DataGridHyperlinkColumn

The DataGridHyperlinkColumn allows you to display text values that contain a single URL each. For example, if the Product class has a string property named ProductLink, and that property contained values like http://myproducts.com/info?productID=10432, you could display this information in a DataGridHyperlinkColumn. Every bound value would be displayed using the Hyperlink element, and rendered like this:

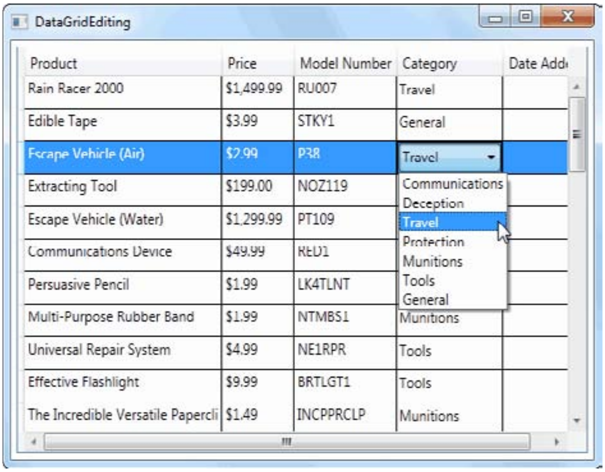
```
<Hyperlink NavigateUri="http://myproducts.com/info?productID=10432"
 >http://myproducts.com/info?productID=10432</Hyperlink>
```

Then the user could click a hyperlink to trigger navigation and visit the related page, with no code required. However, there's a major caveat: this automatic navigation trick works only if you've placed your DataGrid in a container that supports navigation events, like the Frame or NavigationWindow. You'll learn about both controls and the Hyperlink in Chapter 24. If you want a more versatile way to accomplish a similar effect, consider using the DataGridTemplateColumn. You can use it to show underlined, clickable text (in fact, you can even use the Hyperlink control), but you'll have the flexibility of handling click events in your code.

Ordinarily, the DataGridHyperlinkColumn uses the same piece of information for navigation and for display. However, you can specify these details separately if you want. To do so, just set the URI with the Binding property, and use the optional ContentBinding property to get display text from a different property in the bound data object.

DataGridComboBoxColumn

The DataGridComboBoxColumn shows ordinary text initially, but provides a streamlined editing experience that allows the user to pick from a list of available options in a ComboBox control. (In fact, the user will be forced to choose from the list, as the ComboBox does not allow direct text entry.) Figure 22-8 shows an example where the user is choosing the product category from a DataGridComboBoxColumn.



Product	Price	Model Number	Category	Date Add
Rain Racer 2000	$1,499.99	RU007	Travel	
Edible Tape	$3.99	STKY1	General	
Escape Vehicle (Air)	$2.99	P38	Travel	
Extracting Tool	$199.00	NOZ119		
Escape Vehicle (Water)	$1,299.99	PT109		
Communications Device	$49.99	RED1		
Persuasive Pencil	$1.99	LK4TLNT		
Multi-Purpose Rubber Band	$1.99	NTMBS1	Munitions	
Universal Repair System	$4.99	NE1RPR	Tools	
Effective Flashlight	$9.99	BRTLGT1	Tools	
The Incredible Versatile Papercli	$1.49	INCPPRCLP	Munitions	

Figure 22-8. Choosing from a list of allowed values

To use the DataGridComboBoxColumn, you need to decide how to populate the combo box in edit mode. To do that, you simply set the DataGridComboBoxColumn.ItemsSource collection. The absolute simplest approach is to fill it by hand, in markup. For example, this example adds a list of strings to the combo box:

```
<DataGridComboBoxColumn Header="Category"
 SelectedItemBinding="{Binding Path=CategoryName}">
  <DataGridComboBoxColumn.ItemsSource>
    <col:ArrayList>
      <sys:String>General</sys:String>
      <sys:String>Communications</sys:String>
      <sys:String>Deception</sys:String>
      <sys:String>Munitions</sys:String>
      <sys:String>Protection</sys:String>
      <sys:String>Tools</sys:String>
      <sys:String>Travel</sys:String>
    </col:ArrayList>
  </DataGridComboBoxColumn.ItemsSource>
</DataGridComboBoxColumn>
```

In order for this markup to work as written, you must map the sys and col prefixes to the appropriate .NET namespaces:

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

This works perfectly well, but it's not the best design, as it embeds data details deep into your user interface markup. Fortunately, you have several other options:

- Pull the data collection out of a resource. It's up to you whether you want to define the collection using markup (as in the previous example) or generate it in code (as in the following example).
- Pull the ItemsSource collection out of a static method, using the Static markup extension. But for solid code design, limit yourself to calling a method in your window class, not one in a data class.
- Pull the data collection out of an ObjectProvider resource, which can then call a data access class.
- Set the DataGridComboBox.Column property directly in code.

In many situations, the values you display in the list aren't the values you want to store in the data object. One common case is when dealing with related data (for example, orders that link to products, billing records that link to customers, and so on).

The StoreDB example includes one such relationship, between products and categories. In the back-end database, each product is linked to a specific category using the CategoryID field. This fact is hidden in the simplified data model that all the examples have used so far, which gives the Product class a CategoryName property (rather than a CategoryID property). The advantage of this approach is convenience, as it keeps the salient information—the category name for each product—close at hand.

The disadvantage is that the CategoryName property isn't really editable, and there's no straightforward way to change a product from one category into another.

The following example considers a more realistic case, where each Product includes a CategoryID property. On its own, the CategoryID number doesn't mean much to the application user. To display the category name instead, you need to rely on one of several possible techniques: you can add an additional CategoryName property to the Product class (which works, but is a bit clumsy), you can use a data converter in your CategoryID bindings (which could look up the matching category name in a cached list), or you can display the CategoryID column with the DataGridComboBoxColumn (which is the approach demonstrated next).

Using this approach, instead of a list of simple strings, you bind an entire list of Category objects to the DataGridComboBoxColumn.ItemsSource property:

```
categoryColumn.ItemsSource = App.StoreDb.GetCategories();
gridProducts.ItemsSource = App.StoreDb.GetProducts();
```

You then configure the DataGridComboBoxColumn. You must set three properties:

```
<DataGridComboBoxColumn Header="Category" x:Name="categoryColumn"
 DisplayMemberPath="CategoryName" SelectedValuePath="CategoryID"
 SelectedValueBinding="{Binding Path=CategoryID}"></DataGridComboBoxColumn>
```

DisplayMemberPath tells the column which text to extract from the Category object and display in the list. SelectedValuePath tells the column what *data* to extract from the Category object. SelectedValueBinding specifies the linked field in the Product object.

