you considered earlier. For example, the GridSplitter in Figure 3-15 has a RowSpan of 2. As a result, it stretches over the entire column. If you didn't add this setting, it would appear only in the top row (where it's placed), *even though* dragging the splitter bar would resize the entire column.

- Initially, the GridSplitter is invisibly small. To make it usable, you need to give it a minimum size. In the case of a vertical splitter bar (like the one in Figure 3-15), you need to set VerticalAlignment to Stretch (so it fills the whole height of the available area) and Width to a fixed size (such as 10 device-independent units). In the case of a horizontal splitter bar, you need to set HorizontalAlignment to Stretch and set Height to a fixed size.
- The GridSplitter alignment also determines whether the splitter bar is horizontal (used to resize rows) or vertical (used to resize columns). In the case of a horizontal splitter bar, you should set VerticalAlignment to Center (which is the default value) to indicate that dragging the splitter resizes the rows that are above and below. In the case of a vertical splitter bar (like the one in Figure 3-15), you should set HorizontalAlignment to Center to resize the columns on either side.

■ **Note** You can change the resizing behavior using the ResizeDirection and ResizeBehavior properties of the GridSplitter. However, it's simpler to let this behavior depend entirely on the alignment settings, which is the default.

Dizzy yet? To reinforce these rules, it helps to take a look at the actual markup for the example shown in Figure 3-15. In the following listing, the GridSplitter details are highlighted:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition MinWidth="100"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition MinWidth="50"></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Button Grid.Row="0" Grid.Column="0" Margin="3">Left</Button>
  <Button Grid.Row="0" Grid.Column="2" Margin="3">Right</Button>
  <Button Grid.Row="1" Grid.Column="0" Margin="3">Left</Button>
  <Button Grid.Row="1" Grid.Column="2" Margin="3">Right</Button>

  <GridSplitter Grid.Row="0" Grid.Column="1" Grid.RowSpan="2"
   Width="3" VerticalAlignment="Stretch" HorizontalAlignment="Center"
   ShowsPreview="False"></GridSplitter>
</Grid>
```

■ **Tip** To create a successful GridSplitter, make sure you supply values for the VerticalAlignment, HorizontalAlignment, and Width (or Height) properties.

This markup includes one additional detail. When the GridSplitter is declared, the ShowsPreview property is set to false. As a result, when the splitter bar is dragged from one side to another, the columns are resized immediately. But if you set ShowsPreview to true, when you drag, you'll see a gray shadow follow your mouse pointer to show you where the split will be. The columns won't be resized until you release the mouse button. It's also possible to use the arrow keys to resize a GridSplitter once it receives focus.

The ShowsPreview isn't the only GridSplitter property that you can set. You can also adjust the DragIncrement property if you want to force the splitter to move in coarser "chunks" (such as 10 units at a time). If you want to control the maximum and minimum allowed sizes of the columns, you simply make sure the appropriate properties are set in the ColumnDefinitions section, as shown in the previous example.

■ **Tip** You can change the fill that's used for the GridSplitter so that it isn't just a shaded gray rectangle. The trick is to apply a fill using the Background property, which accepts simple colors and more complex brushes.

A Grid usually contains no more than a single GridSplitter. However, you can nest one Grid inside another, and if you do, each Grid may have its own GridSplitter. This allows you to create a window that's split into two regions (for example, a left and right pane) and then further subdivide one of these regions (say, the pane on the right) into more sections (such as a resizable top and bottom portion). Figure 3-16 shows an example.

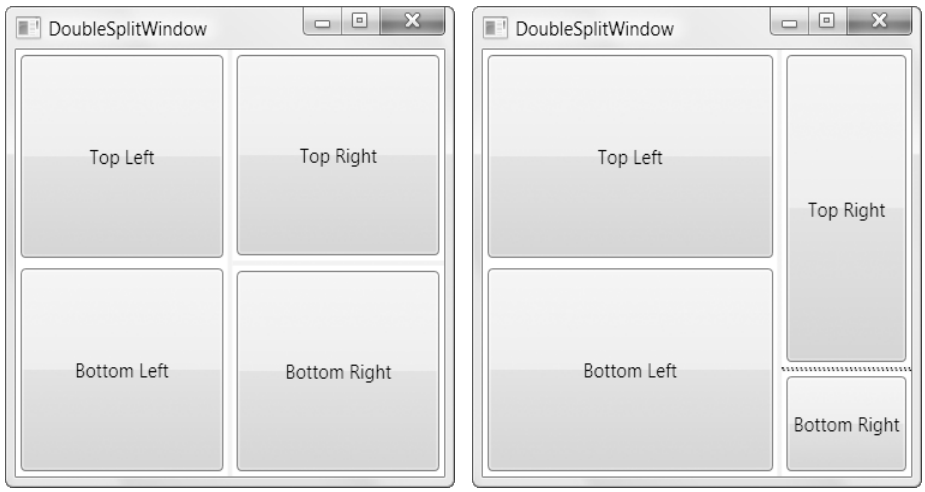

Figure 3-16. *Resizing a window with two splits*

Creating this window is fairly straightforward, although it's a chore to keep track of the three Grid containers that are involved: the overall Grid, the nested Grid on the left, and the nested Grid on the right. The only trick is to make sure the GridSplitter is placed in the correct cell and given the correct alignment. Here's the complete markup:

