CHAPTER 3

Layout

Half the battle in any user interface design is organizing the content in a way that's attractive, practical, and flexible. But the real challenge is making sure that your layout can adapt itself gracefully to different window sizes.

In WPF, you shape layout by using different *containers*. Each container has its own layout logic—some stack elements, others arrange them in a grid of invisible cells, and so on. But coordinate-based layout is strongly discouraged in WPF. Instead, the emphasis is on creating more-flexible layouts that can adapt to changing content, different languages, and a variety of window sizes. For many developers moving to WPF, the layout system is a great surprise—and the first real challenge.

In this chapter, you'll see how the WPF layout model works, and you'll begin using the basic layout containers. You'll also consider several common layout examples—everything from a basic dialog box to a resizable split window—in order to learn the fundamentals of WPF layout.

Understanding Layout in WPF

The WPF layout model represents a dramatic shift in the way Windows developers approach user interfaces. In the past, Windows developers used strict coordinate-based layouts to anchor controls into place. Although this approach is possible in WPF, it's far less common. Most applications will use weblike *flow* layouts, which let controls grow and bump other controls out of the way. This model allows developers to create resolution-independent, size-independent interfaces that scale well on different monitors, adjust themselves when content changes, and handle the transition to other languages effortlessly. However, before you can take advantage of this system, you need to understand a bit more about its underlying concepts and assumptions.

The WPF Layout Philosophy

A WPF window can hold only a single element. To fit in more than one element and create a more practical user interface, you need to place a container in your window and then add other elements to that container.

Note This limitation stems from the fact that the Window class is derived from ContentControl, which you'll study more closely in Chapter 6.

In WPF, layout is determined by the container that you use. Although there are several containers to choose from, the "ideal" WPF window follows a few key principles:

- *Elements (such as controls) should not be explicitly sized.* Instead, they grow to fit their content. For example, a button expands as you add more text. You can limit controls to acceptable sizes by setting a maximum and minimum size.
- *Elements do not indicate their position with screen coordinates.* Instead, they are arranged by their container based on their size, order, and (optionally) other information that's specific to the layout container. If you need to add whitespace between elements, you use the Margin property.

■ **Tip** Hard-coded sizes and positions are evil because they limit your ability to localize your interface, and they make it much more difficult to deal with dynamic content.

- *Layout containers "share" the available space among their children.* They attempt to give each element its preferred size (based on its content) if the space is available. They can also distribute extra space to one or more children.
- *Layout containers can be nested.* A typical user interface begins with the Grid, WPF's most capable container, and contains other layout containers that arrange smaller groups of elements, such as captioned text boxes, items in a list, icons on a toolbar, a column of buttons, and so on.

Although there are exceptions to these rules, they reflect the overall design goals of WPF. In other words, if you follow these guidelines when you build a WPF application, you'll create a better, more flexible user interface. If you break these rules, you'll end up with a user interface that isn't well suited to WPF and is much more difficult to maintain.

The Layout Process

WPF layout takes place in two stages: a measure stage and an arrange stage. In the *measure stage*, the container loops through its child elements and asks them to provide their preferred size. In the *arrange stage*, the container places the child elements in the appropriate position.

Of course, an element can't always get its preferred size—sometimes the container isn't large enough to accommodate it. In this case, the container must truncate the offending element to fit the visible area. As you'll see, you can often avoid this situation by setting a minimum window size.

■ **Note** Layout containers don't provide any scrolling support. Instead, scrolling is provided by a specialized content control—the ScrollViewer—that can be used just about anywhere. You'll learn about the ScrollViewer in Chapter 6.

The Layout Containers

All the WPF layout containers are panels that derive from the abstract System.Windows.Controls.Panel class (see Figure 3-1). The Panel class adds a small set of members, including the three public properties that are detailed in Table 3-1.

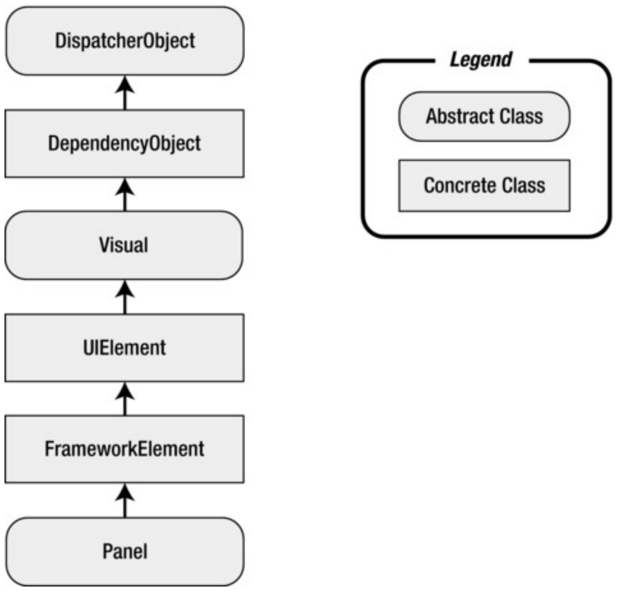


Figure 3-1. *The hierarchy of the Panel class*

Table 3-1. *Public Properties of the Panel Class*

Name	Description
Background	The brush that's used to paint the panel background. You must set this property to a non-null value if you want to receive mouse events. (If you want to receive mouse events but you don't want to display a solid background, just set the background color to Transparent.) You'll learn more about basic brushes in Chapter 6 (and about advanced brushes in Chapter 12).
Children	The collection of items that's stored in the panel. This is the first level of items—in other words, these items may themselves contain more items.
IsItemsHost	A Boolean value that's true if the panel is being used to show the items that are associated with an ItemsControl (such as the nodes in a TreeViewor the list entries in a ListBox). Most of the time you won't even be aware that a list control is using a behind-the-scenes panel to manage the layout of its items. However, this detail becomes more important if you want to create a customized list that lays out children in a different way (for example, a ListBox that tiles images). You'll use this technique in Chapter 20.

■ **Note** The Panel class also has a bit of internal plumbing you can use if you want to create your own layout container. Most notably, you can override the MeasureOverride() and ArrangeOverride methods inherited from FrameworkElement to change the way the panel handles the measure stage and the arrange stage when organizing its child elements. You'll learn how to create a custom panel in Chapter 18.

On its own, the base Panel class is nothing but a starting point for other, more-specialized classes. WPF provides a number of Panel-derived classes that you can use to arrange layout. The most fundamental of these are listed in Table 3-2. As with all WPF controls and most visual elements, these classes are found in the System.Windows.Controls namespace.

Table 3-2. *Core Layout Panels*

Name	Description
StackPanel	Places elements in a horizontal or vertical stack. This layout container is typically used for small sections of a larger, more complex window.
WrapPanel	Places elements in a series of wrapped lines. In horizontal orientation, the WrapPanel lays items out in a row from left to right and then onto subsequent lines. In vertical orientation, the WrapPanel lays out items in a top-to-bottom column and then uses additional columns to fit the remaining items.
DockPanel	Aligns elements against an entire edge of the container.
Grid	Arranges elements in rows and columns according to an invisible table. This is one of the most flexible and commonly used layout containers.
UniformGrid	Places elements in an invisible table but forces all cells to have the same size. This layout container is used infrequently.
Canvas	Allows elements to be positioned absolutely by using fixed coordinates. This layout container is the most similar to old-fashioned Windows Forms applications, without the anchoring and docking features. As a result, it's an unsuitable choice for a resizable window unless you're willing to do a fair bit of work.

Along with these core containers, you'll encounter several more specialized panels in various controls. These include panels that are dedicated to holding the child items of a particular control, such as TabPanel (the tabs in a TabControl), ToolbarPanel (the buttons in a Toolbar), and ToolbarOverflowPanel (the commands in a Toolbar's overflow menu). There's also a VirtualizingStackPanel, which data-bound list controls use to dramatically reduce their overhead, and an InkCanvas, which is similar to the Canvas but has support for handling stylus input on the TabletPC. (For example, depending on the mode you choose, the InkCanvas supports drawing with the pointer to select onscreen elements. And although it's a little counterintuitive, you can use the InkCanvas with an ordinary computer and a mouse.) You'll learn about the InkCanvas in this chapter, and you'll take a closer look at the VirtualizingStackPanel in Chapter 19. You'll learn about the other specialized panels when you consider the related control, elsewhere in this book.

Simple Layout with the StackPanel

The StackPanel is one of the simplest layout containers. It simply stacks its children in a single row or column.