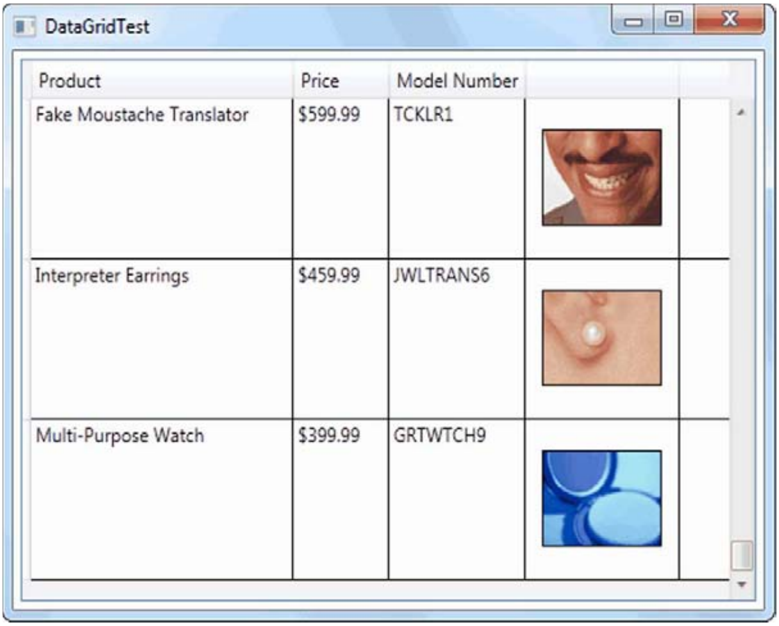
The DataGridTemplateColumn

The DataGridTemplateColumn uses a data template, which works in the same way as the data-template features you explored with list controls earlier. The only different in the DataGridTemplateColumn is that it allows you to define two templates: one for data display (the CellTemplate) and one for data editing (the CellEditingTemplate), which you'll consider shortly. Here's an example that uses the template data column to place a thumbnail image of each product in the grid (see Figure 22-9):

```
<DataGridTemplateColumn>
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <Image Stretch="None" Source=
       "{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
      </Image>
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```

This example assumes you've added the ImagePathConverter value converter to the UserControl.Resources collection:

```
<UserControl.Resources>
  <local:ImagePathConverter x:Key="ImagePathConverter"></local:ImagePathConverter>
</UserControl.Resources>
```



Product	Price	Model Number	
Fake Moustache Translator	$599.99	TCKLR1	
Interpreter Earrings	$459.99	JWLTRANS6	
Multi-Purpose Watch	$399.99	GRTWTCH9	

Figure 22-9. *A DataGrid with image content*

Formatting and Styling Columns

You can format a DataGridTextColumn in the same way that you format a TextBlock element, by setting the Foreground, FontFamily, FontSize, FontStyle, and FontWeight properties. However, the DataGridTextColumn doesn't expose all the properties of the TextBlock. For example, there's no way to set the often-used Wrapping property if you want to create a column that shows multiple lines of text. In this case, you need to use the ElementStyle property instead.

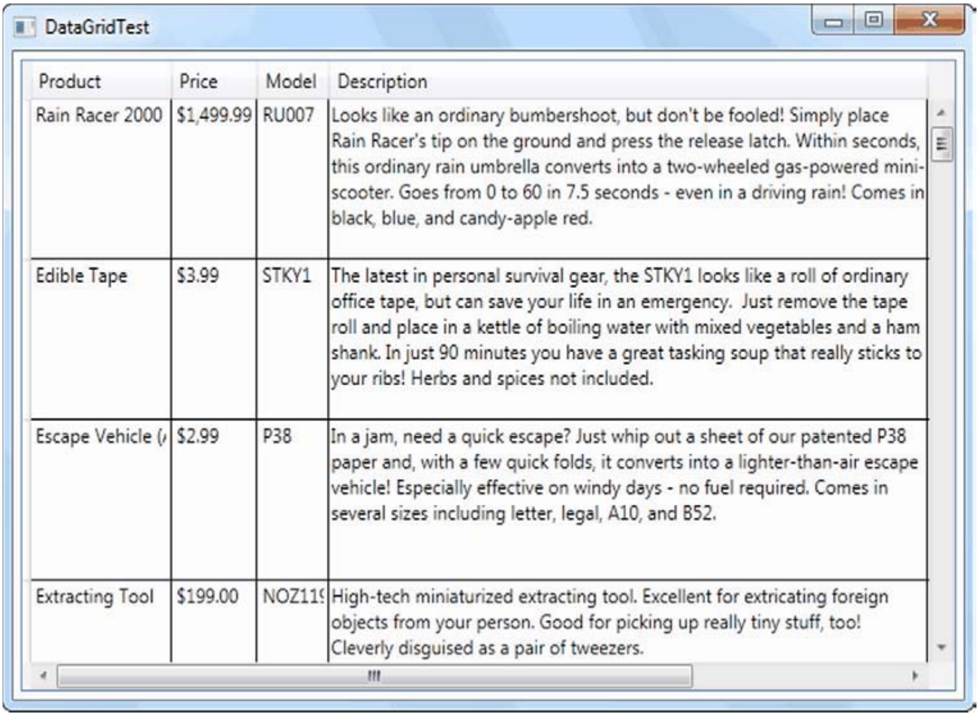
Essentially, the ElementStyle property lets you create a style that is applied to the element inside the DataGrid cell. In the case of a simple DataGridTextColumn, that's a TextBlock. In a DataGridCheckBoxColumn, it's a check box. In a DataGridTemplateColumn, it's whatever element you've created in the data template.

Here's a simple style that allows the text in a column to wrap:

```
<DataGridTextColumn Header="Description" Width="400"
 Binding="{Binding Path=Description}">
  <DataGridTextColumn.ElementStyle>
    <Style TargetType="TextBlock">
      <Setter Property="TextWrapping" Value="Wrap"></Setter>
    </Style>
  </DataGridTextColumn.ElementStyle>
</DataGridTextColumn>
```

To see the wrapped text, you must expand the row height. Unfortunately, the DataGrid can't size itself as flexibly as WPF layout containers can. Instead, you're forced to set a fixed row height using the

DataGrid.RowHeight property. This height applies to all rows, regardless of the amount of content they contain. Figure 22-10 shows an example with the row height set to 70 units.



Product	Price	Model	Description
Rain Racer 2000	$1,499.99	RU007	Looks like an ordinary bumbershoot, but don't be fooled! Simply place Rain Racer's tip on the ground and press the release latch. Within seconds, this ordinary rain umbrella converts into a two-wheeled gas-powered mini-scooter. Goes from 0 to 60 in 7.5 seconds - even in a driving rain! Comes in black, blue, and candy-apple red.
Edible Tape	$3.99	STKY1	The latest in personal survival gear, the STKY1 looks like a roll of ordinary office tape, but can save your life in an emergency. Just remove the tape roll and place in a kettle of boiling water with mixed vegetables and a ham shank. In just 90 minutes you have a great tasking soup that really sticks to your ribs! Herbs and spices not included.
Escape Vehicle (	$2.99	P38	In a jam, need a quick escape? Just whip out a sheet of our patented P38 paper and, with a few quick folds, it converts into a lighter-than-air escape vehicle! Especially effective on windy days - no fuel required. Comes in several sizes including letter, legal, A10, and B52.
Extracting Tool	$199.00	NOZ11	High-tech miniaturized extracting tool. Excellent for extricating foreign objects from your person. Good for picking up really tiny stuff, too! Cleverly disguised as a pair of tweezers.

Figure 22-10. A DataGrid with wrapped text

■ **Tip** If you want to apply the same style to multiple columns (for example, to deal with wrappable text in several places), you can define the style in the Resources collection and then refer to it in each column using a StaticResource.

You can use EditingElementStyle to style the element that's employed when you're editing a column. In the case of DataGridTextColumn, the editing element is the TextBox control.

The ElementStyle, ElementEditingStyle, and column properties give you a way to format all the cells in a specific column. However, in some cases, you might want to apply formatting settings to every cell in every column. The simplest way to do so is to configure a style for the DataGrid.RowStyle property. The DataGrid also exposes a small set of additional properties that allow you to format other parts of the grid, like the column headers and row headers. Table 22-2 has the full story.

Table 22-2. *Style-Based DataGrid Properties*

Property	Style Applies To...
ColumnHeaderStyle	The TextBlock that's used for the column headers at the top of the grid
RowHeaderStyle	The TextBlock that's used for the row headers
DragIndicatorStyle	The TextBlock that's used for a column header when the user is dragging it to a new position
RowStyle	The TextBlock that's used for ordinary rows (rows in columns that haven't been expressly customized through the ElementStyle property of the column)

Formatting Rows

By setting the properties of the DataGrid column objects, you can control how entire columns are formatted. But in many cases, it's more useful to flag rows that contain specific data. For example, you may want to draw attention to high-priced products or expired shipments. You can apply this sort of formatting programmatically by handling the DataGrid.LoadingRow event.

The LoadingRow event is a powerful tool for row formatting. It gives you access to the data object for the current row, allowing you to perform simple range checks, comparison, and more complex manipulations. It also provides the DataGridRow object for the row, letting you format the row with different colors or a different font. However, you can't format just a single cell in that row—for that, you need DataGridTemplateColumn and a custom value converter.

The LoadingRow event fires once for each row when it appears on screen. The advantage of this approach is that your application is never forced to format the whole grid; instead, LoadingRow fires only for the rows that are currently visible. But there's also a downside. As the user scrolls through the grid, the LoadingRow event is triggered continuously. As a result, you can't place time-consuming code in the LoadingRow method unless you want scrolling to grind to a halt.

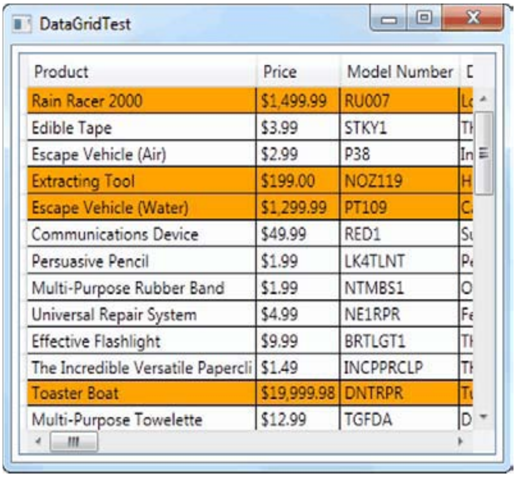
There's also another consideration: item container recycling. To lower its memory overhead, the DataGrid reuses the same DataGridRow objects to show new data as you scroll through the data. (That's why the event is called LoadingRow rather than CreatingRow.) If you're not careful, the DataGrid can load data into an already-formatted DataGridRow. To prevent this from happening, you must explicitly restore each row to its initial state.

In the following example, high-priced items are given a bright orange background (see Figure 22-11). Regular-price items are given the standard white background:

```
// Reuse brush objects for efficiency in large data displays.
private SolidColorBrush highlightBrush = new SolidColorBrush(Colors.Orange);
private SolidColorBrush normalBrush = new SolidColorBrush(Colors.White);

private void gridProducts_LoadingRow(object sender, DataGridRowEventArgs e)
{
    // Check the data object for this row.
    Product product = (Product)e.Row.DataContext;
```