```
<!-- This is the Grid for the entire window. -->
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <!-- This is the nested Grid on the left.
       It isn't subdivided further with a splitter. -->
  <Grid Grid.Column="0" VerticalAlignment="Stretch">
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Button Margin="3" Grid.Row="0">Top Left</Button>
    <Button Margin="3" Grid.Row="1">Bottom Left</Button>
  </Grid>

  <!-- This is the vertical splitter that sits between the two nested
       (left and right) grids. -->
  <GridSplitter Grid.Column="1"
   Width="3" HorizontalAlignment="Center" VerticalAlignment="Stretch"
   ShowsPreview="False"></GridSplitter>

  <!-- This is the nested Grid on the right. -->
  <Grid Grid.Column="2">
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>

    <Button Grid.Row="0" Margin="3">Top Right</Button>
    <Button Grid.Row="2" Margin="3">Bottom Right</Button>

    <!-- This is the horizontal splitter that subdivides it into
         a top and bottom region.. -->
    <GridSplitter Grid.Row="1"
     Height="3" VerticalAlignment="Center" HorizontalAlignment="Stretch"
     ShowsPreview="False"></GridSplitter>
  </Grid>
</Grid>
```

■ **Tip** Remember, if a Grid has just a single row or column, you can leave out the RowDefinitions section. Also, elements that don't have their row position explicitly set are assumed to have a Grid.Row value of 0 and are placed in the first row. The same holds true for elements that don't supply a Grid.Column value.

Shared Size Groups

As you've seen, a Grid contains a collection of rows and columns, which are sized explicitly, proportionately, or based on the size of their children. There's one other way to size a row or a column—to match the size of another row or column. This works through a feature called *shared size groups*.

The goal of shared size groups is to keep separate portions of your user interface consistent. For example, you might want to size one column to fit its content and size another column to match that size exactly. However, the real benefit of shared size groups is to give the same proportions to separate Grid controls.

To understand how this works, consider the example shown in Figure 3-17. This window features two Grid objects—one at the top of the window (with three columns) and one at the bottom (with two columns). The leftmost column of the first Grid is sized proportionately to fit its content (a long text string). The leftmost column of the second Grid has exactly the same width, even though it contains less content. That's because it shares the same size group. No matter how much content you stuff in the first column of the first Grid, the first column of the second Grid stays synchronized.

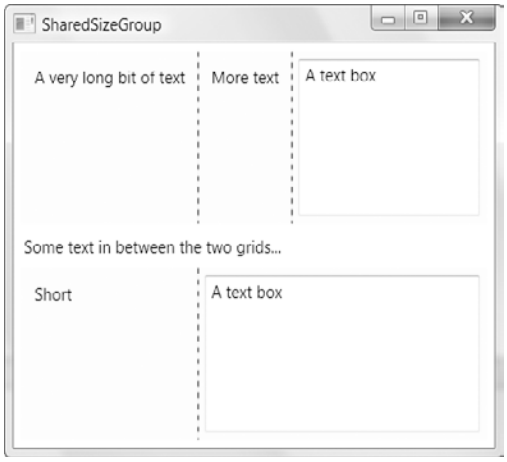

Figure 3-17. *Two grids that share a column definition*

As this example demonstrates, a shared column can be used in otherwise different grids. In this example, the top Grid has an extra column, and so the remaining space is divided differently. Similarly, the shared columns can occupy different positions, so you could create a relationship between the first column in one Grid and the second column in another. And obviously, the columns can host completely different content.

When you use a shared size group, it's as if you've created one column (or row) definition, which is reused in more than one place. It's not a simple one-way copy of one column to another. You can test this with the previous example by changing the content in the shared column of the second Grid. Now, the column in the first Grid will be lengthened to match (Figure 3-18).

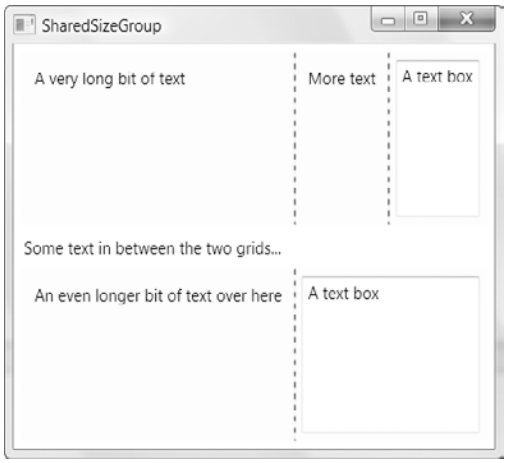

Figure 3-18. *Shared-size columns remain synchronized*

You can even add a GridSplitter to one of the Grid objects. As the user resizes the column in one Grid, the shared column in the other Grid will follow along, resizing itself at the same time.

Creating a shared group is easy. You simply need to set the SharedSizeGroup property on both columns, using a matching string. In the current example, both columns use a group named TextLabel:

```
<Grid Margin="3" Background="LightYellow" ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="TextLabel"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Label Margin="5">A very long bit of text</Label>
  <Label Grid.Column="1" Margin="5">More text</Label>
  <TextBox Grid.Column="2" Margin="5">A text box</TextBox>
</Grid>
...
<Grid Margin="3" Background="LightYellow"  ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="TextLabel"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Label Margin="5">Short</Label>
  <TextBox Grid.Column="1" Margin="5">A text box</TextBox>
</Grid>
```

There's one other detail. Shared size groups aren't global to your entire application because more than one window might inadvertently use the same name. You might assume that shared size groups are limited to the current window, but WPF is even more stringent than that. To share a group, you need to explicitly set the attached Grid.IsSharedSizeScope property to true on a container somewhere upstream that holds the Grid objects with the shared column. In the current example, the top and bottom Grid are wrapped in another Grid that accomplishes this purpose, although you could just as easily use a different container such as a DockPanel or StackPanel.

Here's the markup for the top-level Grid:

```
<Grid Grid.IsSharedSizeScope="True" Margin="3">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>

  <Grid Grid.Row="0" Margin="3" Background="LightYellow" ShowGridLines="True">
    ...
  </Grid>
  <Label Grid.Row="1" >Some text in between the two grids...</Label>
  <Grid Grid.Row="2" Margin="3" Background="LightYellow" ShowGridLines="True">
    ...
  </Grid>
</Grid>
```

■ **Tip** You could use a shared size group to synchronize a separate Grid with column headers. The width of each column can then be determined by the content in the column, which the header will share. You could even place a GridSplitter in the header, which the user could drag to resize the header and the entire column underneath.

The UniformGrid

There is a grid that breaks all the rules you've learned about so far: the UniformGrid. Unlike the Grid, the UniformGrid doesn't require (or even support) predefined columns and rows. Instead, you simply set the Rows and Columns properties to set its size. Each cell is always the same size because the available space is divided equally. Finally, elements are placed into the appropriate cell based on the order in which you define them. There are no attached Row and Column properties, and no blank cells.

Here's an example that fills a UniformGrid with four buttons:

```
<UniformGrid Rows="2" Columns="2">
  <Button>Top Left</Button>
  <Button>Top Right</Button>
  <Button>Bottom Left</Button>
  <Button>Bottom Right</Button>
</UniformGrid>
```

The UniformGrid is used far less frequently than the Grid. The Grid is an all-purpose tool for creating window layouts from the simple to the complex. The UniformGrid is a much more specialized layout container that's primarily useful when quickly laying out elements in a rigid grid (for example, when building a playing board for certain games). Many WPF programmers will never use the UniformGrid.

Coordinate-Based Layout with the Canvas

The only layout container you haven't considered yet is the Canvas. It allows you to place elements using exact coordinates, which is a poor choice for designing rich data-driven forms and standard dialog boxes, but it's a valuable tool if you need to build something a little different (such as a drawing surface for a diagramming tool). The Canvas is also the most lightweight of the layout containers. That's because it doesn't include any complex layout logic to negotiate the sizing preferences of its children. Instead, it simply lays them all out at the position they specify, with the exact size they want.

To position an element on the Canvas, you set the attached Canvas.Left and Canvas.Top properties. Canvas.Left sets the number of units between the left edge of your element and the left edge of the Canvas. Canvas.Top sets the number of units between the top of your element and the top of the Canvas. As always, these values are set in device-independent units, which line up with ordinary pixels exactly when the system DPI is set to 96 dpi.

■ **Note** Alternatively, you can use Canvas.Right instead of Canvas.Left to space an element from the right edge of the Canvas, and Canvas.Bottom instead of Canvas.Top to space it from the bottom. You just can't use both Canvas.Right and Canvas.Left at once, or both Canvas.Top and Canvas.Bottom.

Optionally, you can size your element explicitly using its Width and Height properties. This is more common when using the Canvas than it is in other panels because the Canvas has no layout logic of its own. (And often, you'll use the Canvas when you need precise control over how a combination of elements is arranged.) If you don't set the Width and Height properties, your element will get its desired size—in other words, it will grow just large enough to fit its content.

Here's a simple Canvas that includes four buttons:

```
<Canvas>
  <Button Canvas.Left="10" Canvas.Top="10">(10,10)</Button>
  <Button Canvas.Left="120" Canvas.Top="30">(120,30)</Button>
  <Button Canvas.Left="60" Canvas.Top="80" Width="50" Height="50">
   (60,80)</Button>
  <Button Canvas.Left="70" Canvas.Top="120" Width="100" Height="50">
   (70,120)</Button>
</Canvas>
```

Figure 3-19 shows the result.

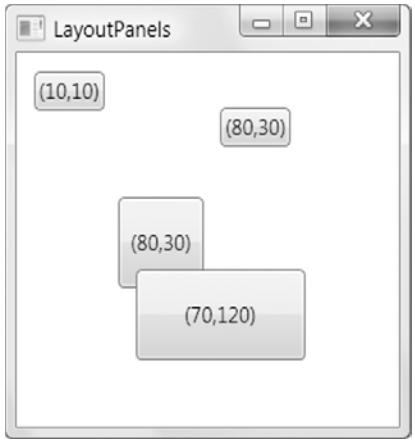

Figure 3-19. *Explicitly positioned buttons in a Canvas*

If you resize the window, the Canvas stretches to fill the available space, but none of the controls in the Canvas moves or changes size. The Canvas doesn't include any of the anchoring or docking features that were provided with coordinate layout in Windows Forms. Part of the reason for this gap is to keep the Canvas lightweight. Another reason is to prevent people from using the Canvas for purposes for which it's not intended (such as laying out a standard user interface).

Like any other layout container, the Canvas can be nested inside a user interface. That means you can use the Canvas to draw some detailed content in a portion of your window, while using more standard WPF panels for the rest of your elements.

■ **Tip** If you use the Canvas alongside other elements, you may want to consider setting its ClipToBounds to true. That way, elements inside the Canvas that stretch beyond its bounds are clipped off at the edge of the Canvas. (This prevents them from overlapping other elements elsewhere in your window.) All the other layout containers always clip their children to fit, regardless of the ClipToBounds setting.

Z-Order

If you have more than one overlapping element, you can set the attached Canvas.ZIndex property to control how they are layered.

Ordinarily, all the elements you add have the same ZIndex—0. When elements have the same ZIndex, they're displayed in the same order that they exist in Canvas.Children collection, which is based on the order that they're defined in the XAML markup. Elements declared later in the markup—such as button (70,120)—are displayed overtop of elements that are declared earlier—such as button (120,30).

However, you can promote any element to a higher level by increasing its ZIndex. That's because higher ZIndex elements *always* appear over lower ZIndex elements. Using this technique, you could reverse the layering in the previous example:

```
<Button Canvas.Left="60" Canvas.Top="80" Canvas.ZIndex="1" Width="50" Height="50">
 (60,80)</Button>
<Button Canvas.Left="70" Canvas.Top="120" Width="100" Height="50">
 (70,120)</Button>
```

■ **Note** The actual values you use for the Canvas.ZIndex property have no meaning. The important detail is how the ZIndex value of one element compares to the ZIndex value of another. You can set the ZIndex using any positive or negative integer.

The ZIndex property is particularly useful if you need to change the position of an element programmatically. Just call Canvas.SetZIndex() and pass in the element you want to modify and the new ZIndex you want to apply. Unfortunately, there is no BringToFront() or SendToBack() method—it's up to you to keep track of the highest and lowest ZIndex values if you want to implement this behavior.

The InkCanvas

WPF also includes an InkCanvas element that's similar to the Canvas in some respects (and wholly different in others). Like the Canvas, the InkCanvas defines four attached properties that you can apply to child elements for coordinate-based positioning (Top, Left, Bottom, and Right). However, the underlying plumbing is quite a bit different—in fact, the InkCanvas doesn't derive from Canvas or even from the base Panel class. Instead, it derives directly from FrameworkElement.

The primary purpose of the InkCanvas is to allow *stylus* input. The stylus is the pen-like input device that's used in tablet PCs. However, the InkCanvas works with the mouse in the same way as it works with the stylus. Thus, a user can draw lines or select and manipulate elements in the InkCanvas using the mouse.

The InkCanvas actually holds two collections of child content. The familiar Children collection holds arbitrary elements, just as with the Canvas. Each element can be positioned based on the Top, Left, Bottom, and Right properties. The Strokes collection holds System.Windows.Ink.Stroke objects, which represent graphical input that the user has drawn in the InkCanvas. Each line or curve that the user draws becomes a separate Stroke object. Thanks to these dual collections, you can use the InkCanvas to let the user annotate content (stored in the Children collection) with strokes (stored in the Strokes collection).

For example, Figure 3-20 shows an InkCanvas that contains a picture that has been annotated with extra strokes. Here's the markup for the InkCanvas in this example, which defines the image:

```
<InkCanvas Name="inkCanvas" Background="LightYellow"
 EditingMode="Ink">
  <Image Source="office.jpg" InkCanvas.Top="10" InkCanvas.Left="10"
   Width="287" Height="319"></Image>
</InkCanvas>
```

The strokes are drawn at runtime by the user.

Figure 3-20. *Adding strokes in an InkCanvas*

The InkCanvas can be used in some significantly different ways, depending on the value you set for the InkCanvas.EditingMode property. Table 3-5 lists all your options.

Table 3-5. *Values of the InkCanvasEditingMode Enumeration*

Name	Description
Ink	The InkCanvas allows the user to draw annotations. This is the default mode. When the user draws with the mouse or stylus, a stroke is drawn.
GestureOnly	The InkCanvas doesn't allow the user to draw stroke annotations but pays attention to specific predefined *gestures* (such as dragging the stylus in one direction, or scratching out content). The full list of recognized gestures is listed by the System.Windows.Ink.ApplicationGesture enumeration.
InkAndGesture	The InkCanvas allows the user to draw stroke annotations and also recognizes predefined gestures.

Name	Description
EraseByStroke	The InkCanvas erases a stroke when it's clicked. If the user has a stylus, he can switch to this mode by using the back end of the stylus. (You can determine the current mode using the read-only ActiveEditingMode property, and you can change the mode used for the back end of the stylus by changing the EditingModeInverted property.)
EraseByPoint	The InkCanvas erases a portion of a stroke (a point in a stroke) when that portion is clicked.
Select	The InkCanvas allows the user to select elements that are stored in the Children collection. To select an element, the user must click it or drag a selection "lasso" around it. Once an element is selected, it can be moved, resized, or deleted.
None	The InkCanvas ignores mouse and stylus input.

The InkCanvas raises events when the editing mode changes (ActiveEditingModeChanged), a gesture is detected in GestureOnly or InkAndGesture mode (Gesture), a stroke is drawn (StrokeCollected), a stroke is erased (StrokeErasing and StrokeErased), and an element is selected or changed in Select mode (SelectionChanging, SelectionChanged, SelectionMoving, SelectionMoved, SelectionResizing, and SelectionResized). The events that end in *ing* represent an action that is about to take place but can be canceled by setting the Cancel property of the EventArgs object.

In Select mode, the InkCanvas provides a fairly capable design surface for dragging content around and manipulating it. Figure 3-21 shows a Button control in an InkCanvas as it's being selected (on the left) and then repositioned and resized (on the right).

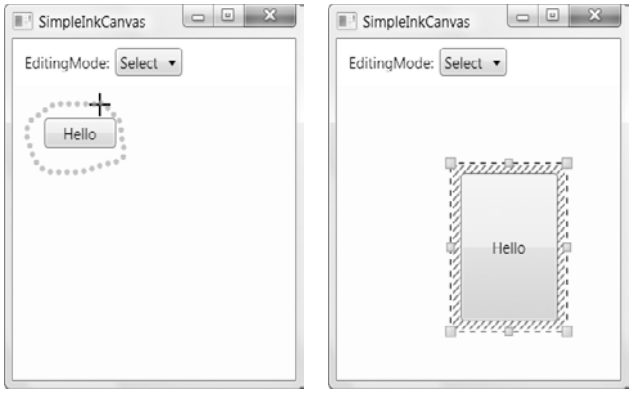

Figure 3-21. *Moving and resizing an element in the InkCanvas*

As interesting as Select mode is, it isn't a perfect fit if you're building a drawing or diagramming tool. You'll see a better example of how to create a custom drawing surface in Chapter 14.

Layout Examples

You've now spent a considerable amount of time poring over the intricacies of the WPF layout containers. With this low-level knowledge in mind, it's worth looking at a few complete layout examples. Doing so will give you a better sense of how the various WPF layout concepts (such as size-to-content, stretch, and nesting) work in real-world windows.

A Column of Settings

Layout containers such as the Grid make it dramatically easier to create an overall structure to a window. For example, consider the window with settings shown in Figure 3-22. This window arranges its individual components—labels, text boxes, and buttons—into a tabular structure.

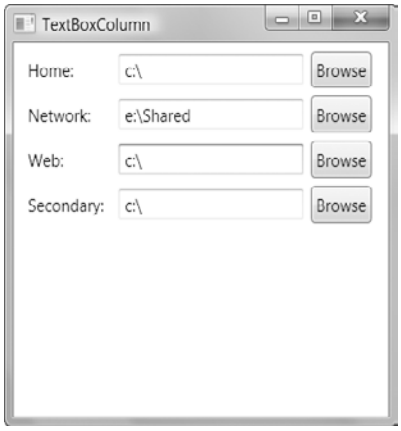

Figure 3-22. *Folder settings in a column*

To create this table, you begin by defining the rows and columns of the grid. The rows are easy enough—each one is simply sized to the height of the containing content. That means the entire row will get the height of the largest element, which in this case is the Browse button in the third column.

```
<Grid Margin="3,3,10,3">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  ...
```

Next, you need to create the columns. The first and last columns are sized to fit their content (the label text and the Browse button, respectively). The middle column gets all the remaining room, which means it grows as the window is resized larger, giving you more room to see the selected folder. (If you

want this stretching to top out at some extremely wide maximum value, you can use the MaxWidth property when defining the column, just as you do with individual elements.)

```
...
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto"></ColumnDefinition>
  <ColumnDefinition Width="*"></ColumnDefinition>
  <ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
...
```

■ **Tip** The Grid needs some minimum space—enough to fit the full label text, the browse button, and a few pixels in the middle column to show the text box. If you shrink the containing window to be smaller than this, some content will be cut off. As always, it makes sense to use the MinWidth and MinHeight properties on the window to prevent this from occurring.

Now that you have your basic structure, you simply need to slot the elements into the right cells. However, you also need to think carefully about margins and alignment. Each element needs a basic margin (a good value is 3 units) to give some breathing room. In addition, the label and text box need to be centered vertically because they aren't as tall as the Browse button. Finally, the text box needs to use automatic sizing mode, so it stretches to fit the entire column.

Here's the markup you need to define the first row in the grid:

```
  ...
  <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>
```

You can repeat this markup to add all your rows by simply incrementing the value of the Grid.Row attribute.

One fact that's not immediately obvious is how flexible this window is because of the use of the Grid control. None of the individual elements—the labels, text boxes, and buttons—have hard-coded positions or sizes. As a result, you can quickly make changes to the entire grid simply by tweaking the ColumnDefinition elements. Furthermore, if you add a row that has longer label text (necessitating a wider first column), the entire grid is adjusted to be consistent, including the rows that you've already added. And if you want to add elements in between the rows—such as separator lines to divide different sections of the window—you can keep the same columns but use the ColumnSpan property to stretch a single element over a larger area.

Dynamic Content

As the column of settings demonstrates, windows that use the WPF layout containers are easy to change and adapt as you revise your application. This flexibility doesn't just benefit you at design time. It's also a great asset if you need to display content that changes dramatically.

One example is *localized text*—text that appears in your user interface and needs to be translated into different languages for different geographic regions. In old-style coordinate-based applications, changing the text can wreak havoc in a window, particularly because a short amount of English text becomes significantly larger in many languages. Even if elements are allowed to resize themselves to fit larger text, doing so often throws off the whole balance of a window.

Figure 3-23 demonstrates how this isn't the case when you use the WPF layout containers intelligently. In this example, the user interface has a short text and a long text option. When the long text is used, the buttons that contain the text are resized automatically and other content is bumped out of the way. And because the resized buttons share the same layout container (in this case, a table column), that entire section of the user interface is resized. The end result is that all buttons keep a consistent size—the size of the largest button.

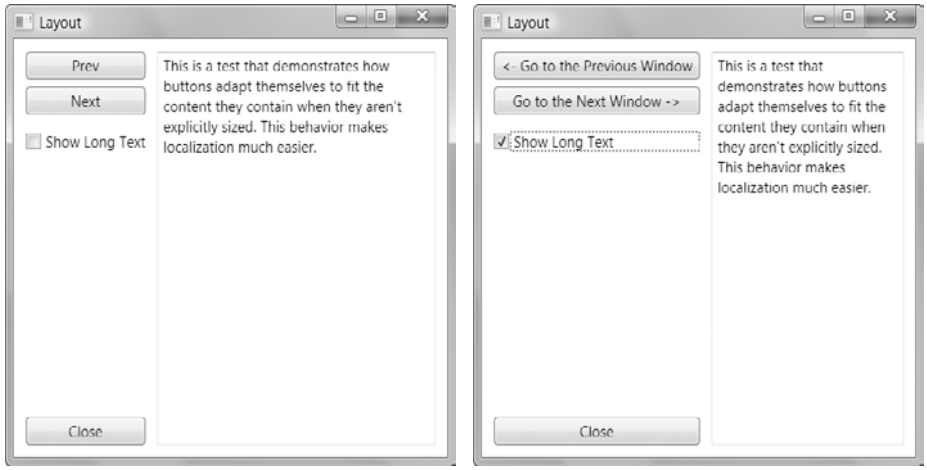

Figure 3-23. *A self-adjusting window*

To make this work, the window is carved into a table with two columns and two rows. The column on the left takes the resizable buttons, while the column on the right takes the text box. The bottom row is used for the Close button. It's kept in the same table so that it resizes along with the top row.

Here's the complete markup:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <StackPanel Grid.Row="0" Grid.Column="0">
    <Button Name="cmdPrev" Margin="10,10,10,3">Prev</Button>
    <Button Name="cmdNext" Margin="10,3,10,3">Next</Button>
```

```
    <CheckBox Name="chkLongText" Margin="10,10,10,10"
     Checked="chkLongText_Checked" Unchecked="chkLongText_Unchecked">
     Show Long Text</CheckBox>
  </StackPanel>
  <TextBox Grid.Row="0" Grid.Column="1" Margin="0,10,10,10"
   TextWrapping="WrapWithOverflow" Grid.RowSpan="2">This is a test that demonstrates
    how buttons adapt themselves to fit the content they contain when they aren't
    explicitly sized. This behavior makes localization much easier.</TextBox>
  <Button Grid.Row="1" Grid.Column="0" Name="cmdClose"
    Margin="10,3,10,10">Close</Button>
</Grid>
```

The event handlers for the CheckBox aren't shown here. They simply change the text in the two buttons.

A Modular User Interface

Many of the layout containers gracefully "flow" content into the available space, like the StackPanel, DockPanel, and WrapPanel. One advantage of this approach is that it allows you to create truly modular interfaces. In other words, you can plug in different panels with the appropriate user interface sections you want to show and leave out those that don't apply. The entire application can shape itself accordingly, somewhat like a portal site on the Web.

Figure 3-24 demonstrates this. It places several separate panels into a WrapPanel. The user can choose which of these panels are visible using the check boxes at the top of the window.

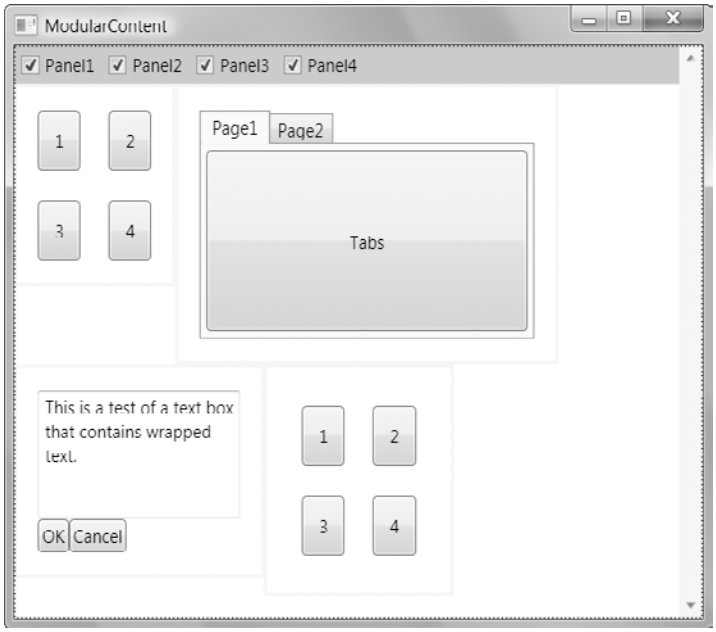

Figure 3-24. *A series of panels in a WrapPanel*

■ **Note** Although you can set the background of a layout panel, you can't set a border around it. This example overcomes that limitation by wrapping each panel in a Border element that outlines the exact dimensions.