For example, consider this window, which contains a stack of four buttons:

```
<Window x:Class="Layout.SimpleStack"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Layout" Height="223" Width="354"
    >
  <StackPanel>
    <Label>A Button Stack</Label>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </StackPanel>
</Window>
```

Figure 3-2 shows the window that results.

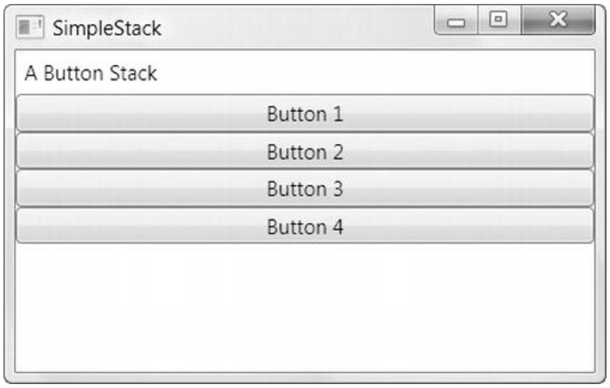


Figure 3-2. *The StackPanel in action*

ADDING A LAYOUT CONTAINER IN VISUAL STUDIO

It's relatively easy to create this example by using the designer in Visual Studio. Begin by deleting the root Grid element (if it's there). Then, drag a StackPanel into the window. Next, drag the other elements (the label and four buttons) into the window, in the top-to-bottom order you want. If you want to rearrange the order of elements in the StackPanel, you can simply drag any one to a new position.

Although we won't spend much time discussing Visual Studio's design-time support in this book, it's improved greatly since the first versions of WPF. For example, Visual Studio no longer assigns names to every new control you add in the designer, and it no longer adds hard-coded Width and Height values, unless you manually resize the control.

By default, a StackPanel arranges elements from top to bottom, making each one as tall as is necessary to display its content. In this example, that means the labels and buttons are sized just large enough to comfortably accommodate the text inside. All elements are stretched to the full width of the StackPanel, which is the width of the window. If you widen the window, the StackPanel widens as well, and the buttons stretch themselves to fit.

The StackPanel can also be used to arrange elements horizontally by setting the Orientation property:

```
<StackPanel Orientation="Horizontal">
```

Now elements are given their minimum width (wide enough to fit their text) and are stretched to the full height of the containing panel. Depending on the current size of the window, this may result in some elements that don't fit, as shown in Figure 3-3.

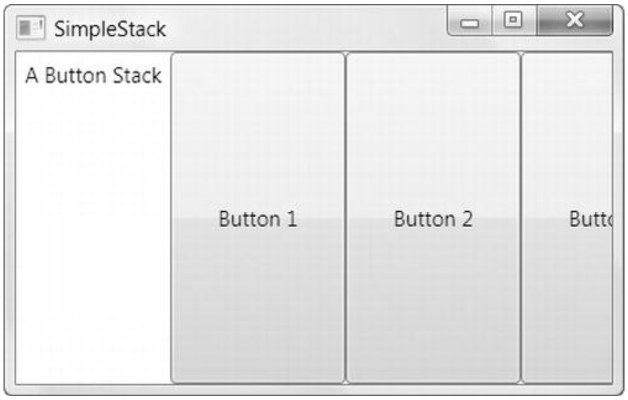


Figure 3-3. *The StackPanel with horizontal orientation*

Clearly, this doesn't provide the flexibility real applications need. Fortunately, you can fine-tune the way the StackPanel and other layout containers work by using layout properties, as described next.

Layout Properties

Although layout is determined by the container, the child elements can still get their say. In fact, layout panels work in concert with their children by respecting a small set of layout properties, listed in Table 3-3.

Table 3-3. *Layout Properties*

Name	Description
HorizontalAlignment	Determines how a child is positioned inside a layout container when there's extra horizontal space available. You can choose Center, Left, Right, or Stretch.
VerticalAlignment	Determines how a child is positioned inside a layout container when there's extra vertical space available. You can choose Center, Top, Bottom, or Stretch.
Margin	Adds a bit of breathing room around an element. The Margin property is an instance of the System.Windows.Thickness structure, with separate components for the top, bottom, left, and right edges.
MinWidth and MinHeight	Sets the minimum dimensions of an element. If an element is too large for its layout container, it will be cropped to fit.
MaxWidth and MaxHeight	Sets the maximum dimensions of an element. If the container has more room available, the element won't be enlarged beyond these bounds, even if the HorizontalAlignment and VerticalAlignment properties are set to Stretch.

Width and Height	Explicitly sets the size of an element. This setting overrides a Stretch value for the HorizontalAlignment or VerticalAlignment properties. However, this size won't be honored if it's outside of the bounds set by the MinWidth, MinHeight, MaxWidth, and MaxHeight.

All of these properties are inherited from the base FrameworkElement class and are therefore supported by all the graphical widgets you can use in a WPF window.

■ **Note** As you learned in Chapter 2, different layout containers can provide *attached properties* to their children. For example, all the children of a Grid object gain Row and Column properties that allow them to choose the cell where they're placed. Attached properties allow you to set information that's specific to a particular layout container. However, the layout properties in Table 3-3 are generic enough that they apply to many layout panels. Thus, these properties are defined as part of the base FrameworkElement class.

This list of properties is just as notable for what it *doesn't* contain. If you're looking for familiar position properties, such as Top, Right, and Location, you won't find them. That's because most layout containers (all except the Canvas) use automatic layout and don't give you the ability to explicitly position elements.

Alignment

To understand how these properties work, take another look at the simple StackPanel shown in Figure 3-2. In this example—a StackPanel with vertical orientation—the VerticalAlignment property has no effect because each element is given as much height as it needs and no more. However, the HorizontalAlignment *is* important. It determines where each element is placed in its row.

Ordinarily, the default HorizontalAlignment is Left for a label and Stretch for a Button. That's why every button takes the full column width. However, you can change these details:

```
<StackPanel>
  <Label HorizontalAlignment="Center">A Button Stack</Label>
  <Button HorizontalAlignment="Left">Button 1</Button>
  <Button HorizontalAlignment="Right">Button 2</Button>
  <Button>Button 3</Button>
  <Button>Button 4</Button>
</StackPanel>
```

Figure 3-4 shows the result. The first two buttons are given their minimum sizes and aligned accordingly, while the bottom two buttons are stretched over the entire StackPanel. If you resize the window, you'll see that the label remains in the middle and the first two buttons stay stuck to either side.

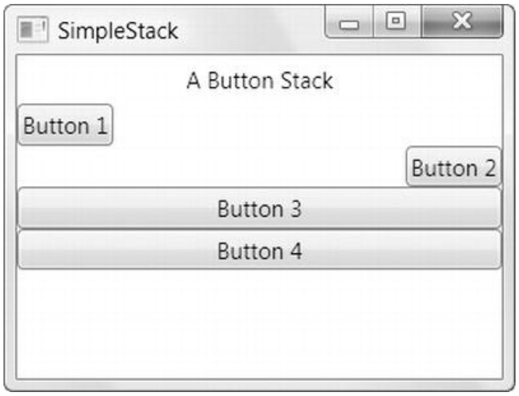


Figure 3-4. *A StackPanel with aligned buttons*

■ **Note** The StackPanel also has its own HorizontalAlignment and VerticalAlignment properties. By default, both of these are set to Stretch, and so the StackPanel fills its container completely. In this example, that means the StackPanel fills the window. If you use different settings, the StackPanel will be made just large enough to fit the widest control.

Margin

There's an obvious problem with the StackPanel example in its current form. A well-designed window doesn't just contain elements—it also includes a bit of extra space between the elements. To introduce this extra space and make the StackPanel example less cramped, you can set control margins.

When setting margins, you can set a single width for all sides, like this:

```
<Button Margin="5">Button 3</Button>
```

Alternatively, you can set different margins for each side of a control in the order *left, top, right, bottom*:

```
<Button Margin="5,10,5,10">Button 3</Button>
```

In code, margins are set by using the Thickness structure:

```
cmd.Margin = new Thickness(5);
```

Getting the right control margins is a bit of an art because you need to consider how the margin settings of adjacent controls influence one another. For example, if you have two buttons stacked on top of each other, and the topmost button has a bottom margin of 5, and the bottommost button has a top margin of 5, you have a total of 10 units of space between the two buttons.