```
        // Apply the conditional formatting.
        if (product.UnitCost > 100)
        {
            e.Row.Background = highlightBrush;
        }
        else
        {
            // Restore the default white background. This ensures that used,
            // formatted DataGrid objects are reset to their original appearance.
            e.Row.Background = normalBrush;
        }
    }
```



Product	Price	Model Number	D
Rain Racer 2000	$1,499.99	RU007	Lc
Edible Tape	$3.99	STKY1	Th
Escape Vehicle (Air)	$2.99	P38	In
Extracting Tool	$199.00	NOZ119	H
Escape Vehicle (Water)	$1,299.99	PT109	C
Communications Device	$49.99	RED1	Su
Persuasive Pencil	$1.99	LK4TLNT	Pe
Multi-Purpose Rubber Band	$1.99	NTMBS1	O
Universal Repair System	$4.99	NE1RPR	Fe
Effective Flashlight	$9.99	BRTLGT1	Th
The Incredible Versatile Papercli	$1.49	INCPPRCLP	Th
Toaster Boat	$19,999.98	DNTRPR	Tc
Multi-Purpose Towelette	$12.99	TGFDA	D

Figure 22-11. *Highlighting rows*

Remember, you have another option for performing value-based formatting: you can use a value converter that examines bound data and converts it to something else. This technique is especially powerful when combined with a DataGridTemplateColumn. For example, you can create a template-based column that contains a TextBlock, and bind the TextBlock.Background property to a value converter that sets the color based on the price. Unlike the LoadingRow approach shown previously, this technique allows you to format just the cell that contains the price, rather than the whole row. For more information about this technique, refer to Chapter 20.

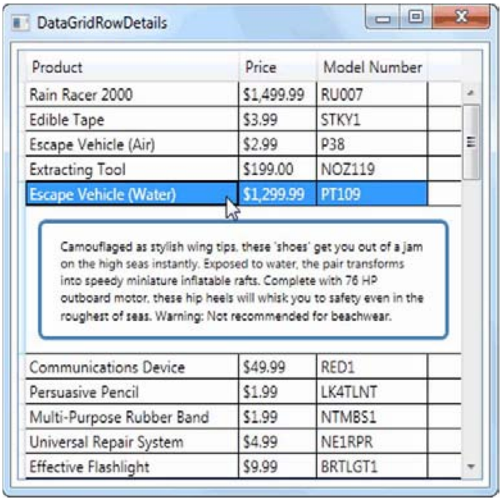
■ **Note** The formatting you apply in the LoadingRow event handler applies only when the row is loaded. If you edit a row, this LoadingRow code doesn't fire (at least, not until you scroll the row out of view and then back into sight).

Row Details

The DataGrid also supports *row details*—an optional, separate display area that appears just under the column values for a row. The row-details area adds two things that you can't get from columns alone:

- It spans the full width of the DataGrid and isn't carved into separate columns, which gives you more space to work with.
- You can configure the row-details area so that it appears only for the selected row, allowing you to tuck the extra details out of the way when they're not needed.

Figure 22-12 shows a DataGrid that uses both of these behaviors. The row-details area displays the wrapped product description text, and it's shown only for the currently selected product.



Product	Price	Model Number
Rain Racer 2000	$1,499.99	RU007
Edible Tape	$3.99	STKY1
Escape Vehicle (Air)	$2.99	P38
Extracting Tool	$199.00	NOZ119
Escape Vehicle (Water)	$1,299.99	PT109

Camouflaged as stylish wing tips, these 'shoes' get you out of a jam on the high seas instantly. Exposed to water, the pair transforms into speedy miniature inflatable rafts. Complete with 76 HP outboard motor, these hip heels will whisk you to safety even in the roughest of seas. Warning: Not recommended for beachwear.

Product	Price	Model Number
Communications Device	$49.99	RED1
Persuasive Pencil	$1.99	LK4TLNT
Multi-Purpose Rubber Band	$1.99	NTMBS1
Universal Repair System	$4.99	NE1RPR
Effective Flashlight	$9.99	BRTLGT1

Figure 22-12. *Using the row-details area*

To create this example, you need to first define the content that's shown in the row-details area by setting the DataGrid.RowDetailsTemplate property. In this case, the row-details area uses a basic template that includes a TextBlock that shows the full product text and adds a border around it:

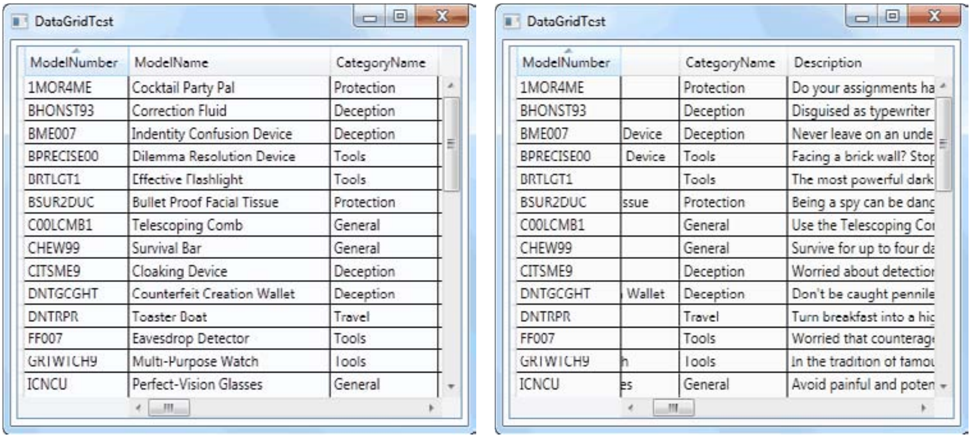
```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <Border Margin="10" Padding="10" BorderBrush="SteelBlue" BorderThickness="3"
     CornerRadius="5">
      <TextBlock Text="{Binding Path=Description}" TextWrapping="Wrap"
       FontSize="10">
      </TextBlock>
    </Border>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

Other options include adding controls that allow you to perform various tasks (for example, getting more information about a product, adding it to a shopping list, editing it, and so on).

You can configure the display behavior of the row-details area by setting the DataGrid.RowDetailsVisibilityMode property. By default, this property is set to VisibleWhenSelected, which means the row-details area is shown when the row is selected. Alternatively, you can set it to Visible, which means the details area of every row will be shown at once. Or, you can use Collapsed, which means the details area won't be shown for any row—at least, not until you change the RowDetailsVisibilityMode in code (for example, when the user selects a certain type of row).

Freezing Columns

A *frozen* column stays in place at the left size of the DataGrid, even as you scroll to the right. Figure 22-13 shows how a frozen Product column remains visible during scrolling. Notice how the horizontal scroll bar extends under only the scrollable columns, not the frozen columns.



