The XAML parser follows two steps to find a type converter:

1. It examines the property declaration, looking for a TypeConverter attribute. (If

present, the TypeConverter attribute indicates what class can perform the

conversion.) For example, when you use a property such as Foreground, .NET

checks the declaration of the Foreground property.

2. If there’s no TypeConverter attribute on the property declaration, the XAML

parser checks the class declaration of the corresponding data type. For example,

the Foreground property uses a Brush object. The Brush class (and its

derivatives) use the BrushConverter because the Brush class is decorated with

the TypeConverter(typeof(BrushConverter)) attribute declaration.*converters*

type converters usually aren’t case-sensitive, which means both Foreground=”White” and

Foreground=”white” have the same result

……

Fortunately, XAML provides another option: ***property-element syntax***. With property-element syntax,

you add a child element with a name in the form Parent.PropertyName. For example, the Grid has a

Background property that allows you to supply a brush that’s used to paint the area behind the controls. If

you want to use a complex brush—one more advanced than a solid-color fill—you’ll need to add a child

tag named Grid.Background, as shown here:

<Grid Name="grid1">

<Grid.Background>

...

</Grid.Background>

...

</Grid>

Pro WPF 4.5 in C# -- p.31

This still leaves one detail—namely, after you’ve identified the complex property you want to

configure, how do you set it? Here’s the trick: inside the nested element, you can add another tag to

instantiate a specific class. In the eight-ball example (shown in Figure 2-1), the background is filled with a

gradient. To define the gradient you want, you need to create a LinearGradientBrush object.

Using the rules of XAML, you can create the LinearGradientBrush object by using an element with the

name LinearGradientBrush:

<Grid Name="grid1">

<Grid.Background>

<LinearGradientBrush>

</LinearGradientBrush>

</Grid.Background>

...

</Grid>

The LinearGradientBrush class is part of the WPF set of namespaces, so you can keep using the default

XML namespace for your tags.

However, it’s not enough to simply create the LinearGradientBrush object—you also need to specify

the colors in that gradient. You do this by filling the LinearGradientBrush.GradientStops property with a

collection of GradientStop objects. Once again, the GradientStops property is too complex to be set with

an attribute value alone. Instead, you need to rely on the property-element syntax:

<Grid Name="grid1">

<Grid.Background>

<LinearGradientBrush>

<LinearGradientBrush.GradientStops>

</LinearGradientBrush.GradientStops>

</LinearGradientBrush>

</Grid.Background>

...

</Grid>

Finally, you can fill the GradientStops collection with a series of GradientStop objects. Each

GradientStop object has an Offset and Color property. You can supply these two values by using the

ordinary property-attribute syntax:

<Grid Name="grid1">

<Grid.Background>

<LinearGradientBrush>

<LinearGradientBrush.GradientStops>

<GradientStop Offset="0.00" Color="Red" />

<GradientStop Offset="0.50" Color="Indigo" />

<GradientStop Offset="1.00" Color="Violet" />

</LinearGradientBrush.GradientStops>

</LinearGradientBrush>

</Grid.Background>

...

</Grid>

You can use property-element syntax for any property. But usually you’ll use the simpler property-attribute

approach if the property has a suitable type converter. Doing so results in more compact code

p.32

Markup Extensions

For most properties, the XAML property syntax works perfectly well. But in some cases, it just isn’t possible

to hard-code the property value. For example, you may want to set a property value to an object that

already exists. Or you may want to set a property value *dynamically* by binding it to a property in another

control. In both of these cases, you need to use a *markup extension*—specialized syntax that sets a

property in a nonstandard way.

Markup extensions can be used in nested tags or in XML attributes, which is more common. When

they’re used in attributes, they are always bracketed by curly braces {}. For example, here’s how you can

use, which allows you to refer to a static property in another class:

<Button ... Foreground="{x:Static SystemColors.ActiveCaptionBrush}" >

Markup extensions use the syntax {MarkupExtensionClass Argument}. In this case, the markup

extension is the StaticExtension class. (By convention, you can drop the final word *Extension* when

referring to an extension class.) The x prefix indicates that StaticExtension is found in one of the XAML

namespaces. You’ll also encounter markup extensions that are part of the WPF namespaces and don’t have

the x prefix.

All markup extensions are implemented by classes that derive from **System.Windows.Markup.**

**MarkupExtension**. The base MarkupExtension class is extremely simple—it provides a single **ProvideValue**

method that gets the value you want. In other words, when the XAML parser encounters the previous

statement, it creates an instance of the StaticExtension class (passing in the string “SystemColors.

ActiveCaptionBrush” as an argument to the constructor) and then calls ProvideValue() to get the object

returned by the SystemColors.ActiveCaption.Brush static property. The Foreground property of the

cmdAnswer button is then set with the retrieved object.