Ideally, you'll be able to keep different margin settings as consistent as possible and avoid setting distinct values for the different margin sides. For instance, in the StackPanel example, it makes sense to use the same margins on the buttons and on the panel itself, as shown here:

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
   A Button Stack</Label>
  <Button Margin="3" HorizontalAlignment="Left">Button 1</Button>
  <Button Margin="3" HorizontalAlignment="Right">Button 2</Button>
  <Button Margin="3">Button 3</Button>
  <Button Margin="3">Button 4</Button>
</StackPanel>
```

This way, the total space between two buttons (the sum of the two button margins) is the same as the total space between the button at the edge of the window (the sum of the button margin and the StackPanel margin). Figure 3-5 shows this more respectable window, and Figure 3-6 shows how the margin settings break down.

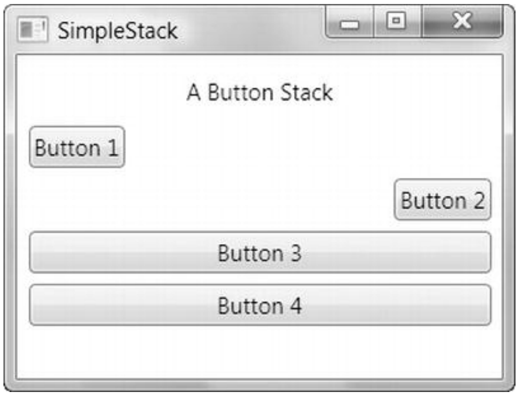


Figure 3-5. *Adding margins between elements*

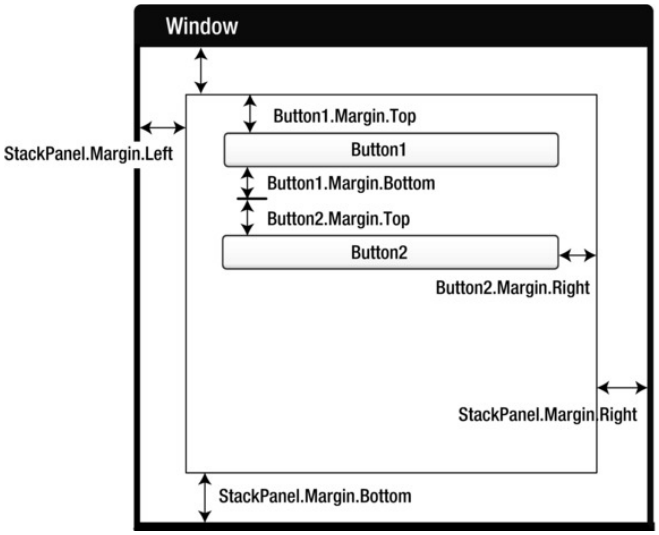


Figure 3-6. *How margins are combined*

Minimum, Maximum, and Explicit Sizes

Finally, every element includes Height and Width properties that allow you to give it an explicit size. However, it's rarely a good idea to take this step. Instead, use the maximum and minimum size properties to lock your control into the right range, if necessary.

■ **Tip** Think twice before setting an explicit size in WPF. In a well-designed layout, it shouldn't be necessary. If you do add size information, you risk creating a more brittle layout that can't adapt to changes (such as different languages and window sizes) and truncates your content.

For example, you might decide that the buttons in your StackPanel should stretch to fit the StackPanel but be made no larger than 200 units wide and no smaller than 100 units wide. (By default, buttons start with a minimum width of 75 units.) Here's the markup you need:

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
   A Button Stack</Label>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 1</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 2</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 3</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 4</Button>
</StackPanel>
```

■ **Tip** At this point, you might be wondering if there's an easier way to set properties that are standardized across several elements, such as the button margins in this example. The answer is *styles*—a feature that allows you to reuse property settings and even apply them automatically. You'll learn about styles in Chapter 11.

When the StackPanel sizes a button, it considers several pieces of information:

The minimum size: Each button will always be at least as large as the minimum size.

The maximum size: Each button will always be smaller than the maximum size (unless you've incorrectly set the maximum size to be smaller than the minimum size).

The content: If the content inside the button requires a greater width, the StackPanel will attempt to enlarge the button. (You can find out the size that the button wants by examining the DesiredSize property, which returns the minimum width or the content width, whichever is greater.)

The size of the container: If the minimum width is larger than the width of the StackPanel, a portion of the button will be cut off. Otherwise, the button will not be allowed to grow wider than the StackPanel, even if it can't fit all its text on the button surface.

The horizontal alignment: Because the button uses a HorizontalAlignment of Stretch (the default), the StackPanel will attempt to enlarge the button to fill the full width of the StackPanel.

The trick to understanding this process is to realize that the minimum and maximum size set the absolute bounds. Within those bounds, the StackPanel tries to respect the button's desired size (to fit its content) and its alignment settings.

Figure 3-7 sheds some light on how this works with the StackPanel. On the left is the window at its minimum size. The buttons are 100 units each, and the window cannot be resized to be narrower. If you shrink the window from this point, the right side of each button will be clipped off. (You can prevent this possibility by applying the MinWidth property to the window itself, so the window can't go below a minimum width.)

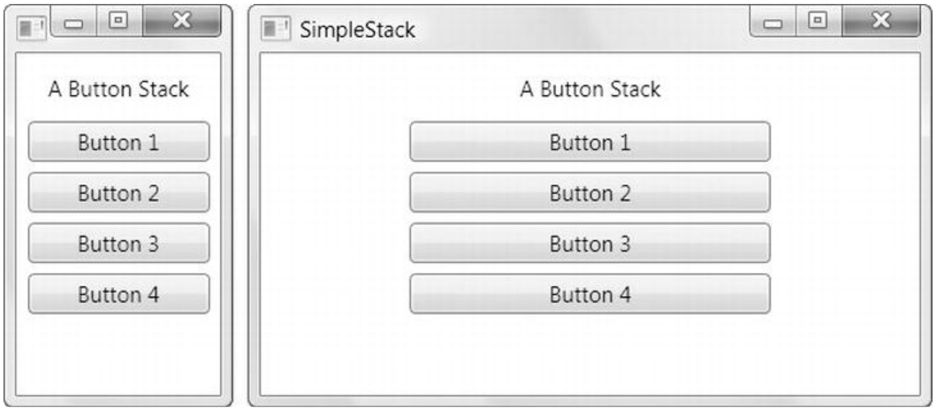


Figure 3-7. *Constrained button sizing*

As you enlarge the window, the buttons grow with it until they reach their maximum of 200 units. From this point on, if you make the window any larger, the extra space is added to either side of the button (as shown on the right).

■ **Note** In some situations, you might want to use code that checks how large an element is in a window. The Height and Width properties are no help because they indicate your desired size settings, which might not correspond to the actual rendered size. In an ideal scenario, you'll let your elements size to fit their content, and the Height and Width properties won't be set at all. However, you can find out the actual size used to render an element by reading the ActualHeight and ActualWidth properties. But remember, these values may change when the window is resized or the content inside it changes.

AUTOMATICALLY SIZED WINDOWS

In this example, there's still one element that has hard-coded sizes: the top-level window that contains the StackPanel (and everything else inside). For a number of reasons it still makes sense to hard-code window sizes:

However, automatically sized windows are possible, and they do make sense if you are constructing a simple window with dynamic content. To enable automatic window sizing, remove the Height and Width properties and set the Window.SizeToContent property to WidthAndHeight. The window will make itself just large enough to accommodate all its content. You can also allow a window to resize itself in just one dimension by using a SizeToContent value of Width or Height.

The Border

The Border isn't one of the layout panels, but it's a handy element that you'll often use alongside them. For that reason, it makes sense to introduce it now, before you go any further.

The Border class is pure simplicity. It takes a single piece of nested content (which is often a layout panel) and adds a background or border around it. To master the Border, you need nothing more than the properties listed in Table 3-4.

Table 3-4. *Properties of the Border Class*

Name	Description
Background	Sets a background that appears behind all the content in the border by using a Brush object. You can use a solid color or something more exotic.
BorderBrush and BorderThickness	Sets the color of the border that appears at the edge of the Border object, using a Brush object, and sets the width of the border, respectively. To show a border, you must set both properties.
CornerRadius	Allows you to gracefully round the corners of your border. The greater the CornerRadius, the more dramatic the rounding effect is.
Padding	Adds spacing between the border and the content inside. (By contrast, Margin adds spacing outside the border.)