As different panels are hidden, the remaining panels reflow themselves to fit the available space (and the order in which they're declared). Figure 3-25 shows a different permutation of panels.

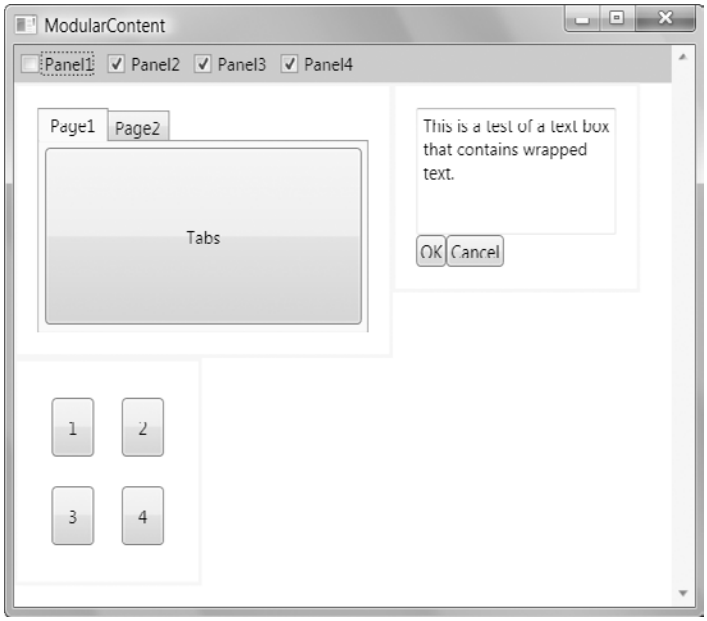

Figure 3-25. *Hiding some panels*

To hide and show the individual panels, a small bit of code handles check box clicks. Although you haven't considered the WPF event handling model in any detail (Chapter 5 has the full story), the trick is to set the Visibility property:

```
panel.Visibility = Visibility.Collapsed;
```

The Visibility property is a part of the base UIElement class and is therefore supported by just about everything you'll put in a WPF window. It takes one of three values, from the System.Windows.Visibility enumeration, as listed in Table 3-6.

Table 3-6. *Values of the Visibility Enumeration*

Value	Description
Visible	The element appears as normal in the window.
Collapsed	The element is not displayed and doesn't take up any space.
Hidden	The element is not displayed, but the space it would otherwise use is still reserved. (In other words, there's a blank space where it would have appeared). This setting is handy if you need to hide and show elements without changing the layout and the relative positioning of the elements in the rest of your window.

■ **Tip** You can use the Visibility property to dynamically tailor a variety of interfaces. For example, you could make a collapsible pane that can appear at the side of your window. All you need to do is wrap all the contents of that pane in some sort of layout container and set its Visibility property to suit. The remaining content will be rearranged to fit the available space.

The Last Word

In this chapter, you took a detailed tour of the new WPF layout model and learned how to place elements in stacks, grids, and other arrangements. You built more complex layouts using nested combinations of the layout containers, and you threw the GridSplitter into the mix to make resizable split windows. And all along, you kept close focus on the reasons for this dramatic change—namely, the benefits you'll get when maintaining, enhancing, and localizing your user interface.

The layout story is still far from over. In the following chapters, you'll see many more examples that use the layout containers to organize groups of elements. You'll also learn about a few additional features that let you arrange content in a window:

- **Specialized containers.** The ScrollViewer, TabItem, and Expander controls give you the ability to scroll content, place it in separate tabs, and collapse it out of sight. Unlike the layout panels, these containers can hold only a single piece of content. However, you can easily use them in concert with a layout panel to get exactly the effect you need. You'll try these containers in Chapter 6.
- **The Viewbox.** Need a way to resize graphical content (such as images and vector drawings)? The Viewbox is yet another specialized container that can help you out, and it has built-in scaling. You'll take your first look at the Viewbox in Chapter 12.
- **Text layout.** WPF adds new tools for laying out large blocks of styled text. You can use floating figures and lists and use paging, columns, and sophisticated wrapping intelligence to get remarkably polished results. You'll see how in Chapter 28.

CHAPTER 4

Dependency Properties

Every .NET programmer is familiar with *properties* and *events*, which are core parts of .NET's object abstraction. Few would expect WPF, a user interface technology, to change either of these fundamentals. But surprisingly enough, that's exactly what WPF does.

In this chapter, you'll learn how WPF replaces ordinary .NET properties with a higher-level *dependency property* feature. Dependency properties use more efficient storage and support additional features such as change notification and property value inheritance (the ability to propagate default values down the element tree). Dependency properties are also the basis for a number of key WPF features, including animation, data binding, and styles. Fortunately, even though the plumbing has changed, you can read and set dependency properties in code in exactly the same way as traditional .NET properties.