ModelNumber	ModelName	CategoryName
1MOR4ME	Cocktail Party Pal	Protection
BHONST93	Correction Fluid	Deception
BME007	Indentity Confusion Device	Deception
BPRECISE00	Dilemma Resolution Device	Tools
BRTLGT1	Effective Flashlight	Tools
BSUR2DUC	Bullet Proof Facial Tissue	Protection
C00LCMB1	Telescoping Comb	General
CHEW99	Survival Bar	General
CITSME9	Cloaking Device	Deception
DNTGCGHT	Counterfeit Creation Wallet	Deception
DNTRPR	Toaster Boat	Travel
FF007	Eavesdrop Detector	Tools
GRIWICH9	Multi-Purpose Watch	Tools
ICNCU	Perfect-Vision Glasses	General

ModelNumber		CategoryName	Description
1MOR4ME		Protection	Do your assignments ha
BHONST93		Deception	Disguised as typewriter
BME007	Device	Deception	Never leave on an unde
BPRECISE00	Device	Tools	Facing a brick wall? Stop
BRTLGT1		Tools	The most powerful dark
BSUR2DUC	ssue	Protection	Being a spy can be danc
C00LCMB1		General	Use the Telescoping Co
CHEW99		General	Survive for up to four da
CITSME9		Deception	Worried about detectio
DNTGCGHT	Wallet	Deception	Don't be caught pennile
DNTRPR		Travel	Turn breakfast into a hic
FF007		Tools	Worried that counterag
GRIWICH9	h	Tools	In the tradition of famou
ICNCU	es	General	Avoid painful and poten

Figure 22-13. *Freezing the Product column*

Column freezing is a useful feature for very wide grids, especially when you want to make sure certain information (like the product name or a unique identifier) is always visible. To use it, you set the DataGrid.FrozenColumnCount property to a number greater than 0. For example, a value of 1 freezes just the first column:

```
<DataGrid x:Name="gridProducts" Margin="5" AutoGenerateColumns="False"
 FrozenColumnCount="1">
```

Frozen columns must always be on the left side of the grid. If you freeze one column, it is the leftmost column; if you freeze two columns, they will be the first two on the left; and so on.

Selection

Like an ordinary list control, the DataGrid lets the user select individual items. You can react to the SelectionChanged event when this happens. To find out which data object is currently selected, you can use the SelectedItem property. If you want the user to be able to select multiple rows, set the SelectionMode property to Extended. (Single is the only other option and the default.) To select multiple rows, the user must hold down the Shift or Ctrl key. You can retrieve the collection of selected items from the SelectedItems property.

■ **Tip** You can set the selection programmatically using the SelectedItem property. If you're setting the selection to an item that's not currently in view, it's a good idea to follow up with a call to the DataGrid.ScrollIntoView() method, which forces the DataGrid to scroll forward or backward until the item you've indicated is visible.

Sorting

The DataGrid features built-in sorting as long as you're binding a collection that implements IList (such as the List<T> and ObservableCollection<T> collections). If you meet this requirement, your DataGrid gets basic sorting for free.

To use the sorting, the user needs to click a column header. Clicking once sorts the column in ascending order based on its data type (for example, numbers are sorted from 0 up, and letters are sorted alphabetically). Click the column again, and the sort order is reversed. An arrow appears at the far-right side of the column header, indicating that the DataGrid is sorted based on the values in this column. The arrow points up for an ascending sort and down for a descending sort.

Users can sort based on multiple columns by holding down Shift while they click. For example, if you hold down Shift and click the Category column followed by the Price column, products are sorted into alphabetical category groups, and the items in each category group are ordered by price.

Ordinarily, the DataGrid sorting algorithm uses the bound data that appears in the column, which makes sense. However, you can choose a different property from the bound data object by setting a column's SortMemberPath. And if you have a DataGridTemplateColumn, you need to use SortMemberPath, because there's no Binding property to provide the bound data. If you don't, your column won't support sorting.

You can also disable sorting by setting the CanUserSortColumns property to false (or turn it off for specific columns by setting the column's CanUserSort property).

DataGrid Editing

One of the DataGrid's greatest conveniences is its support for editing. A DataGrid cell switches into edit mode when the user double-clicks it. But the DataGrid lets you restrict this editing ability in several ways:

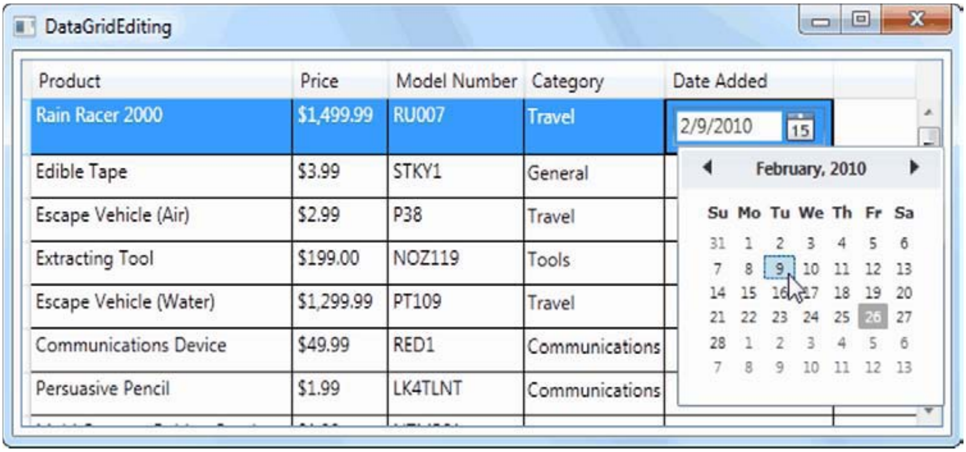
- **DataGrid.IsReadOnly.** When this property is true, users can't edit anything.
- **DataGridColumn.IsReadOnly.** When this property is true, users can't edit any of the values in that column.

- **Read-only properties.** If your data object has a property with no property setter, the DataGrid is intelligent enough to notice this detail and disable column editing, just as if you had set DataGridColumn.IsReadOnly to true. Similarly, if your property isn't a simple text, numeric, or date type, the DataGrid makes it read-only (although you can remedy this situation by switching to the DataGridTemplateColumn, as described shortly).

What happens when a cell switches into edit mode depends on the column type. A DataGridTextColumn shows a text box (although it's a seamless-looking text box that fills the entire cell and has no visible border). A DataGridCheckBox column shows a check box that you can check or uncheck. But the DataGridTemplateColumn is by far the most interesting. It allows you to replace the standard editing text box with a more specialized input control.