Here's a straightforward slightly rounded border around a group of buttons in a StackPanel:

```
<Border Margin="5" Padding="5" Background="LightYellow"
 BorderBrush="SteelBlue" BorderThickness="3,5,3,5" CornerRadius="3"
 VerticalAlignment="Top">
  <StackPanel>
    <Button Margin="3">One</Button>
    <Button Margin="3">Two</Button>
    <Button Margin="3">Three</Button>
  </StackPanel>
</Border>
```

Figure 3-8 shows the result.

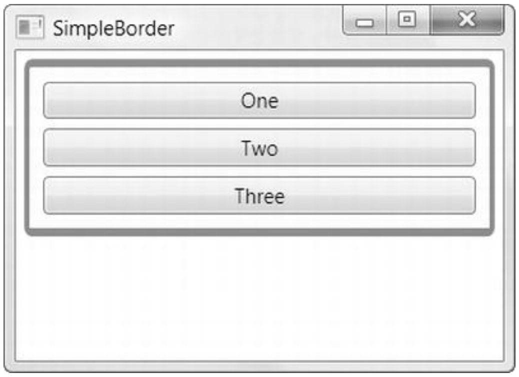


Figure 3-8. *A basic border*

Chapter 6 has more details about brushes and the colors you can use to set BorderBrush and Background.

■ **Note** Technically, the Border is a *decorator*, which is a type of element that's typically used to add some sort of graphical embellishment around an object. All decorators derive from the System.Windows.Controls.Decorator class. Most decorators are designed for use with specific controls. For example, the Button uses a ButtonChrome decorator to get its trademark rounded corner and shaded background, while the ListBox uses the ListBoxChrome decorator. There are also two more general decorators that are useful when composing user interfaces: the Border discussed here and the Viewbox you'll explore in Chapter 12.

The WrapPanel and DockPanel

Obviously, the StackPanel alone can't help you create a realistic user interface. To complete the picture, the StackPanel needs to work with other, more-capable layout containers. Only then can you assemble a complete window.

The most sophisticated layout container is the Grid, which you'll consider later in this chapter. But first, it's worth looking at the WrapPanel and DockPanel, which are two more of the simple layout containers provided by WPF. They complement the StackPanel by offering different layout behavior.

The WrapPanel

The WrapPanel lays out controls in the available space, one line or column at a time. By default, the WrapPanel.Orientation property is set to Horizontal; controls are arranged from left to right and then on subsequent rows. However, you can use Vertical to place elements in multiple columns.

■ **Tip** Like the StackPanel, the WrapPanel is really intended for control over small-scale details in a user interface, not complete window layouts. For example, you might use a WrapPanel to keep together the buttons in a toolbar-like control.

Here's an example that defines a series of buttons with different alignments and places them into the WrapPanel:

```
<WrapPanel Margin="3">
  <Button VerticalAlignment="Top">Top Button</Button>
  <Button MinHeight="60">Tall Button 2</Button>
  <Button VerticalAlignment="Bottom">Bottom Button</Button>
  <Button>Stretch Button</Button>
  <Button VerticalAlignment="Center">Centered Button</Button>
</WrapPanel>
```

Figure 3-9 shows how the buttons are wrapped to fit the current size of the WrapPanel (which is determined by the size of the window that contains it). As this example demonstrates, a WrapPanel in horizontal mode creates a series of imaginary rows, each of which is given the height of the tallest contained element. Other controls may be stretched to fit or aligned according to the VerticalAlignment property. In the example on the left in Figure 3-9, all the buttons fit into one tall row and are stretched or aligned to fit. In the example on the right, several buttons have been bumped to the second row. Because the second row does not include an unusually tall button, the row height is kept at the minimum button height. As a result, it doesn't matter what VerticalAlignment setting the various buttons in this row use.

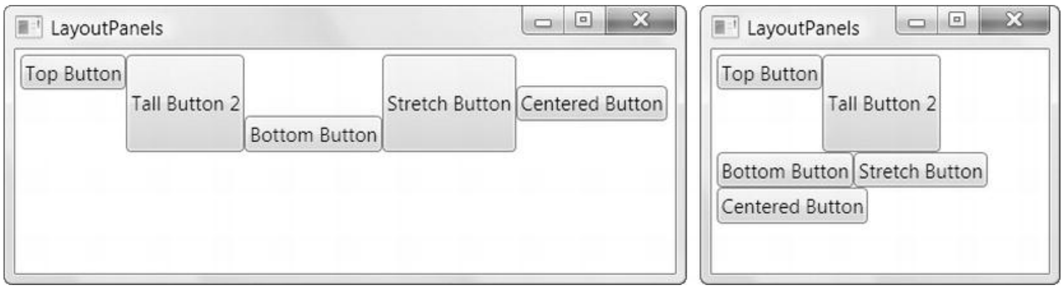


Figure 3-9. *Wrapped buttons*

■ **Note** The WrapPanel is the only panel that can't be duplicated with a crafty use of the Grid.

The DockPanel

The DockPanel is a more interesting layout option. It stretches controls against one of its outside edges. The easiest way to visualize this is to think of the toolbars that sit at the top of many Windows applications. These toolbars are docked to the top of the window. As with the StackPanel, docked elements get to choose one aspect of their layout. For example, if you dock a button to the top of a DockPanel, it's stretched across the entire width of the DockPanel but given whatever height it requires (based on the content and the MinHeight property). On the other hand, if you dock a button to the left side of a container, its height is stretched to fit the container, but its width is free to grow as needed.

The obvious question is, How do child elements choose the side where they want to dock? The answer is through an attached property named Dock, which can be set to Left, Right, Top, or Bottom. Every element that's placed inside a DockPanel automatically acquires this property.

Here's an example that puts one button on every side of a DockPanel:

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top">Top Button</Button>
  <Button DockPanel.Dock="Bottom">Bottom Button</Button>
  <Button DockPanel.Dock="Left">Left Button</Button>
  <Button DockPanel.Dock="Right">Right Button</Button>
  <Button>Remaining Space</Button>
</DockPanel>
```

This example also sets the LastChildFill to true, which tells the DockPanel to give the remaining space to the last element. Figure 3-10 shows the result.

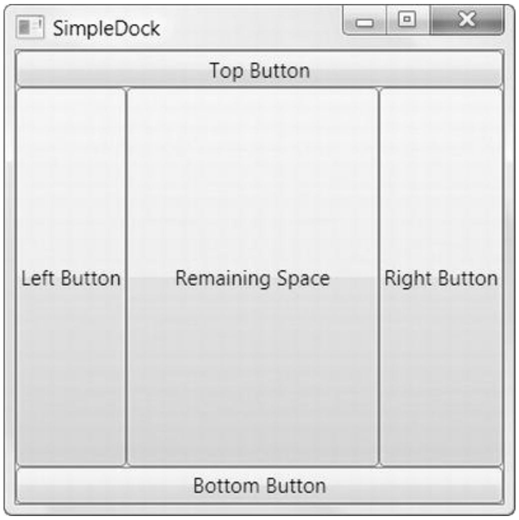


Figure 3-10. *Docking to every side*

Clearly, when docking controls, the order is important. In this example, the top and bottom buttons get the full edge of the DockPanel because they're docked first. When the left and right buttons are docked next, they fit between these two buttons. If you reversed this order, the left and right buttons would get the full sides, and the top and bottom buttons would become narrower because they'd be docked between the two side buttons.

You can dock several elements against the same side. In this case, the elements simply stack up against the side in the order they're declared in your markup. And if you don't like the spacing or the stretch behavior, you can tweak the Margin, HorizontalAlignment, and VerticalAlignment properties, just as you did with the StackPanel. Here's a modified version of the previous example that adjusts the alignment:

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top">A Stretched Top Button</Button>
  <Button DockPanel.Dock="Top" HorizontalAlignment="Center">
   A Centered Top Button</Button>
  <Button DockPanel.Dock="Top" HorizontalAlignment="Left">
   A Left-Aligned Top Button</Button>
  <Button DockPanel.Dock="Bottom">Bottom Button</Button>
  <Button DockPanel.Dock="Left">Left Button</Button>
  <Button DockPanel.Dock="Right">Right Button</Button>
  <Button>Remaining Space</Button>
</DockPanel>
```

The docking behavior is still the same. First the top buttons are docked, and then the bottom button is docked, and finally the remaining space is divided between the side buttons and a final button in the middle. Figure 3-11 shows the resulting window.

