Height="50">
    <TextBlock TextWrapping="Wrap">
     ...
    </TextBlock>
  </ScrollViewer>
</Expander>
```

It would be nice to have a system in which an Expander could set the size of its content region based on the available space in a window. However, this would present obvious complexities. (For example, how would space be shared between multiple regions when an Expander expands?) The Grid layout container might seem like a potential solution, but unfortunately, it doesn't integrate well with the Expander. If you try it out, you'll end up with oddly spaced rows that don't update their heights properly when an Expander is collapsed.

Text Controls

WPF includes three text-entry controls: TextBox, RichTextBox, and PasswordBox. The PasswordBox derives directly from Control. The TextBox and RichTextBox controls go through another level and derive from TextBoxBase.

Unlike the content controls you've seen, the text boxes are limited in the type of content they can contain. The TextBox always stores a string (provided by the Text property). The PasswordBox also deals with string content (provided by the Password property), although it uses a SecureString internally to mitigate against certain types of attacks. Only the RichTextBox has the ability to store more sophisticated content: a FlowDocument that can contain a complex combination of elements.

In the following sections, you'll consider the core features of the TextBox. You'll end by taking a quick look at the security features of the PasswordBox.

■ **Note** The RichTextBox is an advanced control design for displaying FlowDocument objects. You'll learn how to use it when you tackle documents in Chapter 28.

Multiple Lines of Text

Ordinarily, the TextBox control stores a single line of text. (You can limit the allowed number of characters by setting the MaxLength property.) However, there are many cases when you'll want to create a multiline text box for dealing with large amounts of content. In this case, set the TextWrapping property to Wrap or WrapWithOverflow. Wrap always breaks at the edge of the control, even if it means severing an extremely long word in two. WrapWithOverflow allows some lines to stretch beyond the right edge if the line-break algorithm can't find a suitable place (such as a space or a hyphen) to break the line.

To actually see multiple lines in a text box, it needs to be sized large enough. Rather than setting a hard-coded height (which won't adapt to different font sizes and may cause layout problems), you can use the handy MinLines and MaxLines properties. MinLines is the minimum number of lines that must be visible in the text box. For example, if MinLines is 2, the text box will grow to be at least two lines tall. If its container doesn't have enough room, part of the text box may be clipped. MaxLines sets the maximum number of lines that will be displayed. Even if a text box expands to fit its container (for example, a proportionally sized Grid row or the last element in a DockPanel), it won't grow beyond this limit.

■ **Note** The MinLines and MaxLines properties have no effect on the amount of content you can place in a text box. They simply help you size the text box. In your code, you can examine the LineCount property to find out exactly how many lines are in a text box.

If your text box supports wrapping, the odds are good that the user can enter more text that can be displayed at once in the visible lines. For this reason, it usually makes sense to add an always-visible or on-demand scroll bar by setting the VerticalScrollBarVisibility property to Visible or Auto. (You can also set the HorizontalScrollBarVisibility property to show a less common horizontal scroll bar.)

You may want to allow the user to enter hard returns in a multiline text box by pressing the Enter key. (Ordinarily, pressing the Enter key in a text box triggers the default button.) To make sure a text box supports the Enter key, set AcceptsReturn to true. You can also set AcceptsTab to allow the user to insert tabs. Otherwise, the Tab key moves to the next focusable control in the tab sequence.

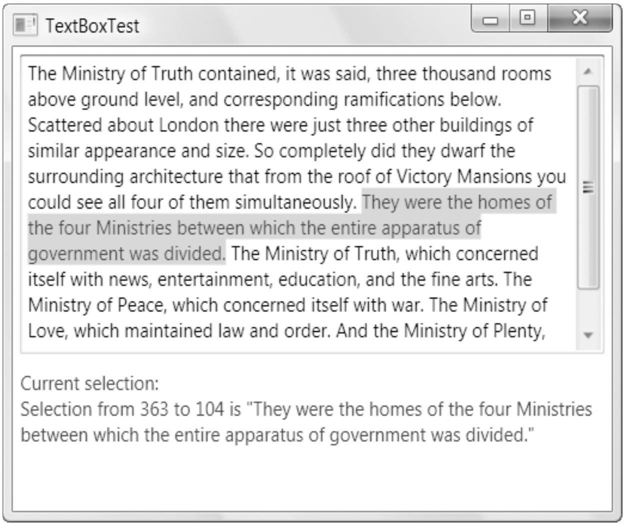
