```
    <!-- This the flip button. -->
    <ToggleButton Grid.Row="1" x:Name="FlipButton" Margin="0,10,0,0">
    </ToggleButton>

  </Grid>
</ControlTemplate>
```

When you create a default control template, it's best to avoid hard-coding details that the control consumer may want to customize. Instead, you need to use template binding expressions. In this example, you set several properties using template-binding expressions: BorderBrush, BorderThickness, CornerRadius, Background, FrontContent, and BackContent. To set the default value for these properties (and thereby ensure that you get the right visual even if the control consumer doesn't set them), you must add additional setters to your control's default style.

The Flip Button

The control template shown in the previous example includes a ToggleButton. However, it uses the ToggleButton's default appearance, which makes the ToggleButton look like an ordinary button, complete with the traditional shaded background. This isn't suitable for the FlipPanel.

Although you can place any content you want inside the ToggleButton, the FlipPanel requires a bit more. It needs to do away with the standard background and change the appearance of the elements inside depending on the state of the ToggleButton. As you saw earlier in Figure 18-5, the ToggleButton points the way the content will be flipped (right initially, when the front faces forward, and left when the back faces forward). This makes the purpose of the button clearer.

To create this effect, you need to design a custom control template for the ToggleButton. This control template can include the shape elements that draw the arrow you need. In this example, the ToggleButton is drawn using an Ellipse element for the circle and a Path element for the arrow, both of which are placed in a single-cell Grid:

```
<ToggleButton Grid.Row="1" x:Name="FlipButton" RenderTransformOrigin="0.5,0.5"
 Margin="0,10,0,0" Width="19" Height="19">
  <ToggleButton.Template>
    <ControlTemplate>
      <Grid>
        <Ellipse Stroke="#FFA9A9A9" Fill="AliceBlue"></Ellipse>
        <Path Data="M1,1.5L4.5,5 8,1.5" Stroke="#FF666666" StrokeThickness="2"
         HorizontalAlignment="Center" VerticalAlignment="Center"></Path>
      </Grid>
    </ControlTemplate>
  </ToggleButton.Template>
</ToggleButton>
```

The ToggleButton needs one more detail—a RotateTransform that turns the arrow away from one side to point at the other. This RotateTransform will be used when you create the state animations:

```
<ToggleButton.RenderTransform>
  <RotateTransform x:Name="FlipButtonTransform" Angle="-90"></RotateTransform>
</ToggleButton.RenderTransform>
```

Defining the State Animations

The state animations are the most interesting part of the control template. They're the ingredients that provide the flipping behavior. They're also the details that are most likely to be changed if a developer creates a custom template for the FlipPanel.

To define state groups, you must add the VisualStateManager.VisualStateGroups element in the root element of your control template, as shown here:

```
<ControlTemplate TargetType="{x:Type local:FlipPanel}">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      ...
    </VisualStateManager.VisualStateGroups>

    ...
  </Grid>
</ControlTemplate>
```

■ **Note** To add the VisualStateManager element to a template, your template must use a layout panel. This layout panel holds both the visuals for your control and the VisualStateManager, which is invisible. The VisualStateManager defines storyboards with the animations that the control can use at the appropriate time to alter its appearance.

Inside the VisualStateGroups element, you can create the state groups using appropriately named VisualStateGroup elements. Inside each VisualStateGroup, you add a VisualState element for each visual state. In the case of the FlipPanel, there is one group that contains two visual states:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ViewStates">
    <VisualState x:Name="Normal">
      ...
    </VisualState>
  </VisualStateGroup>

  <VisualStateGroup x:Name="FocusStates">
    <VisualState x:Name="Flipped">
      ...
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Each state corresponds to a storyboard with one or more animations. If these storyboards exist, they're triggered at the appropriate times. (If they don't, the control should degrade gracefully, without raising an error.)

In the default control template, the animations use a simple fade to change from one content region to the other and use a rotation to flip the ToggleButton arrow around to point in the other direction. Here's the markup that takes care of both tasks:

```
<VisualState x:Name="Normal">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="BackContent"
     Storyboard.TargetProperty="Opacity" To="0" Duration="0" ></DoubleAnimation>
  </Storyboard>
</VisualState>

<VisualState x:Name="Flipped">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
     Storyboard.TargetProperty="Angle" To="90" Duration="0"></DoubleAnimation>
    <DoubleAnimation Storyboard.TargetName="FrontContent"
     Storyboard.TargetProperty="Opacity" To="0" Duration="0"></DoubleAnimation>
  </Storyboard>
</VisualState>
```

You'll notice that the visual states set the animation duration to 0, which means the animation applies its effect instantaneously. This might seem a little odd—after all, don't you need a more gradual change to notice the animated effect?

In fact, this design is perfectly correct, because visual states are meant to indicate how the control looks while it's in the appropriate state. For example, a flipped panel simply shows its background content while in the flipped state. The flipping process is a *transition* that happens just before the FlipControl enters the flipped state, not part of the state itself. (This distinction between states and transitions is important, because some controls *do* have animations that run during a state. For example, think of the button example from Chapter 17 that featured the pulsing background color while the mouse hovers over it.)