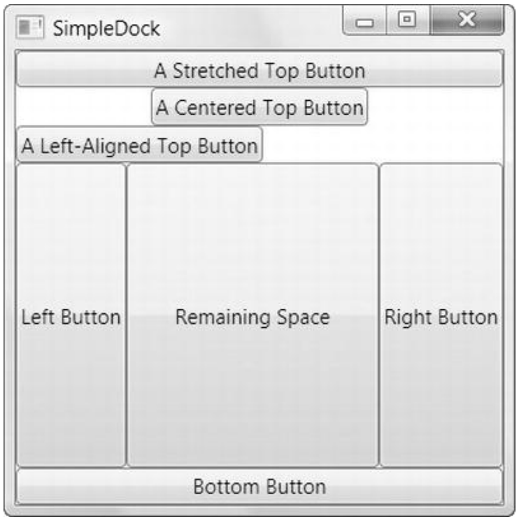


Figure 3-11. *Docking multiple elements to the top*

Nesting Layout Containers

The StackPanel, WrapPanel, and DockPanel are rarely used on their own. Instead, they're used to shape portions of your interface. For example, you could use a DockPanel to place different StackPanel and WrapPanel containers in the appropriate regions of a window.

For example, imagine you want to create a standard dialog box with an OK and Cancel button in the bottom-right corner and a large content region in the rest of the window. You can model this interface with WPF in several ways, but the easiest option that uses the panels you've seen so far is as follows:

1. Create a horizontal StackPanel to wrap the OK and Cancel buttons together.
2. Place the StackPanel in a DockPanel and use that to dock it to the bottom of the window.

3. Set DockPanel.LastChildFill to true so you can use the rest of the window to fill in other content. You can add another layout control here or just an ordinary TextBox control (as in this example).

4. Set the margin properties to give the right amount of whitespace.

Here's the final markup:

```
<DockPanel LastChildFill="True">
  <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right"
   Orientation="Horizontal">
    <Button Margin="10,10,2,10" Padding="3">OK</Button>
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>
  </StackPanel>
  <TextBox DockPanel.Dock="Top" Margin="10">This is a test.</TextBox>
</DockPanel>
```

In this example, the Padding property adds some minimum space between the button border and the content inside (the word *OK* or *Cancel*). Figure 3-12 shows the rather pedestrian dialog box this creates.

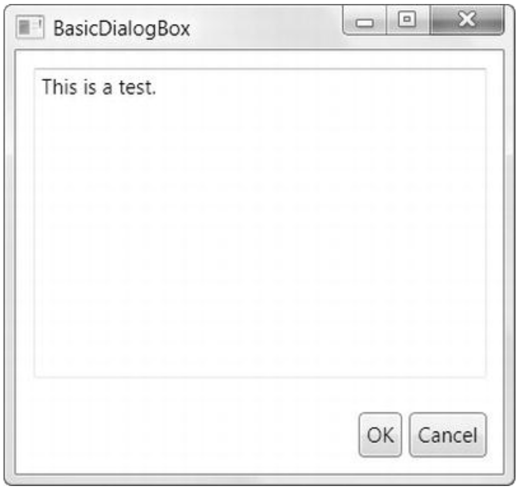


Figure 3-12. *A basic dialog box*

At first glance, this seems like a fair bit more work than placing controls in precise positions by using coordinates. And in many cases, it is. However, the longer setup time is compensated by the ease with which you can change the user interface in the future. For example, if you decide you want the OK and Cancel buttons to be centered at the bottom of the window, you simply need to change the alignment of the StackPanel that contains them:

```
<StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Center" ... >
```

Compared to older user interface frameworks such as Windows Forms, the markup used here is cleaner, simpler, and more compact. If you add a dash of styles to this window (Chapter 11), you can improve it even further and remove other extraneous details (such as the margin settings) to create a truly adaptable user interface.

■ **Tip** If you have a densely nested tree of elements, it's easy to lose sight of the overall structure. Visual Studio provides a handy feature that shows you a tree representation of your elements and allows you to click your way down to the element you want to look at (or modify). This feature is the Document Outline window, and you can show it by choosing View ➤ Other Windows ➤ Document Outline from the menu.

The Grid

The Grid is the most powerful layout container in WPF. Much of what you can accomplish with the other layout controls is also possible with the Grid. The Grid is also an ideal tool for carving your window into smaller regions that you can manage with other panels. In fact, the Grid is so useful that when you add a new XAML document for a window in Visual Studio, it automatically adds the Grid tags as the first-level container, nested inside the root Window element.

The Grid separates elements into an invisible grid of rows and columns Although more than one element can be placed in a single cell (in which case they overlap), it generally makes sense to place just a single element per cell. Of course, that element may itself be another layout container that organizes its own group of contained controls.

■ **Tip** Although the Grid is designed to be invisible, you can set the Grid.ShowGridLines property to true to take a closer look. This feature isn't really intended for prettying up a window. Instead, it's a debugging convenience that's designed to help you understand how the Grid has subdivided itself into smaller regions. This feature is important because you have the ability to control exactly how the Grid chooses column widths and row heights.

Creating a Grid-based layout is a two-step process. First, you choose the number of columns and rows that you want. Next, you assign the appropriate row and column to each contained element, thereby placing it in just the right spot.

You create grids and rows by filling the Grid.ColumnDefinitions and Grid.RowDefinitions collections with objects. For example, if you decide you need two rows and three columns, you'd add the following tags:

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  ...
</Grid>
```

As this example shows, it's not necessary to supply any information in a RowDefinition or ColumnDefinition element. If you leave them empty (as shown here), the Grid will share the space evenly between all rows and columns. In this example, each cell will be exactly the same size, depending on the size of the containing window.

To place individual elements into a cell, you use the attached Row and Column properties. Both these properties take 0-based index numbers. For example, here's how you could create a partially filled grid of buttons:

```
<Grid ShowGridLines="True">
  ...

  <Button Grid.Row="0" Grid.Column="0">Top Left</Button>
  <Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
  <Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
  <Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

Each element must be placed into its cell explicitly. This allows you to place more than one element into a cell (which rarely makes sense) or leave certain cells blank (which is often useful). It also means you can declare your elements out of order, as with the final two buttons in this example. However, it makes for clearer markup if you define your controls row by row, and from right to left in each row.

There is one exception. If you don't specify the Grid.Row property, the Grid assumes that it's 0. The same behavior applies to the Grid.Column property. Thus, you leave both attributes off of an element to place it in the first cell of the Grid.

■ **Note** The Grid fits elements into predefined rows and columns. This is different from layout containers such as the WrapPanel and StackPanel that create implicit rows or columns as they lay out their children. If you want to create a grid that has more than one row and one column, you must define your rows and columns explicitly by using RowDefinition and ColumnDefinition objects.

Figure 3-13 shows how this simple grid appears at two sizes. Notice that the ShowGridLines property is set to true so that you can see the separation between each column and row.

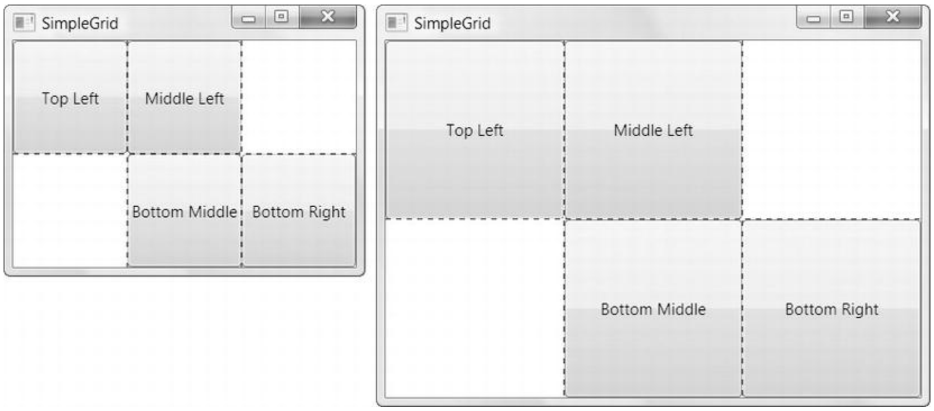


Figure 3-13. *A simple grid*

As you would expect, the Grid honors the basic set of layout properties listed in Table 3-3. That means you can add margins around the content in a cell, you can change the sizing mode so an element doesn't grow to fill the entire cell, and you can align an item along one of the edges of a cell. If you force an element to have a size that's larger than the cell can accommodate, part of the content will be chopped off.