For example, the following column shows a date. When the user double-clicks to edit that value, it turns into a drop-down DatePicker (see Figure 22-14) with the current value preselected:

```
<DataGridTemplateColumn Header="Date Added">
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Margin="4" Text=
 "{Binding Path=DateAdded, Converter={StaticResource DateOnlyConverter}}">
      </TextBlock>
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
  <DataGridTemplateColumn.CellEditingTemplate>
    <DataTemplate>
      <DatePicker SelectedDate="{Binding Path=DateAdded, Mode=TwoWay}">
      </DatePicker>
    </DataTemplate>
  </DataGridTemplateColumn.CellEditingTemplate>
</DataGridTemplateColumn>
```



Product	Price	Model Number	Category	Date Added
Rain Racer 2000	$1,499.99	RU007	Travel	2/9/2010
Edible Tape	$3.99	STKY1	General	
Escape Vehicle (Air)	$2.99	P38	Travel	
Extracting Tool	$199.00	NOZ119	Tools	
Escape Vehicle (Water)	$1,299.99	PT109	Travel	
Communications Device	$49.99	RED1	Communications	
Persuasive Pencil	$1.99	LK4TLNT	Communications	

Su	Mo	Tu	We	Th	Fr	Sa
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	1	2	3	4	5	6
7	8	9	10	11	12	13

Figure 22-14. *Editing dates with the DatePicker*

The DataGrid automatically supports the same basic validation system you learned about in the previous chapter, which reacts to problems in the data binding system (such as the inability to convert supplied text to the appropriate data type) or exceptions thrown by the property setter. Here's an example that uses a custom validation rule to validate the UnitCost field:

```
<DataGridTextColumn Header="Price">
  <DataGridTextColumn.Binding>
    <Binding Path="UnitCost" StringFormat="{}{0:C}">
      <Binding.ValidationRules>
        <local:PositivePriceRule Max="999.99" />
      </Binding.ValidationRules>
    </Binding>
  </DataGridTextColumn.Binding>
</DataGridTextColumn>
```

The default ErrorTemplate for the DataGridCell displays a red outline around the invalid value, much the same as other input controls like the TextBox.

You can implement validation a couple of other ways with a DataGrid. One option is to use the DataGrid's editing events, which are listed in Table 22-3. The order of rows matches the order that the events fire in the DataGrid.

Table 22-3. DataGrid Editing Events

Name	Description
BeginningEdit	Occurs when the cell is about to be put in edit mode. You can examine the column and row that are currently being edited, check the cell value, and cancel this operation using the DataGridBeginningEditEventArgs.Cancel property.
PreparingCellForEdit	Used for template columns. At this point, you can perform any last-minute initialization that's required for the editing controls. Use DataGridPreparingCellForEditEventArgs.EditingElement to access the element in the CellEditingTemplate.
CellEditEnding	Occurs when the cell is about to exit edit mode. DataGridCellEditEndingEventArgs.EditAction tells you whether the user is attempting to accept the edit (for example, by pressing Enter or clicking another cell) or cancel it (by pressing the Escape key). You can examine the new data and set the Cancel property to roll back an attempted change.
RowEditEnding	Occurs when the user navigates to a new row after editing the current row. As with CellEditEnding, you can use this point to perform validation and cancel the change. Typically, you'll perform validation that involves several columns—for example, ensuring that the value in one column isn't greater than the value in another.

If you need a place to perform validation logic that is specific to your page (and so can't be baked into the data objects), you can write custom validation logic that responds to the CellEditEnding and RowEditEnding events. Check column rules in the CellEditEnding event handler, and validate the consistency of the entire row in the RowEditEnding event. And remember that if you cancel an edit, you should provide an explanation of the problem (usually in a TextBlock elsewhere on the page).

The Last Word

In this chapter, you took a closer look at the ItemsControl classes provided by WPF. You learned how to use the ListView to create lists with multiple viewing modes, the TreeView to show hierarchical data, and the DataGrid to view and edit a dense assortment of data in a single place.

The most impressive aspect of all these classes is that they derive from a single base class—the ItemsControl—that defines their essential functionality. The fact that all these controls share the same content model, the same data binding ability, and the same styling and templating features is one of WPF's small miracles. Remarkably, the ItemsControl defines all the basics for any WPF list control, even those that wrap hierarchical data, like the TreeView. The only change in the model is that the children of these controls (TreeViewItem objects) are *themselves* ItemsControl objects, with the ability to host their own children.

CHAPTER 23

■ ■ ■

Windows

Windows are the basic ingredients in any desktop application—so basic that the operating system is named after them. And although WPF has a model for creating navigation applications that divide tasks into separate pages, windows are still the dominant metaphor for creating applications.

In this chapter, you'll explore the Window class. You'll learn the various ways to show and position windows, how window classes should interact, and what built-in dialog boxes WPF provides. You'll also look at more exotic window effects, such as nonrectangular windows, windows with transparency, and windows with the Aero glass effect. Finally, you'll explore WPF"s support for programming the Windows 7 taskbar.

■ **What's New** One of the disappointments in earlier versions of WPF was the lack of built-in support for new Windows Vista features. WPF 4 catches up and does one better by adding support for the Windows 7 taskbar. In the section "Programming the Windows 7 Taskbar," you'll learn how to use cutting-edge taskbar features like jump lists, progress notification, icon overlays, and taskbar previews.

The Window Class

As you learned in Chapter 6, the Window class derives from ContentControl. That means it can contain a single child (usually a layout container such as the Grid control), and you can paint the background with a brush by setting the Background property. You can also use the BorderBrush and BorderThickness properties to add a border around your window, but this border is added inside the window frame (around the edge of the client area). You can remove the window frame altogether by setting the WindowStyle property to None, which allows you to create a completely customized window, as you'll see later in the "Nonrectangular Windows" section.

■ **Note** The *client area* is the surface inside the window boundaries. This is where you place your content. The nonclient area includes the border and the title bar at the top of the window. The operating system manages this area.

In addition, the Window class adds a small set of members that will be familiar to any Windows programmer. The most obvious are the appearance-related properties that let you change the way the nonclient portion of the window appears. Table 23-1 lists these members.

Table 23-1. *Basic Properties of the Window Class*

Name	Description
AllowsTransparency	When set to true, the Window class allows other windows to show through if the background is set to a transparent color. If set to false (the default), the content behind the window never shows through, and a transparent background is rendered as a black background. This property allows you to create irregularly shaped windows when it's used in combination with a WindowStyle of None, as you'll see in the "Nonrectangular Windows" section.
Icon	An ImageSource object that identifies the icon you want to use for your window. Icons appear at the top left of a window (if it has one of the standard border styles), in the taskbar (if ShowInTaskBar is true), and in the selection window that's shown when the user presses Alt+Tab to navigate between running applications. Because these icons are different sizes, your .ico file should include at least a 16×16 pixel image and a 32×32 pixel image. In fact, the modern Windows icon standard (supported in Windows Vista and Windows 7) adds both a 48×48 pixel image and a 256×256 image, which can be sized as needed for other purposes. If Icon is a null reference, the window is given the same icon as the application (which you can set in Visual Studio by double-clicking the Properties node in the Solution Explorer and then choosing the Application tab). If this is omitted, WPF will use a standard but unremarkable icon that shows a window.
Top and Left	Set the distance between the top-left corner of the window and the top and left edges of the screen, in device-independent pixels. The LocationChanged event fires when either of these details changes. If the WindowStartupPosition property is set to Manual, you can set these properties before the window appears to set its position. You can always use these properties to move the position of a window *after* it has appeared, no matter what value you use for WindowStartupPosition.
ResizeMode	Takes a value from the ResizeMode enumeration that determines whether the user can resize the window. This setting also affects the visibility of the maximize and minimize boxes. Use NoResize to lock a window's size completely, CanMinimize to allow minimizing only, CanResize to allow everything, or CanResizeWithGrip to add a visual detail at the bottom-right corner of the window to show that the window is resizable.