■ **Tip** The TextBox class also includes a host of methods that let you move through the text content programmatically in small or large steps. They include LineUp(), LineDown(), PageUp(), PageDown(), ScrollToHome(), ScrollToEnd(), and ScrollToLine().

Sometimes, you'll create a text box purely for the purpose of displaying text. In this case, set the IsReadOnly property to true to prevent editing. This is preferable to disabling the text box by setting IsEnabled to false because a disabled text box shows grayed-out text (which is more difficult to read), does not support selection (or copying to the clipboard), and does not support scrolling.

Text Selection

As you already know, you can select text in any text box by clicking and dragging with the mouse or holding down Shift while you move through the text with the arrow keys. The TextBox class also gives you the ability to determine or change the currently selected text programmatically, using the SelectionStart, SelectionLength, and SelectedText properties.

SelectionStart identifies the zero-based position where the selection begins. For example, if you set this property to 10, the first selected character is the eleventh character in the text box. The Selection Length indicates the total number of selected characters. (A value of 0 indicates no selected characters.) Finally, the SelectedText property allows you to quickly examine or change the selected text in the text box. You can react to the selection being changed by handling the SelectionChanged event. Figure 6-16 shows an example that reacts to this event and displays the current selection information.



The Ministry of Truth contained, it was said, three thousand rooms above ground level, and corresponding ramifications below. Scattered about London there were just three other buildings of similar appearance and size. So completely did they dwarf the surrounding architecture that from the roof of Victory Mansions you could see all four of them simultaneously. They were the homes of the four Ministries between which the entire apparatus of government was divided. The Ministry of Truth, which concerned itself with news, entertainment, education, and the fine arts. The Ministry of Peace, which concerned itself with war. The Ministry of Love, which maintained law and order. And the Ministry of Plenty,

Current selection:
Selection from 363 to 104 is "They were the homes of the four Ministries between which the entire apparatus of government was divided."

Figure 6-16. *Selecting text*

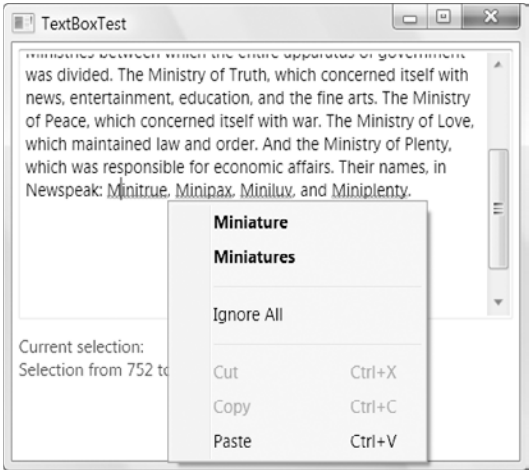
The TextBox class also includes one property that lets you control its selection behavior: AutoWordSelection. If this is true, the text box selects entire words at a time as you drag through the text.

Another useful feature of the TextBox control is Undo, which allows the user to reverse recent changes. The Undo feature is available programmatically (using the Undo() method), and it's available using the Ctrl+Z keyboard shortcut, as long as the CanUndo property has not been set to false.

■ **Tip** When manipulating text in the text box programmatically, you can use the BeginChange() and EndChange() methods to bracket a series of actions that the TextBox will treat as a single block of changes. These actions can then be undone in a single step.

Spell Checking

The TextBox includes an unusual frill: an integrated spell-check feature, which underlines unrecognized words with a red squiggly line. The user can right-click an unrecognized word and choose from a list of possibilities, as shown in Figure 6-17.



Ministries between which the entire apparatus of government was divided. The Ministry of Truth, which concerned itself with news, entertainment, education, and the fine arts. The Ministry of Peace, which concerned itself with war. The Ministry of Love, which maintained law and order. And the Ministry of Plenty, which was responsible for economic affairs. Their names, in Newspeak: Minitrue, Minipax, Miniluv, and Miniplenty.

Current selection:
Selection from 752 t

Figure 6-17. Spell-checking a text box

To turn on the spell-check functionality for the TextBox control, you simply need to set the SpellCheck.IsEnabled dependency property, as shown here:

```
<TextBox SpellCheck.IsEnabled="True">...</TextBox>
```

The spelling checker is WPF-specific and doesn't depend on any other software (such as Office). The spelling checker determines which dictionary to use based on the input language that's configured for the keyboard. You can override this default by setting the Language property of the TextBox, which is inherited from the FrameworkElement class, or you can set the xml:lang attribute on the <TextBox> element. However, the WPF spelling checker is currently limited to just four languages: English, Spanish, French, and German. You can use the SpellingReform property to set whether post-1990 spelling rule changes are applied to French and German languages.

In previous versions of WPF, the spelling checker did not support customization. WPF 4 allows you to add a list of words that will not be treated as errors (and will be used as right-click suggestions, when appropriate). To do so, you must first create a lexicon file, which is nothing more than a text file with the extension .lex. In the lexicon file, you add the list of words. Place each word on a separate line, in any order, as shown here:

```
acantholysis
atypia
bulla
chromonychia
dermatoscopy
desquamation
...
```

In this example, the words are used regardless of the current language setting. However, you can specify that a lexicon should be used only for a specific language by adding a locale ID. Here's how you would specify that the custom words should be used only when the current language is English:

```
#LID 1033
acantholysis
atypia
bulla
chromonychia
dermatoscopy
desquamation
...
```

The other supported locale IDs are 3082 (Spanish), 1036 (French), and 1031 (German).

■ **Note** The custom dictionary feature is not designed to allow you to use additional languages. Instead, it simply augments an already supported language (like English) with the words you supply. For example, you can use a custom dictionary to recognize proper names or to allow medical terms in a medical application.

Once you've created the lexicon file, make sure the SpellCheck.IsEnabled property is set to true for your TextBox. The final step is to attach a Uri object that points to your custom dictionary, using the SpellCheck.CustomDictionaries property. If you choose to specify it in XAML, as in the following example, you must first import the System namespace so that you can declare a Uri object in markup:

```
<Window xmlns:sys="clr-namespace:System;assembly=system" ... >
```

You can use multiple custom dictionaries at once, as long as you add a Uri object for each one. Each Uri can use a hard-coded path to the file on a local drive or network share. But the safest approach is to use an application resource. For example, if you've added the file CustomWords.lex to a project named SpellTest, and you've set the Build Action of that file to Resource (using the Solution Explorer), you will use markup like this:

```
<TextBox TextWrapping="Wrap" SpellCheck.IsEnabled="True"
 Text="Now the spell checker recognizes acantholysis and offers the right correction
for acantholysi">
  <SpellCheck.CustomDictionaries>
    <sys:Uri>pack://application:,,,/SpellTest;component/CustomWords.lex</sys:Uri>
  </SpellCheck.CustomDictionaries>
</TextBox>
```

The odd `pack://application:,,,/` portion at the beginning of the URI is the pack URI syntax that WPF uses to refer to an assembly resource. You'll take a closer look at it when you consider resources in detail in Chapter 7.

If you need to load the lexicon file from the application directory, the easiest option is to create the URI you need using code, and add it to the SpellCheck.CustomDictionaries collection when the window is initialized.

The PasswordBox

The PasswordBox looks like a TextBox, but it displays a string of circle symbols to mask the characters it shows. (You can choose a different mask character by setting the PasswordChar property.) Additionally, the PasswordBox does not support the clipboard, so you can't copy the text inside.

Compared to the TextBox class, the PasswordBox has a much simpler, stripped-down interface. Much like the TextBox class, it provides a MaxLength property; Clear(), Paste() and SelectAll() methods; and an event that fires when the text is changed (named PasswordChanged). But that's it. Still, the most important difference between the TextBox and the PasswordBox is on the inside. Although you can set text and read it as an ordinary string using the Password property, internally the PasswordBox uses a System.Security.SecureString object exclusively.

A SecureString is a text-only object much like the ordinary string. The difference is how it's stored in memory. A SecureString is stored in memory in an encrypted form. The key that's used to encrypt the string is generated randomly and stored in a portion of memory that's never written to disk. The end result is that even if your computer crashes, malicious users won't be able to examine the paging file to retrieve the password data. At best, they will find the encrypted form.

The SecureString class also includes on-demand disposal. When you call SecureString.Dispose(), the in-memory password data is overwritten. This guarantees that all password information has been wiped out of memory and is no longer subject to any kind of exploit. As you would expect, the PasswordBox is conscientious enough to call Dispose() on the SecureString that it stores internally when the control is destroyed.

List Controls

WPF includes many controls that wrap a collection of items, ranging from the simple ListBox and ComboBox that you'll examine here to more specialized controls such as the ListView, the TreeView, and the ToolBar, which are covered in future chapters. All of these controls derive from the ItemsControl class (which itself derives from Control).

The ItemsControl class fills in the basic plumbing that's used by all list-based controls. Notably, it gives you two ways to fill the list of items. The most straightforward approach is to add them directly to the Items collection, using code or XAML. However, in WPF, it's more common to use data binding. In this case, you set the ItemsSource property to the object that has the collection of data items you want to display. (You'll learn more about data binding with a list in Chapter 19.)

The class hierarchy that leads from ItemsControls is a bit tangled. One major branch is the *selectors*, which includes the ListBox, the ComboBox, and the TabControl. These controls derive from Selector and have properties that let you track down the currently selected item (SelectedItem) or its position (SelectedIndex). Separate from these are controls that wrap lists of items but don't support selection in the same way. These include the classes for menus, toolbars, and trees—all of which are ItemsControls but aren't selectors.

In order to unlock most of the features of any ItemsControl, you'll need to use data binding. This is true even if you aren't fetching your data from a database or an external data source. WPF data binding is general enough to work with data in a variety of forms, including custom data objects and collections. But you won't consider the details of data binding just yet. For now, you'll take only a quick look at the ListBox and ComboBox.

The ListBox

The ListBox class represents a common staple of Windows design—the variable-length list that allows the user to select an item.

■ **Note** The ListBox class also allows multiple selection if you set the SelectionMode property to Multiple or Extended. In Multiple mode, you can select or deselect any item by clicking it. In Extended mode, you need to hold down the Ctrl key to select additional items or the Shift key to select a range of items. In either type of multiple-selection list, you use the SelectedItems collection instead of the SelectedItem property to get all the selected items.

To add items to the ListBox, you can nest ListBoxItem elements inside the ListBox element. For example, here's a ListBox that contains a list of colors:

```
<ListBox>
  <ListBoxItem>Green</ListBoxItem>
  <ListBoxItem>Blue</ListBoxItem>
  <ListBoxItem>Yellow</ListBoxItem>
  <ListBoxItem>Red</ListBoxItem>
</ListBox>
```

As you'll remember from Chapter 2, different controls treat their nested content in different ways. The ListBox stores each nested object in its Items collection.

The ListBox is a remarkably flexible control. Not only can it hold ListBoxItem objects, but it can also host any arbitrary element. This works because the ListBoxItem class derives from ContentControl, which gives it the ability to hold a single piece of nested content. If that piece of content is a UIElement-derived class, it will be rendered in the ListBox. If it's some other type of object, the ListBoxItem will call ToString() and display the resulting text.

For example, if you decided you want to create a list with images, you could create markup like this:

```
<ListBox>
  <ListBoxItem>
    <Image Source="happyface.jpg"></Image>
  </ListBoxItem>
  <ListBoxItem>
    <Image Source="happyface.jpg"></Image>
  </ListBoxItem>
</ListBox>
```

The ListBox is actually intelligent enough to create the ListBoxItem objects it needs implicitly. That means you can place your objects directly inside the ListBox element. Here's a more ambitious example that uses nested StackPanel objects to combine text and image content:

```
<ListBox>
  <StackPanel Orientation="Horizontal">
    <Image Source="happyface.jpg"  Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A happy face</Label>
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <Image Source="redx.jpg" Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A warning sign</Label>
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <Image Source="happyface.jpg"  Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A happy face</Label>
  </StackPanel>
</ListBox>
```

In this example, the StackPanel becomes the item that's wrapped by the ListBoxItem. This markup creates the rich list shown in Figure 6-18.

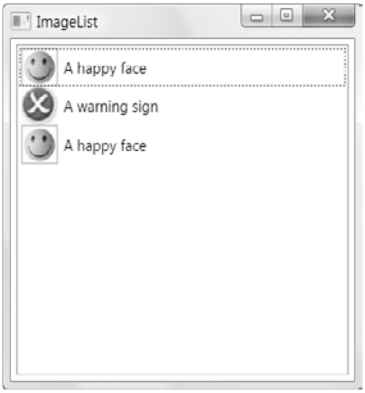


Figure 6-18. *A list of images*

■ **Note** One flaw in the current design is that the text color doesn't change when the item is selected. This isn't ideal because it's difficult to read the black text with a blue background. To solve this problem, you need to use a data template, as described in Chapter 20.

This ability to nest arbitrary elements inside list box items allows you to create a variety of list-based controls without needing to use other classes. For example, the Windows Forms toolkit includes a

CheckedListBox class that's displayed as a list with a check box next to every item. No such specialized class is required in WPF because you can quickly build one using the standard ListBox:

```
<ListBox Name="lst" SelectionChanged="lst_SelectionChanged"
  CheckBox.Click="lst_SelectionChanged">
  <CheckBox Margin="3">Option 1</CheckBox>
  <CheckBox Margin="3">Option 2</CheckBox>
</ListBox>
```

There's one caveat to be aware of when you use a list with different elements inside. When you read the SelectedItem value (and the SelectedItems and Items collections), you won't see ListBoxItem objects; instead, you'll see whatever objects you placed in the list. In the CheckedListBox example, that means SelectedItem provides a CheckBox object.

For example, here's some code that reacts when the SelectionChanged event fires. It then gets the currently selected CheckBox and displays whether that item has been checked:

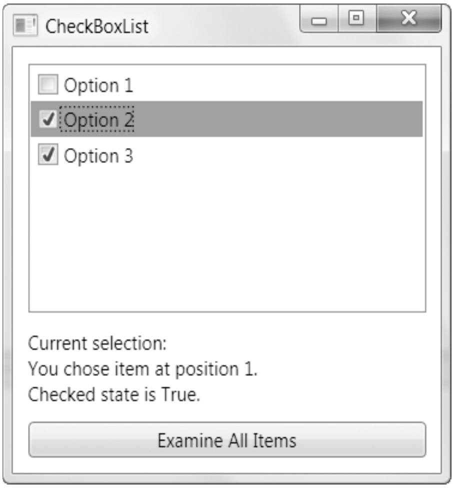
```
private void lst_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lst.SelectedItem == null) return;
    txtSelection.Text = String.Format(
      "You chose item at position {0}.\r\nChecked state is {1}.",
      lst.SelectedIndex,
      ((CheckBox)lst.SelectedItem).IsChecked);
}
```

■ **Tip** If you want to find the current selection, you can read it directly from the SelectedItem or SelectedItems property, as shown here. If you want to determine which item (if any) was *unselected*, you can use the RemovedItems property of the SelectionChangedEventArgs object. Similarly, the AddedItems property tells you which items were added to the selection. In single-selection mode, one item is always added and one item is always removed whenever the selection changes. In multiple or extended mode, this isn't necessarily the case.

In the following code snippet, similar code loops through the collection of items to determine which ones are checked. (You could write similar code that loops through the collection of selected items in a multiple-selection list with check boxes.)

```
private void cmd_ExamineAllItems(object sender, RoutedEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    foreach (CheckBox item in lst.Items)
    {
        if (item.IsChecked == true)
        {
            sb.Append(item.Content);
            sb.Append(" is checked.");
            sb.Append("\r\n");
        }
    }
    txtSelection.Text = sb.ToString();
}
```

Figure 6-19 shows the list box that uses this code.



□ Option 1
☑ Option 2
☑ Option 3

Current selection:
You chose item at position 1.
Checked state is True.

***Figure 6-19.** A check box list*

When manually placing items in a list, it's up to you whether you want to insert the items directly or explicitly wrap each one in a ListBoxItem object. The second approach is often cleaner, albeit more tedious. The most important consideration is to be consistent. For example, if you place StackPanel objects in your list, the ListBox.SelectedItem object will be a StackPanel. If you place StackPanel objects wrapped by ListBoxItem objects, the ListBox.SelectedItem object will be a ListBoxItem, so code accordingly.

The ListBoxItem offers a little extra functionality from what you get with directly nested objects. Namely, it defines an IsSelected property that you can read (or set) and a Selected and Unselected event that tells you when that item is highlighted. However, you can get similar functionality using the members of the ListBox class, such as the SelectedItem (or SelectedItems) property, and the SelectionChanged event.

Interestingly, there's a technique to retrieve a ListBoxItem wrapper for a specific object when you use the nested object approach. The trick is the often overlooked ContainerFromElement() method. Here's the code that checks whether the first item is selected in a list using this technique:

```
ListBoxItem item = (ListBoxItem)lst.ContainerFromElement(
  (DependencyObject)lst.SelectedItems[0]);
MessageBox.Show("IsSelected: " + item.IsSelected.ToString());
```

The ComboBox

The ComboBox is similar to the ListBox control. It holds a collection of ComboBoxItem objects, which are created either implicitly or explicitly. As with the ListBoxItem, the ComboBoxItem is a content control that can contain any nested element.

The key difference between the ComboBox and ListBox classes is the way they render themselves in a window. The ComboBox control uses a drop-down list, which means only one item can be selected at a time.

If you want to allow the user to type text in the combo box to select an item, you must set the IsEditable property to true, and you must make sure you are storing ordinary text-only ComboBoxItem objects or an object that provides a meaningful ToString() representation. For example, if you fill an editable combo box with Image objects, the text that appears in the upper portion is simply the fully qualified Image class name, which isn't much use.

One limitation of the ComboBox is the way it sizes itself when you use automatic sizing. The ComboBox widens itself to fit its content, which means that it changes size as you move from one item to the next. Unfortunately, there's no easy way to tell the ComboBox to take the size of its largest contained item. Instead, you may need to supply a hard-coded value for the Width property, which isn't ideal.

Range-Based Controls

WPF includes three controls that use the concept of a *range*. These controls take a numeric value that falls in between a specific minimum and maximum value. These controls—ScrollBar, ProgressBar, and Slider—all derive from the RangeBase class (which itself derives from the Control class). But although they share an abstraction (the range), they work quite differently.

The RangeBase class defines the properties shown in Table 6-4.

Table 6-4. *Properties of the RangeBase Class*

Name	Description
Value	The current value of the control (which must fall between the minimum and maximum). By default, it starts at 0. Contrary to what you might expect, Value isn't an integer—it's a double, so it accepts fractional values. You can react to the ValueChanged event if you want to be notified when the value is changed.
Maximum	The upper limit (the largest allowed value).
Minimum	The lower limit (the smallest allowed value).
SmallChange	The amount the Value property is adjusted up or down for a small change. The meaning of a "small change" depends on the control (and may not be used at all). For the ScrollBar and Slider, this is the amount the value changes when you use the arrow keys. For the ScrollBar, you can also use the arrow buttons at either end of the bar.
LargeChange	The amount the Value property is adjusted up or down for a large change. The meaning of a "large change" depends on the control (and may not be used at all). For the ScrollBar and Slider, this is the amount the value changes when you use the Page Up and Page Down keys or when you click the bar on either side of the thumb (which indicates the current position).

Ordinarily, there's no need to use the ScrollBar control directly. The higher-level ScrollViewer control, which wraps two ScrollBar controls, is typically much more useful. The Slider and ProgressBar are more practical, and are often useful on their own.

The Slider

The Slider is a specialized control that's occasionally useful—for example, you might use it to set numeric values in situations where the number itself isn't particularly significant. For example, it makes sense to set the volume in a media player by dragging the thumb in a slider bar from side to side. The general position of the thumb indicates the relative loudness (normal, quiet, or loud), but the underlying number has no meaning to the user.

The key Slider properties are defined in the RangeBase class. Along with these, you can use all the properties listed in Table 6-5.

Table 6-5. *Additional Properties in the Slider Class*

Name	Description
Orientation	Switches between a vertical and a horizontal slider.
Delay and Interval	Control how fast the thumb moves along the track when you click and hold down either side of the slider. Both are millisecond values. The Delay is the time before the thumb moves one (small change) unit after you click, and the Interval is the time before it moves again if you continue holding down the mouse button.
TickPlacement	Determines where the tick marks appear. (Tick marks are notches that appear near the bar to help you visualize the scale.) By default, TickPlacement is set to None, and no tick marks appear. If you have a horizontal slider, you can place the tick marks above (TopLeft) or below (BottomRight) the track. With a vertical slider, you can place them on the left (TopLeft) and right (BottomRight). (The TickPlacement names are a bit confusing because two values cover four possibilities, depending on the orientation of the slider.)
TickFrequency	Sets the interval in between ticks, which determines how many ticks appear. For example, you could place them every 5 numeric units, every 10, and so on.
Ticks	If you want to place ticks in specific, irregular positions, you can use the Ticks collection. Simply add one number (as a double) to this collection for each tick mark. For example, you could place ticks at the positions 1, 1.5, 2, and 10 on the scale by adding these numbers.