USING THE GRID IN VISUAL STUDIO

When you use a Grid on the Visual Studio design surface, you'll find that it works a bit differently than other layout containers. As you drag an element into a Grid, Visual Studio allows you to place it in a precise position. Visual Studio works this magic by setting the Margin property of your element.

When setting margins, Visual Studio uses the closest corner. For example, if your element is nearest to the top-left corner of the Grid, Visual Studio pads the top and left margins to position the element (and leaves the right and bottom margins at 0). If you drag your element down closer to the bottom-left corner, Visual Studio sets the bottom and left margins instead and sets the VerticalAlignment property to Bottom. This obviously affects how the element will move when the Grid is resized.

Visual Studio's margin-setting process seems straightforward enough, but most of the time it won't create the results you want. Usually, you'll want a more flexible flow layout that allows some elements to expand dynamically and push others out of the way. In this scenario, you'll find that hard-coding a position with the Margin property is extremely inflexible. The problems get worse when you add multiple elements, because Visual Studio won't automatically add new cells. As a result, all the elements will be placed in the same cell. Different elements may be aligned to different corners of the Grid, which will cause them to move with respect to one another (and even overlap each other) as the window is resized.

Once you understand how the Grid works, you can correct these problems. The first trick is to configure your Grid before you begin adding elements by defining its rows and columns. (You can edit the RowDefinitions and ColumnDefinitions collections by using the Properties window.) After you've set up the Grid, you can drag and drop the elements you want into the Grid and configure their margin and alignment settings in the Properties window or by editing the XAML by hand.

Fine-Tuning Rows and Columns

If the Grid were simply a proportionately sized collection of rows and columns, it wouldn't be much help. Fortunately, it's not. To unlock the full potential of the Grid, you can change the way each row and column is sized.

The Grid supports three sizing strategies:

Absolute sizes: You choose the exact size by using device-independent units. This is the least useful strategy because it's not flexible enough to deal with changing content size, changing container size, or localization.

Automatic sizes: Each row or column is given exactly the amount of space it needs, and no more. This is one of the most useful sizing modes.

Proportional sizes: Space is divided between a group of rows or columns. This is the standard setting for all rows and columns. For example, in Figure 3-13 you'll see that all cells increase in size proportionately as the Grid expands.

For maximum flexibility, you can mix and match these sizing modes. For example, it's often useful to create several automatically sized rows and then let one or two remaining rows get the leftover space through proportional sizing.

You set the sizing mode by using the Width property of the ColumnDefinition object or the Height property of the RowDefinition object to a number. For example, here's how you set an absolute width of 100 device-independent units:

```
<ColumnDefinition Width="100"></ColumnDefinition>
```

To use automatic sizing, you use a value of Auto:

```
<ColumnDefinition Width="Auto"></ColumnDefinition>
```

Finally, to use proportional sizing, you use an asterisk (*):

```
<ColumnDefinition Width="*"></ColumnDefinition>
```

If you use a mix of proportional sizing and other sizing modes, the proportionally sized rows or columns get whatever space is left over.

If you want to divide the remaining space unequally, you can assign a *weight* which you must place before the asterisk. For example, if you have two proportionately sized rows and you want the first to be half as high as the second, you could share the remaining space like this:

```
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="2*"></RowDefinition>
```

This tells the Grid that the height of the second row should be twice the height of the first row. You can use whatever numbers you like to portion out the extra space.

■ **Note** It's easy to interact with ColumnDefinition and RowDefinition objects programmatically. You simply need to know that the Width and Height properties are GridLength objects. To create a GridLength that represents a specific size, just pass the appropriate value to the GridLength constructor. To create a GridLength that represents a proportional (*) size, pass the number to the GridLength constructor, and pass GridUnitType.Star as the second constructor argument. To indicate automatic sizing, use the static property GridLength.Auto.

Using these size modes, you can duplicate the simple dialog box example shown in Figure 3-12 by using a top-level Grid container to split the window into two rows, rather than a DockPanel. Here's the markup you'd need:

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <TextBox Margin="10" Grid.Row="0">This is a test.</TextBox>
  <StackPanel Grid.Row="1" HorizontalAlignment="Right" Orientation="Horizontal">
    <Button Margin="10,10,2,10" Padding="3">OK</Button>
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>
  </StackPanel>
</Grid>
```

■ **Tip** This Grid doesn't declare any columns. This is a shortcut you can take if your Grid uses just one column and that column is proportionately sized (so it fills the entire width of the Grid).

This markup is slightly longer, but it has the advantage of declaring the controls in the order they appear, which makes it easier to understand. In this case, the approach you take is simply a matter of preference. And if you want, you could replace the nested StackPanel with a one-row, two-column Grid.

■ **Note** You can create almost any interface by using nested Grid containers. (One exception is wrapped rows or columns that use the WrapPanel.) However, when you're dealing with small sections of user interface or laying out a small number of elements, it's often simpler to use the more specialized StackPanel and DockPanel containers.

Layout Rounding

As you learned in Chapter 1, WPF uses a resolution-independent system of measurement. Although this gives it the flexibility to work on a variety of hardware, it also sometimes introduces a few quirks. One of these is that elements can be aligned on subpixel boundaries—in other words, positioned with fractional coordinates that don't exactly line up with physical pixels. You can force this to happen by giving adjacent layout containers nonintegral sizes. But this quirk also crops up in some situations when you might not expect it, such as when creating a proportionately sized Grid.

For example, imagine that a two-column Grid has 200 pixels to work with. If you've split it evenly into two proportional columns, that means each gets 100 pixels. But if you have 175 pixels, the division isn't as clean, and each column gets 87.5 pixels. That means the second column is slightly displaced from the ordinary pixel boundaries. Ordinarily, this isn't a problem, but if that column contains one of the shape elements, a border, or an image, that content may appear blurry because WPF uses anti-aliasing to "blend" what would otherwise be sharp edges over pixel boundaries. Figure 3-14 shows the problem in action. It magnifies a portion of a window that contains two Grid containers. The topmost Grid does not use layout rounding, and as a result, the sharp edge of the rectangle inside becomes blurry at certain window sizes.

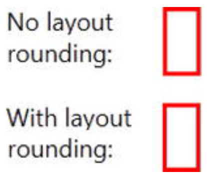


Figure 3-14. *Blur from proportionate sizing*

If this problem affects your layout, there's an easy fix. Just set the UseLayoutRounding property to true on your layout container:

```
<Grid UseLayoutRounding="True">
```

Now WPF will ensure that all the content in that layout container is snapped to the nearest pixel boundary, removing any blurriness.

Spanning Rows and Columns

You've already seen how to place elements in cells by using the Row and Column attached properties. You can also use two more attached properties to make an element stretch over several cells: RowSpan and ColumnSpan. These properties take the number of rows or columns that the element should occupy.

For example, this button will take all the space that's available in the first and second cell of the first row:

```
<Button Grid.Row="0" Grid.Column="0" Grid.~~RowSpan=~~"2">Span Button</Button>
```

And this button will stretch over four cells in total by spanning two columns and two rows:

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2" Grid.ColumnSpan="2">
  Span Button</Button>
```

Row and column spanning can achieve some interesting effects and is particularly handy when you need to fit elements in a tabular structure that's broken up by dividers or longer sections of content.

Using column spanning, you could rewrite the simple dialog box example from Figure 3-12 by using just a single Grid. This Grid divides the window into three columns, spreads the text box over all three, and uses the last two columns to align the OK and Cancel buttons.