The end result of this piece of XAML is the same as if you’d written this:

cmdAnswer.Foreground = SystemColors.ActiveCaptionBrush;

p.33

Like most markup extensions, StaticExtension needs to be evaluated at runtime because only then

can you determine the current system colors. Some markup extensions can be evaluated at compile time.

These include NullExtension (which constructs an object that represents a .NET type). Throughout this

book, you’ll see many examples of markup extensions at work, particularly with resources and data

binding.

Attached Properties

Along with ordinary properties, XAML also includes the concept of ***attached properties***—properties that

may apply to several controls but are defined in a different class. In WPF, attached properties are frequently

used to control layout.

Attached properties always use a two-part name in this form: DefiningType.PropertyName. This twopart

naming syntax allows the XAML parser to distinguish between a normal property and an attached

property.

Attached properties aren’t really properties at all. They’re translated into method calls. The XAML

parser calls the static method that has this form: *DefiningType*.Set*PropertyName*(). For example, in the

previous XAML snippet, the defining type is the Grid class, and the property is Row, so the parser calls

Grid.SetRow().

p.34

When calling SetPropertyName(), the parser passes two parameters: the object that’s being modified

and the property value that’s specified. For example, when you set the Grid.Row property on the TextBox

control, the XAML parser executes this code:

Grid.SetRow(txtQuestion, 0);

This pattern (calling a static method of the defining type) is a convenience that conceals what’s really

taking place. To the casual eye, this code implies that the row number is stored in the Grid object. However,

the row number is actually stored in the object that it *applies to*—in this case, the TextBox object.

This sleight of hand works because TextBox derives from the **DependencyObject** base class, as do all

WPF controls. And as you’ll learn in Chapter 4, the DependencyObject is designed to store a virtually

**unlimited collection of dependency properties**. (The attached properties that were discussed earlier are a

special type of dependency property.)

In fact, the Grid.SetRow() method is a shortcut that’s equivalent to calling the DependencyObject.

SetValue() method, as shown here:

txtQuestion.SetValue(Grid.RowProperty, 0);

p.35

XAML allows each element to decide how it deals with nested elements. This interaction is mediated

through one of three mechanisms that are evaluated in this order:

• If the parent implements IList, the parser calls IList.Add() and passes in the child.

• If the parent implements IDictionary, the parser calls IDictionary.Add() and passes

in the child. When using a dictionary collection, you must also set the x:Key attribute

to give a key name to each item.

• If the parent is decorated with the ContentProperty attribute, the parser uses the

child to set that property.

……

<Grid Name="grid1">

...

<TextBox Name="txtQuestion" ... >

...

</TextBox>

<Button Name="cmdAnswer" ... >

...

</Button>

<TextBox Name="txtAnswer" ... >

...

</TextBox>

</Grid>

What the Grid *does* support is the **ContentProperty** attribute, which indicates the property that should **receive any**

**nested content**. Technically, the ContentProperty attribute is applied to the Panel class, from which the

Grid derives, and looks like this:

p.36

[ContentPropertyAttribute("**Children**")]

public abstract class Panel

This indicates that any **nested elements** should be used to set the **Children** property. The XAML parser

treats the content property differently depending on whether it’s a collection property (in which case it

implements the IList or IDictionary interface). Because the Panel.Children property returns a

UIElementCollection and because UIElementCollection implements IList, the parser uses the IList.Add()

method to add nested content to the grid.

In other words, when the XAML parser meets the previous markup, it creates an instance of each

nested element and passes it to the Grid by using the Grid.Children.Add() method:

The ContentProperty attribute is frequently used in WPF. Not only is it used for container controls

(such as Grid) and controls that contain a collection of visual items (such as ListBox and TreeView), it’s also

used for controls that contain singular content. For example, the TextBox and Button controls are able to

hold only a single element or piece of text, but they both use a content property to deal with nested

content like this:

<TextBox Name="txtQuestion" ... >

[Place question here.]

</TextBox>

<Button Name="cmdAnswer" ... >

Ask the Eight Ball

</Button>

<TextBox Name="txtAnswer" ... >

[Answer will appear here.]

</TextBox>

The TextBox class uses the ContentProperty attribute to flag the TextBox.Text property. The Button

class uses the ContentProperty attribute to flag the Button.Content property. The XAML parser uses the

supplied text to set these properties.

p.37

**Note** As a general rule of thumb, all controls that derive from ContentControl allow a single nested element*(to set to its ContentPropertyAttribute flagged properpery*). All controls that derive from ItemsControl allow a collection of items that map to some part of the control (such as a list of items or a tree of nodes). All controls that derive from Panel are containers that are used to organize groups of controls. The ContentControl, ItemsControl, and Panel base classes all use the ContentProperty attribute.