Name	Description
RestoreBounds	Gets the bounds of the window. However, if the window is currently maximized or minimized, this property provides the bounds that were last used before the window was maximized or minimized. This is extremely useful if you need to store the position and dimensions of a window, as described later in this chapter.
ShowInTaskbar	If set to true, the window appears in the taskbar and the Alt+Tab list. Usually, you will set this to true only for your application's main window.
SizeToContent	Allows you to create a window that enlarges itself automatically. This property takes a value from the SizeToContent enumeration. Use Manual to disable automatic sizing; or use Height, Width, or WidthAndHeight to allow the window to expand in different dimensions to accommodate dynamic content. When using SizeToContent, the window may be sized larger than the bounds of the screen.
Title	The caption that appears in the title bar for the window (and in the taskbar).
Topmost	When set to true, this window is always displayed on top of every other window in your application (unless these other windows also have TopMost set to true). This is a useful setting for palettes that need to "float" above other windows.
WindowStartupLocation	Takes a value from the WindowStartupLocation enumeration. Use Manual to position a window exactly with the Left and Top properties, CenterScreen to place the window in the center of the screen, or CenterOwner to center the window with respect to the window that launched it. When showing a modeless window with CenterOwner, make sure you set the Owner property of the new window before you show it.
WindowState	Takes a value from the WindowState enumeration. Informs you (and allows you to change) whether the window is currently maximized, minimized, or in its normal state. The StateChanged event fires when this property changes.
WindowStyle	Takes a value from the WindowStyle enumeration, which determines the border for the window. Your options include SingleBorderWindow (the default), ThreeDBorderWindow (which is rendered in a slightly different way on Windows XP), ToolWindow (a thin border good for floating tool windows, with no maximize or minimize buttons), and None (a very thin raised border with no title bar region). Figure 23-1 shows the difference.

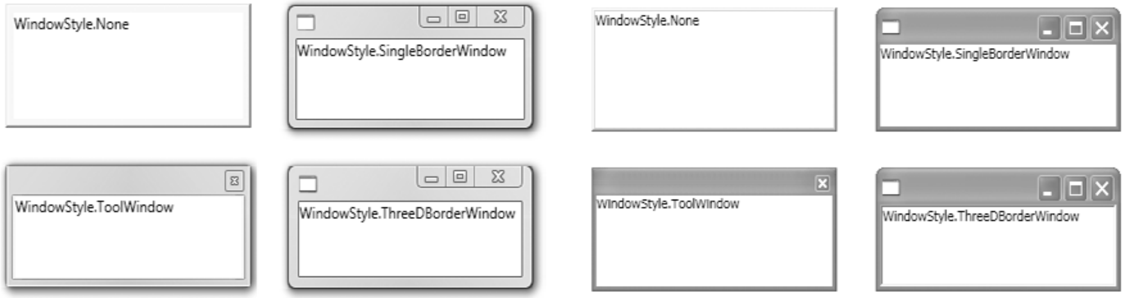

WindowStyle.None
WindowStyle.SingleBorderWindow
WindowStyle.None
WindowStyle.SingleBorderWindow
WindowStyle.ToolWindow
WindowStyle.ThreeDBorderWindow
WindowStyle.ToolWindow
WindowStyle.ThreeDBorderWindow

Figure 23-1. *Different values for WindowStyle: Windows 7/Vista (left), Windows XP (right)*

You've already learned about the lifetime events that fire when a window is created, activated, and unloaded (in Chapter 5). In addition, the Window class includes LocationChanged and WindowStateChanged events, which fire when the window's position and WindowState change, respectively.

Showing a Window

To display a window, you need to create an instance of the Window class and use the Show() or ShowDialog() method.

The ShowDialog() method shows a *modal* window. Modal windows stop the user from accessing the parent window by blocking any mouse or keyboard input to it, until the modal window is closed. In addition, the ShowDialog() method doesn't return until the modal window is closed, so any code that you've placed after the ShowDialog() call is put on hold. (However, that doesn't mean other code can't run—for example, if you have a timer running, its event handler will still run.) A common pattern in code is to show a modal window, wait until it's closed, and then act on its data.

Here's an example that uses the ShowDialog() method:

```
TaskWindow winTask = new TaskWindow();
winTask.ShowDialog();
// Execution reaches this point after winTask is closed.
```

The Show() method shows a *modeless* window, which doesn't block the user from accessing any other window. The Show() method also returns immediately after the window is shown, so subsequent code statements are executed immediately. You can create and show several modeless windows, and the user can interact with them all at once. When using modeless windows, synchronization code is sometimes required to make sure that changes in one window update the information in another window to prevent a user from working with invalid information.

Here's an example that uses the Show() method:

```
MainWindow winMain = new MainWindow();
winMain.Show();
// Execution reaches this point immediately after winMain is shown.
```

Modal windows are ideal for presenting the user with a choice that needs to be made before an operation can continue. For example, consider Microsoft Word, which shows its Options and Print windows modally, forcing you to make a decision before continuing. On the other hand, the windows used to search for text or check the spelling in a document are shown modelessly, allowing the user to edit text in the main document window while performing the task.

Closing a window is equally easy, using the Close() method. Alternatively, you can hide a window from view using Hide() or by setting the Visibility property to Hidden. Either way, the window remains open and available to your code. Generally, it makes sense to hide only modeless windows. That's because if you hide a modal window, your code remains stalled until the window is closed, and the user can't close an invisible window.

Positioning a Window

Usually, you won't need to position a window exactly on the screen. You'll simply use CenterOwner for the WindowState and forget about the whole issue. In other, less common cases, you'll use Manual for the Windows state and set an exact position using the Left and Right properties.