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <TextBox Margin="10" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
    This is a test.</TextBox>
  <Button Margin="10,10,2,10" Padding="3"
    Grid.Row="1" Grid.Column="1">OK</Button>
  <Button Margin="2,10,10,10" Padding="3"
    Grid.Row="1" Grid.Column="2">Cancel</Button>
</Grid>
```

Most developers will agree that this layout isn't clear or sensible. The column widths are determined by the size of the two buttons at the bottom of the window, which makes it difficult to add new content into the existing Grid structure. If you make even a minor addition to this window, you'll probably be forced to create a new set of columns.

As this shows, when you choose the layout containers for a window, you aren't simply interested in getting the correct layout behavior—you also want to build a layout structure that's easy to maintain and enhance in the future. A good rule of thumb is to use smaller layout containers such as the StackPanel for one-off layout tasks, such as arranging a group of buttons. On the other hand, if you need to apply a consistent structure to more than one area of your window (as with the text box column shown later in Figure 3-22), the Grid is an indispensable tool for standardizing your layout.

Splitting Windows

Every Windows user has seen *splitter bars*—draggable dividers that separate one section of a window from another. For example, when you use Windows Explorer, you're presented with a list of folders (on the left) and a list of files (on the right). You can drag the splitter bar in between to determine what proportion of the window is given to each pane.

In WPF, splitter bars are represented by the GridSplitter class and are a feature of the Grid. By adding a GridSplitter to a Grid, you give the user the ability to resize rows or columns. Figure 3-15 shows a window that has a GridSplitter between two columns. By dragging the splitter bar, the user can change the relative widths of both columns.

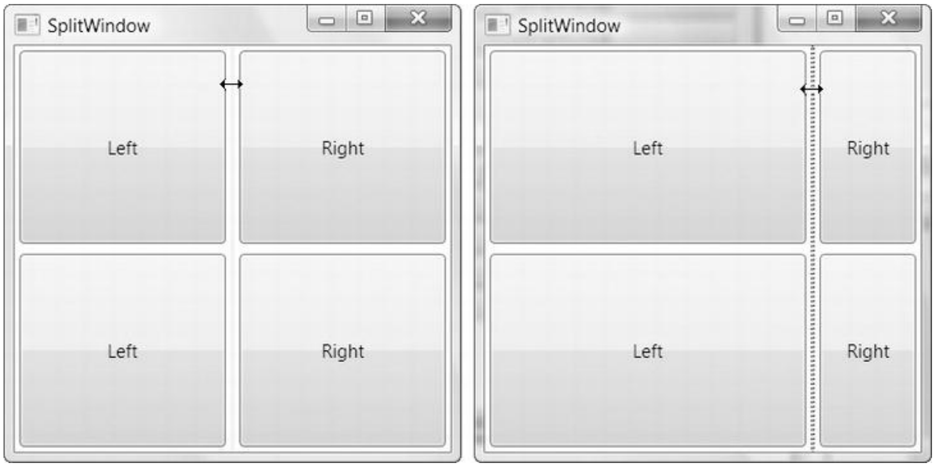


Figure 3-15. *Moving a splitter bar*

Most programmers find that the GridSplitter isn't the most intuitive part of WPF. Understanding how to use it to get the effect you want takes a little experimentation. Here are a few guidelines:

- The GridSplitter must be placed in a Grid cell. You can place the GridSplitter in a cell with existing content, in which case you need to adjust the margin settings so it doesn't overlap. A better approach is to reserve a dedicated column or row for the GridSplitter, with a Height or Width value of Auto.

- The GridSplitter always resizes entire rows or columns (not single cells). To make the appearance of the GridSplitter consistent with this behavior, you should stretch the GridSplitter across an entire row or column, rather than limit it to a single cell. To accomplish this, you use the RowSpan or ColumnSpan properties 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 by 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 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.

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 by 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.

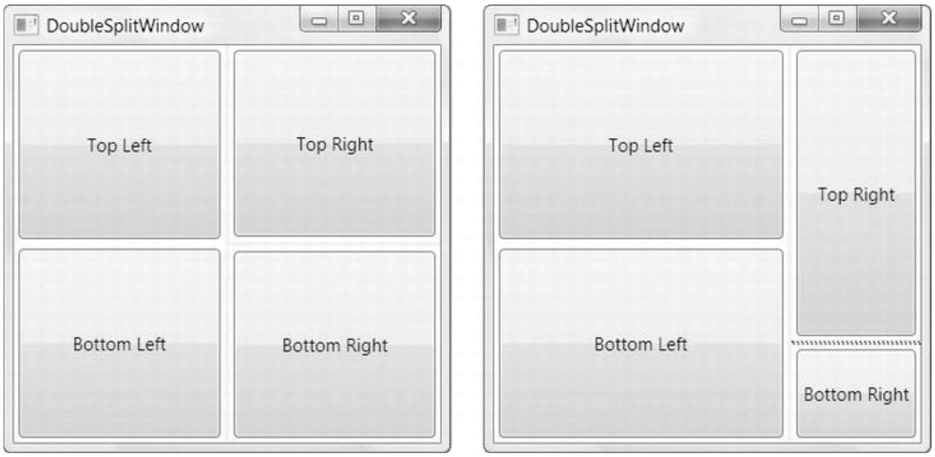


Figure 3-16. *Resizing a window with two splits*

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.

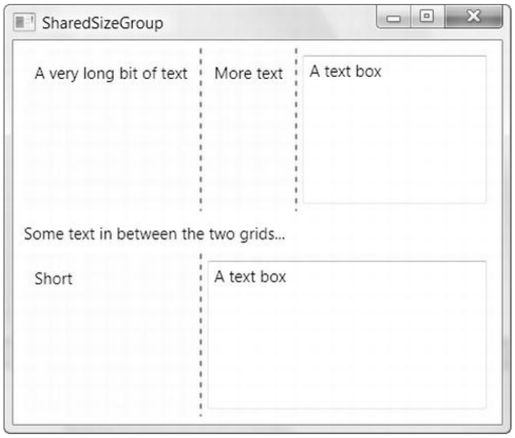


