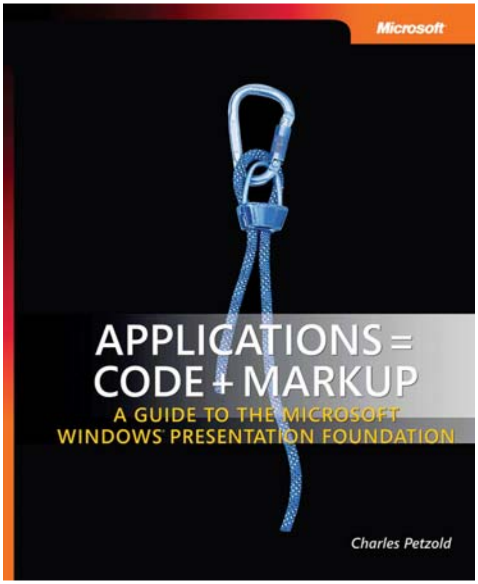

Applications = Code + Markup: A Guide to the Microsoft® Windows® Presentation Foundation

Charles Petzold

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/6476.aspx

9780735619579
Publication Date: August 2006

Table of Contents

Chapter 14

The Menu Hierarchy

The traditional focus of the user interface in a Windows application is the menu. The menu occupies prime real estate at the top of the application window, right under the caption bar and extending the full width of the window. In repose, the menu is commonly a horizontal list of text items. Clicking an item on this top-level menu generally displays a boxed list of other items, called a drop-down menu or a submenu. Each submenu contains other menu items that can either trigger commands or invoke other nested submenus.

In short, the menu is a hierarchy. Every item on the menu is an object of type *MenuItem*. The menu itself is an object of type *Menu*. To understand where these two controls fit into the other Windows Presentation Foundation controls, it is helpful to examine the following partial class hierarchy:

Control

    ContentControl

        HeaderedContentControl

    ItemsControl

        HeaderedItemsControl

These four classes that derive from *Control* encompass many familiar controls:

- Controls that derive from *ContentControl* are characterized by a property named *Content*. These controls include buttons, labels, tool tips, the scroll viewer, list box items, and the window itself.
- The *HeaderedContentControl* derives from *ContentControl* and adds a *Header* property. The group box falls under this category.
- *ItemsControl* defines a property named *Items* that is a collection of other objects. This category includes the list box and combo box.
- *HeaderedItemsControls* adds a *Header* property to the properties it inherits from *Items-Control*. A menu item is one such control.

The *Header* property of the *MenuItem* object is the visual representation of the item itself, usually a short text string that is optionally accompanied by a small bitmap. Each menu item also potentially contains a collection of items that appear in a submenu. These submenu items are collected in the *Items* property. For menu items that invoke commands directly, the *Items* collection is empty.

For example, the first item on the top-level menu is typically File. This is a *MenuItem* object. The *Header* property is the text string "File" and the *Items* collection includes the *MenuItem* objects for New, Open, Save, and so forth.

The only part of the menu that doesn't follow this pattern is the top-level menu itself. The top-level menu certainly is a collection of items (File, Edit, View, and Help, for example) but there is no header associated with this collection. For that reason, the top-level menu is an object of type *Menu*, which derives from *ItemsControl*. This partial class hierarchy shows the menu-related classes:

Control

- *ItemsControl*
  - *HeaderItemsControl*
    - *MenuItem*
  - *MenuBase* (abstract)
    - *ContextMenu*
    - *Menu*
- *Separator*

The *Separator* control simply displays a horizontal or vertical line (depending on its context) that's often used on submenus to separate menu items into functional categories.

The items in the menu can actually be objects of almost any type, but you'll generally use *MenuItem* because it defines several properties and events commonly associated with menu items. Like *ButtonBase*, *MenuItem* defines a *Click* event and a *Command* property. Your program can handle many menu items just as if they were buttons.

MenuItem also defines a property named *Icon* that lets you put a little picture in the menu item in a standard location. Interestingly, the *Icon* property is of type *Object*, which means you can easily use an element from the Shapes library (as I'll demonstrate in this chapter).

Menu items can be checked, either to denote on/off options or to indicate one item selected from a group of mutually exclusive items. *MenuItem* includes a Boolean *IsChecked* property to turn checkmarks on and off, and an *IsCheckable* property to automate the toggling of checkmarks. The *Checked* event is fired when the *IsChecked* property changes from *false* to *true*. The *Unchecked* event indicates when *IsChecked* changes from *true* to *false*.

Sometimes it's necessary for a program to disable certain items on a submenu. For example, the Save option on the File menu should be disabled if the program currently has no document to save. It is often most convenient to disable menu items when the submenu is being displayed. *MenuItem* defines a *SubmenuOpened* event to help out.

Constructing a menu generally begins at the top and proceeds downward. You first create an object of type *Menu*:

```
Menu menu = new Menu();
```

Commonly, the first item is File:

```
MenuItem itemFile = new MenuItem();
itemFile.Header = "_File";
```

As in other controls, the underline character facilitates navigation with the keyboard. When the user presses the Alt key, the *F* in "File" becomes underlined, and pressing the F key then opens the File submenu. On the top-level menu, and in each submenu, underlined letters should be unique.