Defining the State Transitions

A transition is an animation that starts from the current state and ends at the new state. One of the advantages of the transition model is that you don't need to create the storyboard for this animation. For example, if you add the markup shown here, WPF creates a 0.7-second animation to change the opacity of the FlipPanel, creating the pleasant fade effect you want:

```
<VisualStateGroup x:Name="ViewStates">
  <VisualStateGroup.Transitions>
    <VisualTransition GeneratedDuration="0:0:0.7"></VisualTransition>
  </VisualStateGroup.Transitions>

  <VisualState x:Name="Normal">
    ...
  </VisualState>

  <VisualState x:Name="Flipped">
    ...
  </VisualState>
</VisualStateGroup>
```

Transitions apply to state groups. When you define a transition, you must add it to the VisualStateGroup.Transitions collection. This example uses the simplest sort of transition: a *default transition*, which applies to all the state changes for that group.

A default transition is convenient, but it's a one-size-fits-all solution that's not always suitable. For example, you may want the FlipPanel to transition at different speeds depending on which state its entering. To set this up, you need to define multiple transitions, and you need to set the To property to specify when the transition will come into effect.

For example, if you have these transitions

```
<VisualStateGroup.Transitions>
  <VisualTransition To="Flipped" GeneratedDuration="0:0:0.5" />
  <VisualTransition To="Normal" GeneratedDuration="0:0:0.1" />
</VisualStateGroup.Transitions>
```

the FlipPanel will switch to the Flipped state in 0.5 seconds, and it will enter the Normal state in 0.1 seconds.

This example shows transitions that apply when entering specific states, but you can also use the From property to create a transition that applies when leaving a state, and you can use the To and From properties in conjunction to create even more specific transitions that apply only when moving between two specific states. When applying transitions, WPF looks through the collection of transitions to find the most specific one that applies, and it uses only that one.

For even more control, you can create custom transition animations that take the place of the automatically generated transitions WPF would normally use. You may create a custom transition for several reasons. Here are some examples: to control the pace of the animation with a more sophisticated animation, to use an animation easing, to run several animations in succession, or to play a sound at the same time as an animation.

To define a custom transition, you place a storyboard with one or more animations inside the VisualTransition element. In the FlipPanel example, you can use custom transitions to make sure the ToggleButton arrow rotates itself quickly, while the fade takes place more gradually.

```
<VisualStateGroup.Transitions>
  <VisualTransition GeneratedDuration="0:0:0.7" To="Flipped">
    <Storyboard>
      <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
       Storyboard.TargetProperty="Angle" To="90"
       Duration="0:0:0.2"></DoubleAnimation>
    </Storyboard>
  </VisualTransition>
  <VisualTransition GeneratedDuration="0:0:0.7" To="Normal">
    <Storyboard>
      <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
       Storyboard.TargetProperty="Angle" To="-90"
       Duration="0:0:0.2"></DoubleAnimation>
    </Storyboard>
  </VisualTransition>
</VisualStateGroup.Transitions>
```

■ **Note** When you use a custom transition, you must still set the VisualTransition.GeneratedDuration property to match the duration of your animation. Without this detail, the VisualStateManager can't use your transition, and it will apply the new state immediately. (The actual time value you use still has no effect on your custom transition, because it applies only to automatically generated animations.)

Unfortunately, many controls will require custom transitions, and writing them is tedious. You still need to keep the zero-length state animations, which also creates some unavoidable duplication of details between your visual states and your transitions.

Wiring Up the Elements

Now that you've polished off a respectable control template, you need to fill in the plumbing in the FlipPanel control to make it work.

The trick is the OnApplyTemplate(), which you also used to set bindings in the ColorPicker. The OnApplyTemplate() method for the FlipPanel retrieves the ToggleButton for the FlipButton and FlipButtonAlternate parts and attaches event handlers to each so it can react when the user clicks to flip the control. Finally, the OnApplyTemplate() method ends by calling a custom method named ChangeVisualState(), which ensures that the control's visuals match its current state:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    // Wire up the ToggleButton.Click event.
    ToggleButton flipButton = base.GetTemplateChild("FlipButton") as ToggleButton;
    if (flipButton != null) flipButton.Click += flipButton_Click;

    // Allow for two flip buttons if needed (one for each side of the panel).
    ToggleButton flipButtonAlternate =
      base.GetTemplateChild("FlipButtonAlternate") as ToggleButton;
    if (flipButtonAlternate != null) flipButtonAlternate.Click += flipButton_Click;

    // Make sure the visuals match the current state.
    this.ChangeVisualState(false);
}
```

■ **Tip** When calling GetTemplateChild(), you need to indicate the string name of the element you want. To avoid possible errors, you can declare this string as a constant in your control. You can then use that constant in the TemplatePart attribute and when calling GetTemplateChild().

Here's the very simple event handler that allows the user to click the ToggleButton and flip the panel:

```
private void flipButton_Click(object sender, RoutedEventArgs e)
{
    this.IsFlipped = !this.IsFlipped;
    ChangeVisualState(true);
}
```

Fortunately, you don't need to manually trigger the state animations. Nor do you need to create or trigger the transition animations. Instead, to change from one state to another, you call the static VisualStateManager.GoToState() method. When you do, you pass in a reference to the control object that's changing state, the name of the new state, and a Boolean value that determines whether a transition is shown. This value should be true when it's a user-initiated change (for example, when the user clicks the ToggleButton) but false when it's a property setting (for example, if the markup for your page sets the initial value of the IsFlipped property).

Dealing with all the different states a control supports can become messy. To avoid scattering GoToState() calls throughout your control code, most controls add a custom method like the ChangeVisualState() method in the FlipPanel. This method has the responsibility of applying the correct state in each state group. The code inside uses one if block (or switch statement) to apply the current state in each state group. This approach works because it's completely acceptable to call GoToState() with the name of the current state. In this situation, when the current state and the requested state are the same, nothing happens.

Here's the code for the FlipPanel's version of the ChangeVisualState() method:

```
private void ChangeVisualState(bool useTransitions)
{
    if (!IsFlipped)
    {
        VisualStateManager.GoToState(this, "Normal", useTransitions);
    }
    else
    {
        VisualStateManager.GoToState(this, "Flipped", useTransitions);
    }
}
```

Usually, you call the ChangeVisualState() method (or your equivalent) in the following places:

- After initializing the control at the end of the OnApplyTemplate() method.
- When reacting to an event that represents a state change, such as a mouse movement or a click of the ToggleButton.
- When reacting to a property change or a method that's triggered through code. (For example, the IsFlipped property setter calls ChangeVisualState() and always supplies true, thereby showing the transition animations. If you want to give the control consumer the choice of not showing the transition, you can add a Flip() method that takes the same Boolean parameter you pass to ChangeVisualState().

As written, the FlipPanel control is remarkably flexible. For example, you can use it without a ToggleButton and flip it programmatically (perhaps when the user clicks a different control). Or, you can include one or two flip buttons in the control template and allow the user to take control.

Using the FlipPanel

Now that you've completed the control template and code for the FlipPanel, you're ready to use it in an application. Assuming you've added the necessary assembly reference, you can then map an XML prefix to the namespace that holds your custom control:

```
<Window x:Class="FlipPanelTest.Page"
  xmlns:lib="clr-namespace:FlipPanelControl;assembly=FlipPanelControl" ... >
```

Next, you can add instances of the FlipPanel to your page. Here's an example that creates the FlipPanel shown earlier in Figure 18-5, using a StackPanel full of elements for the front content region and a Grid for the back:

```
<lib:FlipPanel x:Name="panel" BorderBrush="DarkOrange"
 BorderThickness="3" CornerRadius="4" Margin="10">
  <lib:FlipPanel.FrontContent>
    <StackPanel Margin="6">
      <TextBlock TextWrapping="Wrap" Margin="3" FontSize="16"
       Foreground="DarkOrange">This is the front side of the FlipPanel.</TextBlock>
      <Button Margin="3" Padding="3" Content="Button One"></Button>
      <Button Margin="3" Padding="3" Content="Button Two"></Button>
      <Button Margin="3" Padding="3" Content="Button Three"></Button>
      <Button Margin="3" Padding="3" Content="Button Four"></Button>
    </StackPanel>
  </lib:FlipPanel.FrontContent>

  <lib:FlipPanel.BackContent>
    <Grid Margin="6">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
      </Grid.RowDefinitions>
      <TextBlock TextWrapping="Wrap" Margin="3" FontSize="16"
       Foreground="DarkMagenta">This is the back side of the FlipPanel.</TextBlock>
      <Button Grid.Row="2" Margin="3" Padding="10" Content="Flip Back to Front"
       HorizontalAlignment="Center" VerticalAlignment="Center"
       Click="cmdFlip_Click"></Button>
    </Grid>
  </lib:FlipPanel.BackContent>
</lib:FlipPanel>
```

When clicked, the button on the back side of the FlipPanel programmatically flips the panel:

```
private void cmdFlip_Click(object sender, RoutedEventArgs e)
{
    panel.IsFlipped = !panel.IsFlipped;
}
```

This has the same result as clicking the ToggleButton with the arrow, which is defined as part of the default control template.

Using a Different Control Template

Custom controls that have been designed properly are extremely flexible. In the case of the FlipPanel, you can supply a new template to change the appearance and placement of the ToggleButton and the animated effects that are used when flipping between the front and back content regions.

Figure 18-6 shows one such example. Here, the flip button is placed in a special bar that's at the bottom of the front side and the top of the back side. And when the panel flips, it doesn't turn its content like a sheet of paper. Instead, it squares the front content into nothingness at the top of the panel while simultaneously expanding the back content underneath. When the panel flips the other way, the back content squishes back down, and the front content expands from the top. For even more visual pizzazz, the content that's being squashed is also blurred with the help of the BlurEffect class.

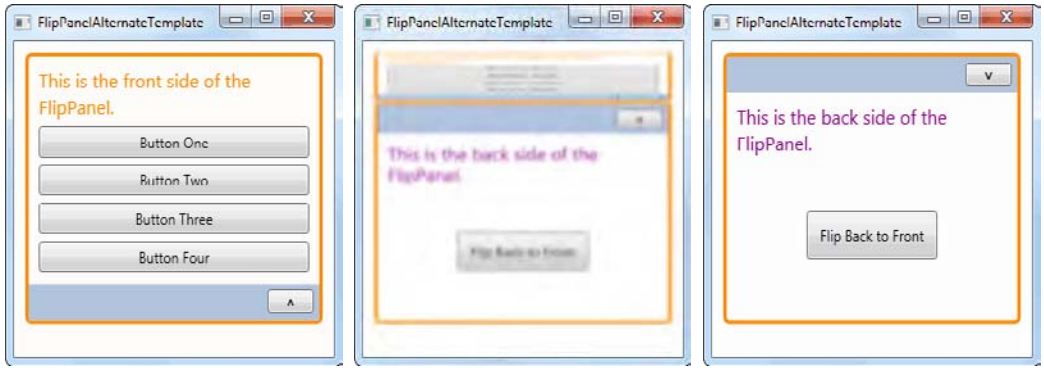

Figure 18-6. *The FlipPanel with a different control template*

Here's the portion of the template that defines the front content region:

```
<Border BorderBrush="{TemplateBinding BorderBrush}"
 BorderThickness="{TemplateBinding BorderThickness}"
 CornerRadius="{TemplateBinding CornerRadius}"
 Background="{TemplateBinding Background}">

  <Border.RenderTransform>
    <ScaleTransform x:Name="FrontContentTransform"></ScaleTransform>
  </Border.RenderTransform>
  <Border.Effect>
    <BlurEffect x:Name="FrontContentEffect" Radius="0"></BlurEffect>
  </Border.Effect>

  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
```

```
    </Grid.RowDefinitions>

    <ContentPresenter Content="{TemplateBinding FrontContent}"></ContentPresenter>
    <Rectangle Grid.Row="1" Stretch="Fill" Fill="LightSteelBlue"></Rectangle>
    <ToggleButton Grid.Row="1" x:Name="FlipButton" Margin="5" Padding="15,0"
     Content="^" FontWeight="Bold" FontSize="12" HorizontalAlignment="Right">
    </ToggleButton>
  </Grid>
</Border>
```

The back content region is almost the same. It consists of a Border that contains a ContentPresenter element, and it includes its own ToggleButton placed at the right edge of the shaded rectangle. It also defines the all-important ScaleTransform and BlurEffect on the Border, which is what the animations use to flip the panel.

Here are the animations that flip the panel. To see all the markup, refer to the downloadable code for this chapter.

```
<VisualState x:Name="Flipped">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="FrontContentTransform"
     Storyboard.TargetProperty="ScaleY" To="0" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="FrontContentEffect"
     Storyboard.TargetProperty="Radius" To="30" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="BackContentTransform"
     Storyboard.TargetProperty="ScaleY" To="1" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="BackContentEffect"
     Storyboard.TargetProperty="Radius" To="0" ></DoubleAnimation>
  </Storyboard>
</VisualState>
```

Because the animation that changes the front content region runs at the same time as the animation that changes the back content region, you don't need a custom transition to manage them.

Custom Panels

So far, you've seen how to develop two custom controls from scratch, with a custom ColorPicker and FlipPanel. In the following sections, you'll consider two more specialized options: deriving a custom Panel and building a custom-drawn control.

Creating a custom panel is a specific but relatively common subset of custom control development. As you learned in Chapter 3, panels host one or more children and implement specific layout logic to arrange them appropriately. Custom panels are an essential ingredient if you want to build your own system for tear-off toolbars or dockable windows. Custom panels are often useful when creating composite controls that need a specific nonstandard layout, like fancy docking toolbars.

You're already familiar with the basic types of panels that WPF includes for organizing content (such as the StackPanel, DockPanel, WrapPanel, Canvas, and Grid). You've also seen that some WPF elements

use their own custom panels (such as the TabPanel, ToolBarOverflowPanel, and VirtualizingPanel). You can find many more examples of custom panels online. Here are some worth exploring:

- A custom Canvas that allows its children to be dragged with no extra event handling code (`http://www.codeproject.com/WPF/DraggingElementsInCanvas.asp`)
- Two panels that implements fisheye and fanning effects on a list of items (`http://www.codeproject.com/WPF/Panels.asp`)
- A panel that uses a frame-based animation to transition from one layout to another (`http://j832.com/BagOTricks`)

In the next sections, you'll learn how to create a custom panel, and you'll consider two straightforward examples—a basic Canvas clone and an enhanced version of the WrapPanel.

The Two-Step Layout Process

Every panel uses the same plumbing: a two-step process that's responsible for sizing and arranging children. The first stage is the *measure* pass, and it's at this point that the panel determines how large its children want to be. The second stage is the *layout* pass, and it's at this point that each control is assigned its bounds. Two steps are required because the panel might need to take into account the desires of all its children before it decides how to partition the available space.

You add the logic for these two steps by overriding the oddly named MeasureOverride() and ArrangeOverride() methods, which are defined in the FrameworkElement class as part of the WPF layout system. The odd names represent that the MeasureOverride() and ArrangeOverride() methods replace the logic that's defined in the MeasureCore() and ArrangeCore() methods that are defined in the UIElement class. These methods are *not* overridable.

MeasureOverride()

The first step is to determine how much space each child wants using the MeasureOverride() method. However, even in the MeasureOverride() method children aren't given unlimited room. At a bare minimum, children are confined to fit in the space that's available to the panel. Optionally, you might want to limit them more stringently. For example, a Grid with two proportionally sized rows will give children half the available height. A StackPanel will offer the first element all the space that's available, then offer the second element whatever's left, and so on.

Every MeasureOverride() implementation is responsible for looping through the collection of children and calling the Measure() method of each one. When you call the Measure() method, you supply the bounding box—a Size object that determines the maximum available space for the child control. At the end of the MeasureOverride() method, the panel returns the space it needs to display all its children and their desired sizes.

Here's the basic structure of the MeasureOverride() method, without the specific sizing details:

```
protected override Size MeasureOverride(Size constraint)
{
    // Examine all the children.
    foreach (UIElement element in base.InternalChildren)
    {
        // Ask each child how much space it would like, given the
```

```
        // availableSize constraint.
        Size availableSize = new Size(...);
        element.Measure(availableSize);
        // (You can now read element.DesiredSize to get the requested size.)
    }

    // Indicate how much space this panel requires.
    // This will be used to set the DesiredSize property of the panel.
    return new Size(...);
}
```

The Measure() method doesn't return a value. After you call Measure() on a child, that child's DesiredSize property provides the requested size. You can use this information in your calculations for future children (and to determine the total space required for the panel).

You *must* call Measure() on each child, even if you don't want to constrain the child's size or use the DesiredSize property. Many elements will not render themselves until you've called Measure(). If you want to give a child free reign to take all the space it wants, pass a Size object with a value of Double.PositiveInfinity for both dimensions. (The ScrollViewer is one element that uses this strategy, because it can handle any amount of content.) The child will then return the space it needs for all its content. Otherwise, the child will normally return the space it needs for its content or the space that's available—whichever is smaller.

At the end of the measuring process, the layout container must return its desired size. In a simple panel, you might calculate the panel's desired size by combining the desired size of every child.

■ **Note** You can't simply return the constraint that's passed to the MeasureOverride() method for the desired size of your panel. Although this seems like a good way to take all the available size, it runs into trouble if the container passes in a Size object with Double.PositiveInfinity for one or both dimensions (which means "take all the space you want"). Although an infinite size is allowed as a sizing constraint, it's not allowed as a sizing result, because WPF won't be able to figure out how large your element should be. Furthermore, you really shouldn't take more space than you need. Doing so can cause extra whitespace and force elements that occur after your layout panel to be bumped further down the window.

Attentive readers may have noticed that there's a close similarity between the Measure() method that's called on each child and the MeasureOverride() method that defines the first step of the panel's layout logic. In fact, the Measure() method triggers the MeasureOverride() method. Thus, if you place one layout container inside another, when you call Measure(), you'll get the total size required for the layout container and all its children.

■ **Tip** One reason the measuring process goes through two steps (a Measure() method that triggers the MeasureOverride() method) is to deal with margins. When you call Measure(), you pass in the total available space. When WPF calls the MeasureOverride() method, it automatically reduces the available space to take margin space into account (unless you've passed in an infinite size).

ArrangeOverride()

Once every element has been measured, it's time to lay them out in the space that's available. The layout system calls the ArrangeOverride() method of your panel, and the panel calls the Arrange() method of each child to tell it how much space it's been allotted. (As you can probably guess, the Arrange() method triggers the ArrangeOverride() method, much as the Measure() method triggers the MeasureOverride() method.)

When measuring items with the Measure() method, you pass in a Size object that defines the bounds of the available space. When placing an item with the Arrange() method, you pass in a System.Windows.Rect object that defines the size *and* position of the item. At this point, it's as though every element is placed with Canvas-style X and Y coordinates that determine the distance between the top-left corner of your layout container and the element.

■ **Note** Elements (and layout panels) are free to break the rules and attempt to draw outside of their allocated bounds. For example, in Chapter 12 you saw how the Line can overlap adjacent items. However, ordinary elements should respect the bounds they're given. Additionally, most containers will clip children that extend outside their bounds.

Here's the basic structure of the ArrangeOverride() method, without the specific sizing details:

```
protected override Size ArrangeOverride(Size arrangeSize)
{
    // Examine all the children.
    foreach (UIElement element in base.InternalChildren)
    {
        // Assign the child its bounds.
        Rect bounds = new Rect(...);
        element.Arrange(bounds);
        // (You can now read element.ActualHeight and element.ActualWidth
        //  to find out the size it used.)
    }

    // Indicate how much space this panel occupies.
    // This will be used to set the ActualHeight and ActualWidth properties
    // of the panel.
    return arrangeSize;
}
```

When arranging elements, you can't pass infinite sizes. However, you can give an element its desired size by passing in the value from its DesiredSize property. You can also give an element *more* space than it requires. In fact, this happens frequently. For example, a vertical StackPanel gives a child as much height as it requests but gives it the full width of the panel itself. Similarly, a Grid might use fixed or proportionally sized rows that are larger than the desired size of the element inside. And even if you've placed an element in a size-to-content container, that element can still be enlarged if an explicit size has been set using the Height and Width properties.

When an element is made larger than its desired size, the HorizontalAlignment and VerticalAlignment properties come into play. The element content is placed somewhere inside the bounds that it has been given.

Because the ArrangeOverride() method always receives a defined size (not an infinite size), you can return the Size object that's passed in to set the final size of your panel. In fact, many layout containers take this step to occupy all the space that's been given. (You aren't in danger of taking up space that could be needed for another control, because the measure step of the layout system ensures that you won't be given more space than you need unless that space is available.)

The Canvas Clone

The quickest way to get a grasp of these two methods is to explore the inner workings of the Canvas class, which is the simplest layout container. To create your own Canvas-style panel, you simply need to derive from Panel and add the MeasureOverride() and ArrangeOverride() methods shown next:

```
public class CanvasClone : System.Windows.Controls.Panel
{ ... }
```

The Canvas places children where they want to be placed and gives them the size they want. As a result, it doesn't need to calculate how the available space should be divided. That makes its MeasureOverride() method extremely simple. Each child is given infinite space to work with:

```
protected override Size MeasureOverride(Size constraint)
{
    Size size = new Size(double.PositiveInfinity, double.PositiveInfinity);
    foreach (UIElement element in base.InternalChildren)
    {
        element.Measure(size);
    }
    return new Size();
}
```

Notice that the MeasureOverride() returns an empty Size object, which means the Canvas doesn't request any space at all. It's up to you to specify an explicit size for the Canvas or place it in a layout container that will stretch it to fill the available space.

The ArrangeOverride() method is only slightly more involved. To determine the proper placement of each element, the Canvas uses attached properties (Left, Right, Top, and Bottom). As you learned in Chapter 4 (and as you'll see in the WrapBreakPanel next), attached properties are implemented with two helper methods in the defining class: a Get*Property*() and a Set*Property*() method.

The Canvas clone that you're considering is a bit simpler—it respects only the Left and Top attached properties (not the redundant Right and Bottom properties). Here's the code it uses to arrange elements:

```
protected override Size ArrangeOverride(Size arrangeSize)
{
    foreach (UIElement element in base.InternalChildren)
    {
        double x = 0;
        double y = 0;
        double left = Canvas.GetLeft(element);
        if (!DoubleUtil.IsNaN(left))
        {
```

```
            x = left;
        }
        double top = Canvas.GetTop(element);
        if (!DoubleUtil.IsNaN(top))
        {
            y = top;
        }
        element.Arrange(new Rect(new Point(x, y), element.DesiredSize));
    }
    return arrangeSize;
}
```

A Better Wrapping Panel

Now that you've examined the panel system in a fair bit of detail, it's worth creating your own layout container that adds something you can't get with the basic set of WPF panels. In this section, you'll see an example that extends the capabilities of the WrapPanel.

The WrapPanel performs a simple function that's occasionally quite useful. It lays out its children one after the other, moving to the next line once the width in the current line is used up. Windows Forms included a similar layout tool, called the FlowLayoutPanel. Unlike the WrapPanel, the FlowLayoutPanel added one extra ability—an attached property that children could use to force an immediate line break. (Technically, this wasn't an attached property but a property that's added through an extender provider, but the two concepts are analogous.)

Although the WrapPanel doesn't provide this capability, it's fairly easy to add one. All you need is a custom panel that adds the necessary attached property. The following listing shows a WrapBreakPanel that adds an attached LineBreakBeforeProperty. When set to true, this property causes an immediate line break before the element.

```
public class WrapBreakPanel : Panel
{
    public static DependencyProperty LineBreakBeforeProperty;

    static WrapBreakPanel()
    {
        FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
        metadata.AffectsArrange = true;
        metadata.AffectsMeasure = true;
        LineBreakBeforeProperty = DependencyProperty.RegisterAttached(
          "LineBreakBefore", typeof(bool), typeof(WrapBreakPanel), metadata);
    }
    ...
}
```

As with any dependency property, the LineBreakBefore property is defined as a static field and then registered in the static constructor for your class. The only difference is that you use the RegisterAttached() method rather than Register().

The FrameworkPropertyMetadata object for the LineBreakBefore property specifically indicates that it affects the layout process. As a result, a new layout pass will be triggered whenever this property is set.

Attached properties aren't wrapped by normal property wrappers, because they aren't set in the same class that defines them. Instead, you need to provide two static methods that can use the

DependencyObject.SetValue() method to set this property on any arbitrary element. Here's the code that you need for the LineBreakBefore property:

```
public static void SetLineBreakBefore(UIElement element, Boolean value)
{
    element.SetValue(LineBreakBeforeProperty, value);
}
public static Boolean GetLineBreakBefore(UIElement element)
{
    return (bool)element.GetValue(LineBreakBeforeProperty);
}
```

The only remaining detail is to take this property into account when performing the layout logic. The layout logic of the WrapBreakPanel is based on the WrapPanel. During the measure stage, elements are arranged into lines so that the panel can calculate the total space it needs. Each element is added into the current line unless it's too large or the LineBreakBefore property is set to true. Here's the full code:

```
protected override Size MeasureOverride(Size constraint)
{
    Size currentLineSize = new Size();
    Size panelSize = new Size();

    foreach (UIElement element in base.InternalChildren)
    {
        element.Measure(constraint);
        Size desiredSize = element.DesiredSize;

        if (GetLineBreakBefore(element) ||
          currentLineSize.Width + desiredSize.Width > constraint.Width)
        {
            // Switch to a new line (either because the element has requested it
            // or space has run out).
            panelSize.Width = Math.Max(currentLineSize.Width, panelSize.Width);
            panelSize.Height += currentLineSize.Height;
            currentLineSize = desiredSize;

            // If the element is too wide to fit using the maximum width
            // of the line, just give it a separate line.
            if (desiredSize.Width > constraint.Width)
            {
                panelSize.Width = Math.Max(desiredSize.Width, panelSize.Width);
                panelSize.Height += desiredSize.Height;
                currentLineSize = new Size();
            }
        }
        else
        {
            // Keep adding to the current line.
            currentLineSize.Width += desiredSize.Width;

            // Make sure the line is as tall as its tallest element.
            currentLineSize.Height = Math.Max(desiredSize.Height,
              currentLineSize.Height);
```