Figure 3-17. *Two grids that share a column definition*

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).

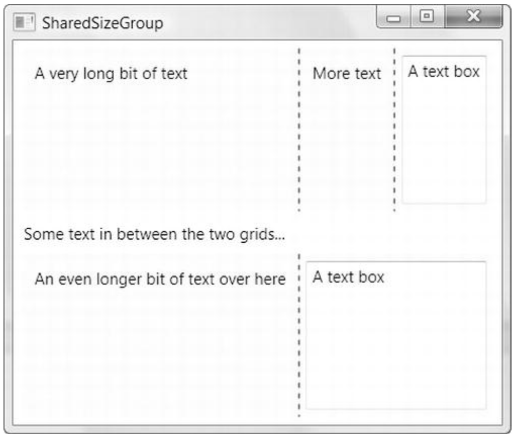

SharedSizeGroup
A very long bit of text
More text
A text box
Some text in between the two grids...
An even longer bit of text over here
A text box

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 propertyon 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 by using exact coordinates, which is a poor choice for designing rich data-driven forms and standard dialog boxes, but 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 by 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.

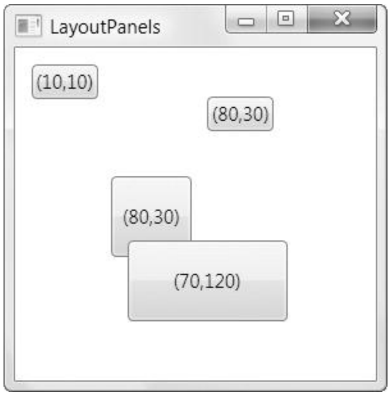


***Figure 3-19.** Explicitly positioned buttons in a Canvas*

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 the 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 on top 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 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 by 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. A *stylus* is a 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 by using the mouse.

The InkCanvas 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.

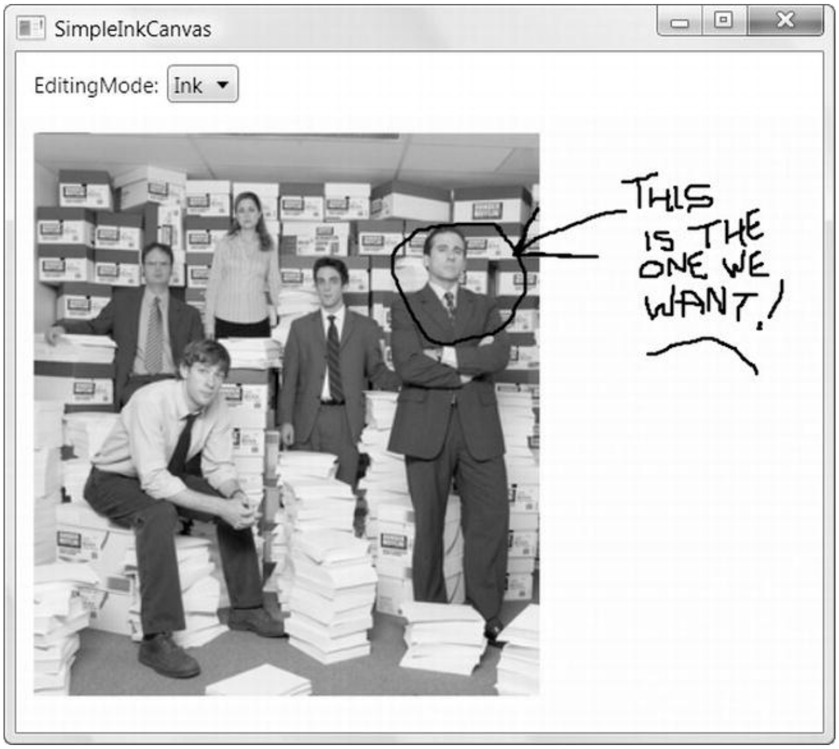


Figure 3-20. *Adding strokes in an InkCanvas*

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.

EraseByStroke	The InkCanvas erases a stroke when it's clicked. A user can switch to this mode by using the back end of a stylus. (You can determine the current mode by 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. After 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).

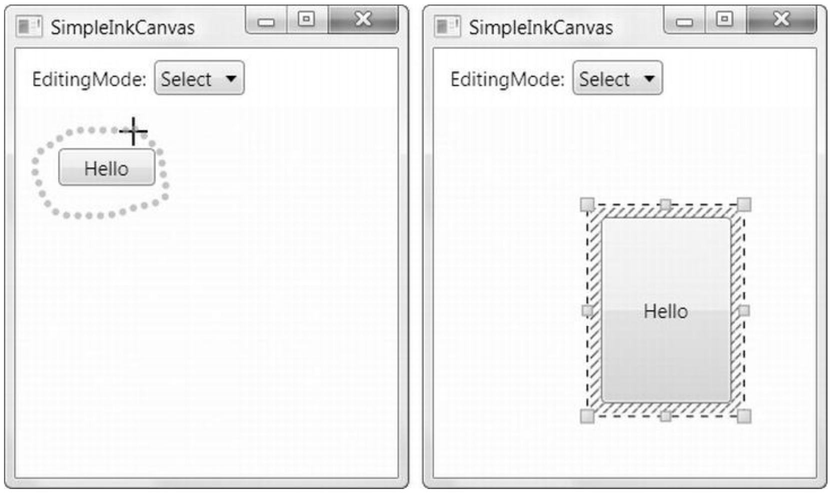


Figure 3-21. *Moving and resizing an element in the InkCanvas*

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.

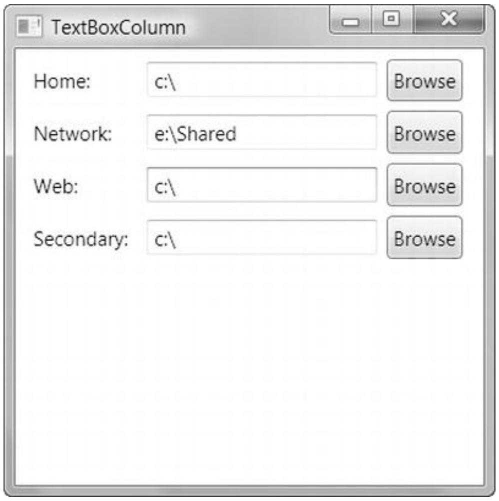


Figure 3-22. *Folder settings in a column*

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.

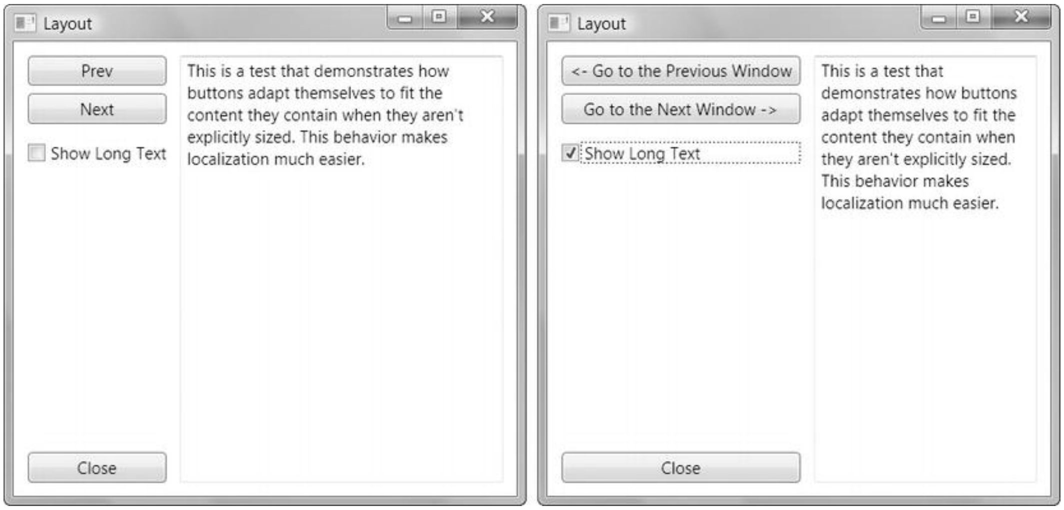
One fact that's not immediately obvious is how flexible this window is because of the use of the Grid control. None of the individual elements—the labels, text boxes, and buttons—have hard-coded positions or sizes. As a result, you can quickly make changes to the entire grid simply by tweaking the ColumnDefinition 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 between the rows—such as separator lines to divide 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.



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.

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:

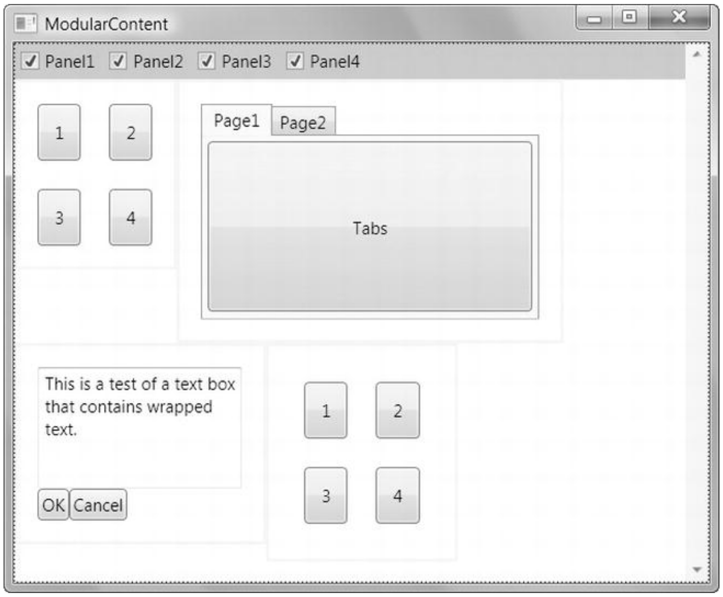
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <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, such as 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 by using the check boxes at the top of the window.



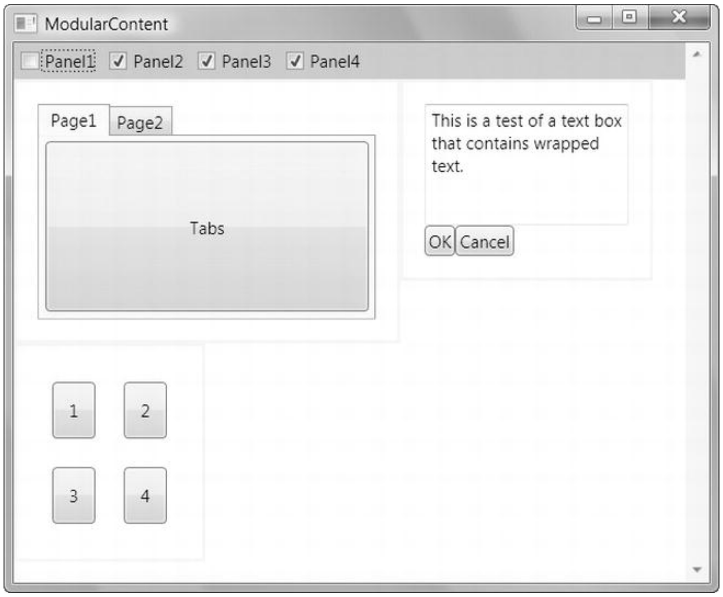
This is a test of a text box that contains wrapped text.

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.

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:



This is a test of a text box that contains wrapped text.

Figure 3-25. Hiding some panels

```
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 WPF layout model and learned how to place elements in stacks, grids, and other arrangements. You built more-complex layouts by 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 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.