The problem here is that it looks like you’re trying to create an element named Click with an attribute

named Me. The solution is to replace the offending characters with entity references—specific codes that

the XAML parser will interpret correctly. Table 2-1 lists the character entities you might choose to use. Note

that the quotation mark character entity is required only when setting values by using an attribute,

because the quotation mark indicates the beginning and ending of an attribute value.

*Table 2-1. XML Character Entities*

**Special Character Character Entity**

Less than (<) &lt;

Greater than (>) &gt;

Ampersand (&) &amp;

Quotation mark (“) &quot;

p.38

Special characters aren’t the only stumbling block you’ll run into with XAML. Another issue is

whitespace handling By default, XML collapses all whitespace, which means a long string of spaces, tabs,

and hard returns is reduced to a single space. Furthermore, if you add whitespace before or after your

element content, this space is ignored completely

…….

The xml:space attribute is part of the XML standard, and it’s an all-or-nothing setting. After you switch

it on, all the whitespace inside that element is retained. For example, consider this markup:

<TextBox Name="txtQuestion" xml:space="preserve" ...>

\_\_\_\_\_\_\_[There is a lot of space inside these quotation marks " ".]\_\_\_\_\_

</TextBox>

……

Events

So far, all the attributes you’ve seen map to properties. However, attributes can also be used to attach event

handlers. The syntax for this is EventName=”EventHandlerMethodName”.

For example, the Button control provides a Click event. You can attach an event handler like this:

<Button ... Click="cmdAnswer\_Click">

……

p.39

In many situations, you’ll use attributes to set properties and attach event handlers on the same

element. **WPF always follows the same sequence: first it sets the Name property (if set); then it attaches any**

**event handlers; and finally it sets the properties**. This means that any event handlers that respond to

property changes will fire when the property is set for the first time.

P.40

To use a class that isn’t defined in one of the WPF namespaces, you need to map the .NET namespace

to an XML namespace. XAML has a special syntax for doing this, which looks like this:

xmlns:*Prefix*="clr-namespace:*Namespace*;assembly=*AssemblyName*"

xmlns:sys="clr-namespace:System;assembly=mscorlib"

xmlns:local="clr-namespace:MyNamespace"

Now, to create an instance of a class in one of these namespaces, you use the namespace prefix:

<local:MyObject ...></local:MyObject>

……

p.40

Ideally, every class you want to use in XAML will have a no-argument constructor. If it does, the XAML

parser can create the corresponding object, set its properties, and attach any event handlers you supply.

XAML doesn’t support parameterized constructors, and all the elements in WPF elements include a noargument

constructor. Additionally, you need to be able to set all the details you want by using public

properties. XAML doesn’t allow you to set public fields or call methods.

…….

If the class you want to use doesn’t have a no-argument constructor, you’re in a bit of a bind. If you’re

trying to create a simple primitive (such as a string, date, or numeric type), you can supply the string

representation of your data as content inside your tag. The XAML parser will then use the type converter to

convert that string into the appropriate object. Here’s an example with the DateTime structure:

p.42

<sys:DateTime>10/30/2013 4:30 PM</sys:DateTime>

This works because the DateTime class uses the TypeConverter attribute to link itself to the

DateTimeConverter. The DateTimeConverter recognizes this string as a valid DateTime object and

converts it. When you’re using this technique, you can’t use attributes to set any properties for your object.

If you want to create a class that doesn’t have a no-argument constructor and there isn’t a suitable type

converter to use, you’re out of luck.

p.43

The **DependencyObject can be placed inside any type of container** (for example, a Panel), but in this example it’s used as the entire content inside the window

p.47

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.

p.53

• *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.

• *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.

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.

p.54

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.

*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 nonnull

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.

p. 55

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.

*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.

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).

Simple Layout with the StackPanel

p.56

**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.

p.57

Layout Properties

Although layout is determined by the container, the **child elements(***below are properties of child element***)** 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.

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>

p.59

Margin

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>

p.60

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.

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.

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.

p.63

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.

p.64

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.

p.64



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.

<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.



p.66

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 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>

p.67

<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*

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.

p.69

Chapter 3 ■ Layout

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.

The 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.

p. 71

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.

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.

p.72

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>

f 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.

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.

p. 73

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">

p. 74

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.

Here’s how all the pieces come into play when an application attempts to set a dependency property:

1. First, the CoerceValueCallback method has the opportunity to modify the

supplied value (usually, to make it consistent with other properties) or return

DependencyProperty.UnsetValue, which rejects the change altogether.

2. Next, the ValidateValueCallback is fired. This method returns true to accept a

value as valid or returns false to reject it. Unlike the CoerceValueCallback, the

ValidateValueCallback does not have access to the actual object on which the

property is being set, which means you can’t examine other property values.

3. Finally, if both these previous stages succeed, the PropertyChangedCallback is

triggered. At this point, you can raise a change event if you want to provide

notification to other classes.

p. 101