In the following pages, you'll take a close look at dependency properties. You'll see how they're defined, registered, and consumed. You'll also learn what features they support and what problems they solve.

Note Understanding dependency properties requires a heavy dose of theory, and you might not want to slog through just yet. If you can't wait to get started building an application, feel free to skip ahead to the following chapters and then return to this one when you need a deeper understanding of how WPF ticks and you want to build dependency properties of your own.

Understanding Dependency Properties

Dependency properties are a completely new implementation of properties—one that has a significant amount of added value. You need dependency properties to plug into core WPF features such as animation, data binding, and styles.

Most of the properties that are exposed by WPF elements are dependency properties. In all the examples you've seen up to this point, you've been using dependency properties without realizing it. That's because dependency properties are designed to be consumed in the same way as normal properties.

However, dependency properties are *not* normal properties. It's comforting to think of a dependency property as a normal property (defined in the typical .NET fashion) with a set of WPF features added on. Conceptually, dependency features behave this way, but that's not how they're implemented behind the scenes. The simple reason why is performance. If the designers of WPF simply

added extra features on top of the .NET property system, they'd need to create a complex, bulky layer for your code to travel through. Ordinary properties could not support all the features of dependency properties without this extra overhead.

Dependency properties are a WPF-specific creation. However, the dependency properties in the WPF libraries are always wrapped by ordinary .NET property procedures. This makes them usable in the normal way, even with code that has no understanding of the WPF dependency property system. It seems odd to think of an older technology wrapping a newer one, but that's how WPF is able to change a fundamental ingredient such as properties without disrupting the rest of the .NET world.

Defining a Dependency Property

You'll spend much more time using dependency properties than creating them. However, there are still many reasons that you'll need to create your own dependency properties. Obviously, they're a key ingredient if you're designing a custom WPF element. However, they're also required in some cases if you want to add data binding, animation, or another WPF feature to a portion of code that wouldn't otherwise support it. Creating a dependency property isn't difficult, but the syntax takes a little getting used to. It's thoroughly different from creating an ordinary .NET property.

■ **Note** You can add dependency properties only to dependency objects—classes that derive from DependencyObject. Fortunately, most of the key pieces of WPF infrastructure derive indirectly from DependencyObject, with the most obvious example being elements.

The first step is to define an object that *represents* your property. This is an instance of the DependencyProperty class. The information about your property needs to be available all the time, and possibly even shared among classes (as is common with WPF elements). For that reason, your DependencyProperty object must be defined as a static field in the associated class.

For example, the FrameworkElement class defines a Margin property that all elements share. Unsurprisingly, Margin is a dependency property. That means it's defined in the FrameworkElement class like this:

```
public class FrameworkElement: UIElement, ...
{
    public static readonly DependencyProperty MarginProperty;

    ...
}
```

By convention, the field that defines a dependency property has the name of the ordinary property, plus the word *Property* at the end. That way, you can separate the dependency property definition from the name of the actual property. The field is defined with the readonly keyword, which means it can be set only in the static constructor for the FrameworkElement, which is the task you'll undertake next.

Registering a Dependency Property

Defining the DependencyProperty object is just the first step. For it to become usable, you need to register your dependency property with WPF. This step needs to be completed before any code uses the property, so it must be performed in a static constructor for the associated class.

WPF ensures that DependencyProperty objects can't be instantiated directly, because the DependencyProperty class has no public constructor. Instead, a DependencyObject instance can be created only using the static DependencyProperty.Register() method. WPF also ensures that DependencyProperty objects can't be changed after they're created, because all DependencyProperty members are read-only. Instead, their values must be supplied as arguments to the Register() method.

The following code shows an example of how a DependencyProperty must be created. Here, the FrameworkElement class uses a static constructor to initialize the MarginProperty:

```
static FrameworkElement()
{
    FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata(
      new Thickness(), FrameworkPropertyMetadataOptions.AffectsMeasure);

    MarginProperty = DependencyProperty.Register("Margin",
      typeof(Thickness), typeof(FrameworkElement), metadata,
      new ValidateValueCallback(FrameworkElement.IsMarginValid));
    ...
}
```

There are two steps involved in registering a dependency property. First, you create a FrameworkPropertyMetadata object that indicates what services you want to use with your dependency property (such as support for data binding, animation, and journaling). Next, you register the property by calling the static DependencyProperty.Register() method. At this point, you are responsible for supplying a few key ingredients:

- The property name (Margin in this example)
- The data type used by the property (the Thickness structure in this example)
- The type that owns this property (the FrameworkElement class in this example)
- Optionally, a FrameworkPropertyMetadata object with additional property settings
- Optionally, a callback that performs validation for the property

The first three details are all straightforward. The FrameworkPropertyMetadata object and the validation callback are more interesting.

You use the FrameworkPropertyMetadata to configure additional features for your dependency property. Most of the properties of the FrameworkPropertyMetadata class are simple Boolean flags that you set to flip on a feature. (The default value for each Boolean flag is false.) A few are callbacks that point to custom methods that you create to perform a specific task. One—FrameworkPropertyMetadata.DefaultValue—sets the default value that WPF will apply when the property is first initialized. Table 4-1 lists all the FrameworkPropertyMetadata properties.