Sometimes you need to take a little more care in choosing an appropriate location and size for your window. For example, you could accidentally create a window that is too large to be accommodated on a low-resolution display. If you are working with a single-window application, the best solution is to create a resizable window. If you are using an application with several floating windows, the answer is not as simple.

You could just restrict your window positions to locations that are supported on even the smallest monitors, but that's likely to frustrate higher-end users (who have purchased better monitors for the express purpose of fitting more information on their screen at a time). In this case, you usually want to make a runtime decision about the best window location. To do this, you need to retrieve some basic information about the available screen real estate using the System.Windows.SystemParameters class.

The SystemParameters class consists of a huge list of static properties that return information about various system settings. For example, you can use the SystemParameters class to determine whether the user has enabled hot tracking and the "drag full windows" option, among many others. With windows, the SystemParameters class is particularly useful because it provides two properties that give the dimensions of the current screen: FullPrimaryScreenHeight and FullPrimaryScreenWidth. Both are quite straightforward, as this bit of code (which centers the window at runtime) demonstrates:

```
double screeHeight = SystemParameters.FullPrimaryScreenHeight;
double screeWidth = SystemParameters.FullPrimaryScreenWidth;
this.Top = (screenHeight - this.Height) / 2;
this.Left = (screenWidth - this.Width) / 2;
```

Although this code is equivalent to using CenterScreen for the WindowState property of the window, it gives you the flexibility to implement different positioning logic and to run this logic at the appropriate time.

An even better choice is to use the SystemParameters.WorkArea rectangle to center the window in the *available* screen area. The work area measurement doesn't include the area where the taskbar is docked (and any other "bands" that are docked to the desktop).

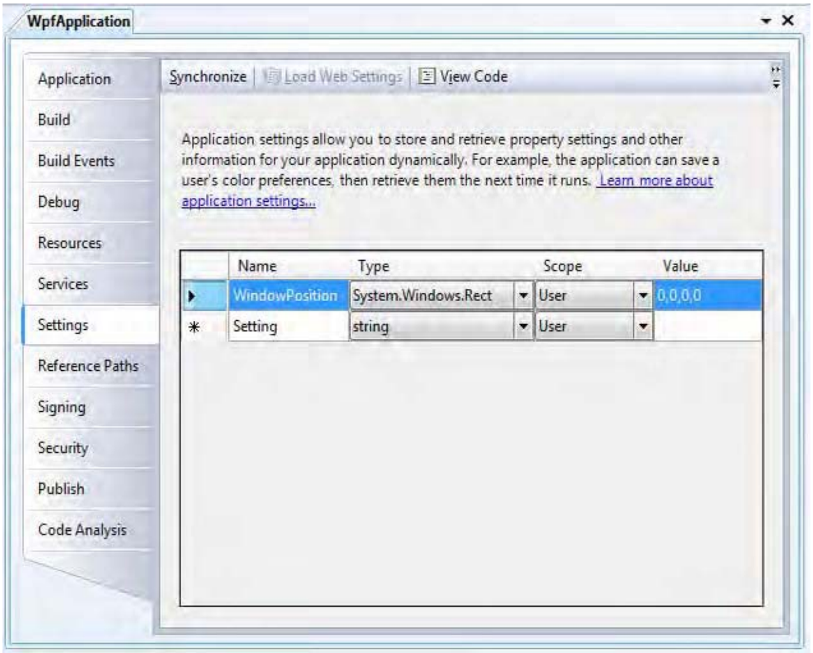
```
double workHeight = SystemParameters.WorkArea.Height;
double workWidth = SystemParameters.WorkArea.Width;
this.Top = (workHeight - this.Height) / 2;
this.Left = (workWidth - this.Width) / 2;
```

■ **Note** Both window-positioning examples have one minor drawback. When the Top property is set on a window that's already visible, the window is moved and refreshed immediately. The same process happens when the Left property is set in the following line of code. As a result, keen-eyed users may see the window move twice. Unfortunately, the Window class does not provide a method that allows you to set both position properties at once. The only solution is to position the window after you create it but before you make it visible by calling Show() or ShowDialog().

Saving and Restoring Window Location

A common requirement for a window is to remember its last location. This information can be stored in a user-specific configuration file or in the Windows registry.

If you wanted to store the position of an important window in a user-specific configuration file, you would begin by double-clicking the Properties node in the Solution Explorer and choosing the Settings section. Then, add a user-scoped setting with a data type of System.Windows.Rect, as shown in Figure 23-2.



Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

	Name	Type	Scope	Value
▶	WindowPosition	System.Windows.Rect	User	0,0,0,0
*	Setting	string	User	

Figure 23-2. *A property for storing a window's position and size*

With this setting in place, it's easy to create code that automatically stores information about a window's size and position, as shown here:

```
Properties.Settings.Default.WindowPosition = win.RestoreBounds;
Properties.Settings.Default.Save();
```

Notice that this code uses the RestoreBounds property, which gives the correct dimensions (the last nonmaximized, nonminimized size), even if the window is currently maximized or minimized. (This handy feature wasn't directly available in Windows Forms, and it necessitated the use of the GetWindowPlacement() unmanaged API function.)

It's just as easy to retrieve this information when you need it:

```
try
{
    Rect bounds = Properties.Settings.Default.WindowPosition;
    win.Top = bounds.Top;
    win.Left = bounds.Left;

    // Restore the size only for a manually sized
    // window.
    if (win.SizeToContent == SizeToContent.Manual)
    {
        win.Width = bounds.Width;
        win.Height = bounds.Height;
    }
}
catch
{
    MessageBox.Show("No settings stored.");
}
```

The only limitation to this approach is that you need to create a separate property for each window that you want to store. If you need to store the position of many different windows, you might want to design a more flexible system. For example, the following helper class stores a position for any window you pass in, using a registry key that incorporates the name of that window. (You could use additional identifying information if you want to store the settings for several windows that will have the same name.)

```
public class WindowPositionHelper
{
    public static string RegPath = @"Software\MyApp\WindowBounds\";

    public static void SaveSize(Window win)
    {
        // Create or retrieve a reference to a key where the settings
        // will be stored.
        RegistryKey key;
        key = Registry.CurrentUser.CreateSubKey(RegPath + win.Name);

        key.SetValue("Bounds", win.RestoreBounds.ToString());
        key.SetValue("Bounds",
          win.RestoreBounds.ToString(CultureInfo.InvariantCulture));
    }
```