```
            }
        }

        // Return the size required to fit all elements.
        // Ordinarily, this is the width of the constraint, and the height
        // is based on the size of the elements.
        // However, if an element is wider than the width given to the panel,
        // the desired width will be the width of that line.
        panelSize.Width = Math.Max(currentLineSize.Width, panelSize.Width);
        panelSize.Height += currentLineSize.Height;
        return panelSize;
    }
```

The key detail in this code is the test that checks the LineBreakBefore property. This implements the additional logic that's not provided in the ordinary WrapPanel.

The code for the ArrangeOverride() is almost the same but slightly more tedious. The difference is that the panel needs to determine the maximum height of the line (which is determined by the tallest element) before it begins laying out that line. That way, each element can be given the full amount of available space, which takes into account the full height of the line. This is the same process that's used to lay out an ordinary WrapPanel. To see the full details, refer to the downloadable code examples for this chapter.

Using the WrapBreakPanel is easy. Here's some markup that demonstrates that the WrapBreakPanel correctly separates lines and calculates the right desired size based on the size of its children:

```
<StackPanel>
  <StackPanel.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Margin" Value="3"></Setter>
      <Setter Property="Padding" Value="3"></Setter>
    </Style>
  </StackPanel.Resources>

  <TextBlock Padding="5" Background="LightGray">
    Content above the WrapBreakPanel.
  </TextBlock>
  <lib:WrapBreakPanel>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button lib:WrapBreakPanel.LineBreakBefore="True" FontWeight="Bold">
     Button with Break
    </Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
  </lib:WrapBreakPanel>
  <TextBlock Padding="5" Background="LightGray">
    Content below the WrapBreakPanel.
  </TextBlock>
</StackPanel>
```

Figure 18-7 shows how this markup is interpreted.

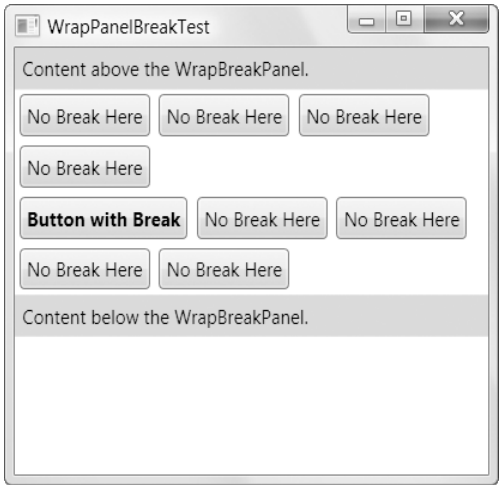

Figure 18-7. The WrapBreakPanel

Custom-Drawn Elements

In the previous section, you began to explore the inner workings of WPF elements—namely, the MeasureOverride() and ArrangeOverride() methods that allow every element to plug into WPF's layout system. In this section, you'll delve a bit deeper and consider how elements render themselves.

Most WPF elements use *composition* to create their visual appearance. In other words, a typical element builds itself out of other, more fundamental elements. You've seen this pattern at work throughout this chapter. For example, you define the composite elements of a user control using markup that's processed in the same way as the XAML in a custom window. You define the visual tree for a custom control using a control template. And when creating a custom panel, you don't need to define any visual details at all. The composite elements are provided by the control consumer and added to the Children collection.

This emphasis is different from what you see in previous user interface technologies such as Windows Forms. In Windows Forms, some controls draw themselves using the User32 library that's part of the Windows API, but most custom controls rely on the GDI+ drawing classes to render themselves from scratch. Because Windows Forms doesn't provide high-level graphical primitives that can be added directly to a user interface (like WPF's rectangles, ellipses, and paths), any control that needs a nonstandard visual appearance requires custom rendering code.

Of course, composition can take you only so far. Eventually, some class needs to take responsibility for drawing content. In WPF, this point is a long way down the element tree. In a typical window, the rendering is performed by individual bits of text, shapes, and bitmaps, rather than high-level elements.

The OnRender() Method

To perform custom rendering, an element must override the OnRender() method, which is inherited from the base UIElement class. The OnRender() method doesn't necessarily replace composition—some controls use OnRender() to paint a visual detail and use composition to layer other elements over it. Two examples are the Border class, which draws its border in the OnRender() method, and the Panel class, which draws its background in the OnRender() method. Both the Border and Panel support child content, and this content is rendered overtop the custom-drawn details.

The OnRender() method receives a DrawingContext object, which provides a set of useful methods for drawing content. You first learned about the DrawingContext class in Chapter 14, when you used it to draw the content for a Visual object. The key difference when performing drawing in the OnRender() method is that you don't explicitly create and close the DrawingContext. That's because several different OnRender() methods could conceivable use the same DrawingContext. For example, a derived element might perform some custom drawing and call the OnRender() implementation in the base class to draw additional content. This works because WPF automatically creates the DrawingContext object at the beginning of this process and closes it when it's no longer needed.

■ **Note** Technically, the OnRender() method doesn't actually *draw* your content to the screen. Instead, it draws your content to the DrawingContext object, and WPF then caches that information. WPF determines when your element needs to be repainted and paints the content that you created with the DrawingContext. This is the essence of WPF's retained graphics system—you define the content, and it manages the painting and refreshing process seamlessly.

The most surprising detail about WPF rendering is that so few classes actually do it. Most classes are built out of other simpler classes, and you need to dig quite a way down the element tree of a typical control before you discover a class that actually overrides OnRender(). Here are some that do:

- **The TextBlock class.** Wherever you place text, there's TextBlock object using OnRender() to draw it.
- **The Image class.** The Image class overrides OnRender() to paint its image content using the DrawingContext.DrawImage() method.
- **The MediaElement class.** The MediaElement overrides OnRender() to draw a frame of video, if it's being used to play a video file.
- **The shape classes.** The base Shape class overrides OnRender() to draw its internally stored Geometry object, with the help of the DrawingContext.DrawGeometry() method. This Geometry object could represent an ellipse, a rectangle, or a more complex path composed of lines and curves, depending on the specific Shape-derived class. Many elements use shapes to draw small visual details.

- **The chrome classes.** Classes such as ButtonChrome and ListBoxChrome draw the outer appearance of a common control and place the content you specify inside. Many other Decorator-derived classes, such as Border, also override OnRender().
- **The panel classes.** Although the content of a panel is supplied by its children, the OnRender() method takes care of drawing a rectangle with the background color if the Background property is set.

Often, the OnRender() implementation is deceptively simple. For example, here's the rendering code for any Shape-derived class:

```
protected override void OnRender(DrawingContext drawingContext)
{
    this.EnsureRenderedGeometry();
    if (this._renderedGeometry != Geometry.Empty)
    {
        drawingContext.DrawGeometry(this.Fill, this.GetPen(),
          this._renderedGeometry);
    }
}
```

Remember, overriding OnRender() isn't the only way to render content and add it to your user interface. You can also create a DrawingVisual object and add that visual to a UIElement using the AddVisualChild() method (and implementing a few other details, as described in Chapter 14). You can then call DrawingVisual.RenderOpen() to retrieve a DrawingContext for your DrawingVisual and use it to render its content.

Some elements use this strategy in WPF to display some graphical detail overtop other element content. For example, you'll see it with drag-and-drop indicators, error indicators, and focus boxes. In all these cases, the DrawingVisual approach allows the element to draw content *over* other content, rather than *under* it. But for the most part, rendering takes place in the dedicated OnRender() method.

Evaluating Custom Drawing

When you create your own custom elements, you may choose to override OnRender() to draw custom content. You might override OnRender() in an element that contains content (most commonly, a Decorator-derived class) so you can add a graphical embellishment around that content. Or, you might override OnRender() in an element that doesn't have any nested content so that you can draw its full visual appearance. For example, you might create a custom element that draws a small graphical detail, which you can then use in another control through composition. One example in WPF is the TickBar element, which draws the tick marks for a Slider. The TickBar is embedded in the visual tree of a Slider through the Slider's default control template (along with a Border and a Track that includes two RepeatButton controls and a Thumb).

The obvious question is when to use the comparatively low-level OnRender() approach and when to use composition with other classes (such as the Shape-derived elements) to draw what you need. To decide, you need to evaluate the complexity of the graphics you need and the interactivity you want to provide.

For example, consider the ButtonChrome class. In WPF's implementation of the ButtonChrome class, the custom rendering code takes various properties into account, including RenderDefaulted, RenderMouseOver, and RenderPressed. The default control template for the Button uses triggers to set these properties at the appropriate time, as you saw in Chapter 17. For example, when the mouse moves over the button, the Button class uses a trigger to set the ButtonChrome.RenderMouseOver property to true.

Whenever the RenderDefaulted, RenderMouseOver, or RenderPressed property is changed, the ButtonChrome calls the base InvalidateVisual() method to indicate that its current appearance is no longer valid. WPF then calls the ButtonChrome.OnRender() method to get its new graphical representation.

If the ButtonChrome class used composition, this behavior would be more difficult to implement. It's easy enough to create the standard appearance for the ButtonChrome class using the right elements, but it's more work to modify it when the button's state changes. You'd need to dynamically change the nested elements that compose the ButtonChrome class or—if the appearance changes more dramatically—you'd be forced to hide one element and show another one in its place.

Most custom elements won't need custom rendering. But if you need to render complex visuals that change significantly when properties are changed or certain actions take place, the custom rendering approach just might be easier to use and more lightweight.

A Custom-Drawn Element

Now that you know how the OnRender() method works and when to use it, the last step is to consider a custom control that demonstrates it in action.

The following code defines an element named CustomDrawnElement that demonstrates a simple effect. It paints a shaded background using the RadialGradientBrush. The trick is that the highlight point where the gradient starts is set dynamically, so it follows the mouse. Thus, as the user moves the mouse over the control, the white glowing center point follows, as shown in Figure 18-8.

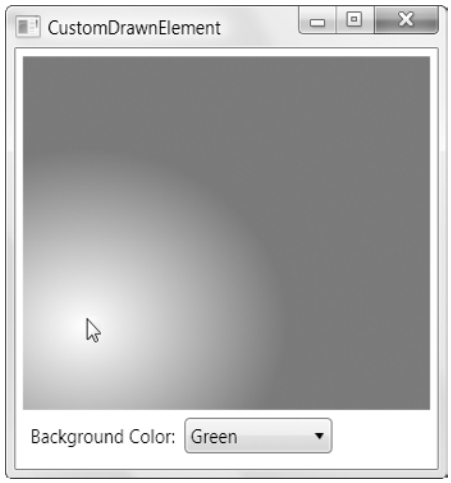

Figure 18-8. *A custom-drawn element*