You make the File item part of the top-level menu by adding it to the *Items* collection of the *Menu* object:

```
menu.Items.Add(itemFile);
```

The first item on the File menu is often New:

```
MenuItem itemNew = new MenuItem();
itemNew.Header = "_New";
itemNew.Click += NewOnClick;
itemFile.Items.Add(itemNew);
```

This New item is a command, so assign the *Click* event a handler to process that command. Add the New item to the *File* collection:

```
itemFile.Items.Add(itemNew);
```

And so forth.

Here's a program that displays a little text in its client area and constructs a menu. Only one of the File items is implemented, but the menu includes a top-level Window item containing four checkable items that let you change the properties of the window.

PeruseTheMenu.cs

```
//----------------------------------------------
// PeruseTheMenu.cs (c) 2006 by Charles Petzold
//----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.PeruseTheMenu
{
    public class PeruseTheMenu : Window
```

```
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new PeruseTheMenu());
        }
        public PeruseTheMenu()
        {
            Title = "Peruse the Menu";

            // Create DockPanel.
            DockPanel dock = new DockPanel();
            Content = dock;

            // Create Menu docked at top.
            Menu menu = new Menu();
            dock.Children.Add(menu);
            DockPanel.SetDock(menu, Dock.Top);

            // Create TextBlock filling the rest.
            TextBlock text = new TextBlock();
            text.Text = Title;
            text.FontSize = 32;     // ie, 24 points.
            text.TextAlignment = TextAlignment.Center;
            dock.Children.Add(text);

            // Create File menu.
            MenuItem itemFile = new MenuItem();
            itemFile.Header = "_File";
            menu.Items.Add(itemFile);

            MenuItem itemNew = new MenuItem();
            itemNew.Header = "_New";
            itemNew.Click += UnimplementedOnClick;
            itemFile.Items.Add(itemNew);

            MenuItem itemOpen = new MenuItem();
            itemOpen.Header = "_Open";
            itemOpen.Click += UnimplementedOnClick;
            itemFile.Items.Add(itemOpen);

            MenuItem itemSave = new MenuItem();
            itemSave.Header = "_Save";
            itemSave.Click += UnimplementedOnClick;
            itemFile.Items.Add(itemSave);

            itemFile.Items.Add(new Separator());

            MenuItem itemExit = new MenuItem();
            itemExit.Header = "E_xit";
            itemExit.Click += ExitOnClick;
            itemFile.Items.Add(itemExit);
```

```
        // Create Window menu.
        MenuItem itemWindow = new MenuItem();
        itemWindow.Header = "_Window";
        menu.Items.Add(itemWindow);

        MenuItem itemTaskbar = new MenuItem();
        itemTaskbar.Header = "_Show in Taskbar";
        itemTaskbar.IsCheckable = true;
        itemTaskbar.IsChecked = ShowInTaskbar;
        itemTaskbar.Click += TaskbarOnClick;
        itemWindow.Items.Add(itemTaskbar);

        MenuItem itemSize = new MenuItem();
        itemSize.Header = "Size to _Content";
        itemSize.IsCheckable = true;
        itemSize.IsChecked = SizeToContent == SizeToContent.WidthAndHeight;
        itemSize.Checked += SizeOnCheck;
        itemSize.Unchecked += SizeOnCheck;
        itemWindow.Items.Add(itemSize);

        MenuItem itemResize = new MenuItem();
        itemResize.Header = "_Resizable";
        itemResize.IsCheckable = true;
        itemResize.IsChecked = ResizeMode == ResizeMode.CanResize;
        itemResize.Click += ResizeOnClick;
        itemWindow.Items.Add(itemResize);

        MenuItem itemTopmost = new MenuItem();
        itemTopmost.Header = "_Topmost";
        itemTopmost.IsCheckable = true;
        itemTopmost.IsChecked = Topmost;
        itemTopmost.Checked += TopmostOnCheck;
        itemTopmost.Unchecked += TopmostOnCheck;
        itemWindow.Items.Add(itemTopmost);
    }
    void UnimplementedOnClick(object sender, RoutedEventArgs args)
    {
        MenuItem item = sender as MenuItem;
        string strItem = item.Header.ToString().Replace("_", "");
        MessageBox.Show("The " + strItem +
                        " option has not yet been implemented", Title);
    }
    void ExitOnClick(object sender, RoutedEventArgs args)
    {
        Close();
    }
    void TaskbarOnClick(object sender, RoutedEventArgs args)
    {
        MenuItem item = sender as MenuItem;
        ShowInTaskbar = item.IsChecked;
    }
    void SizeOnCheck(object sender, RoutedEventArgs args)
```

```
        {
            MenuItem item = sender as MenuItem;
            SizeToContent = item.IsChecked ? SizeToContent.WidthAndHeight :
                                             SizeToContent.Manual;
        }
        void ResizeOnClick(object sender, RoutedEventArgs args)
        {
            MenuItem item = sender as MenuItem;
            ResizeMode = item.IsChecked ? ResizeMode.CanResize :
                                          ResizeMode.NoResize;
        }
        void TopmostOnCheck(object sender, RoutedEventArgs args)
        {
            MenuItem item = sender as MenuItem;
            Topmost = item.IsChecked;
        }
    }
}
```

Notice that the constructor begins by creating a *DockPanel*. Customarily the menu is docked on the top of the client area, and unless you want to baffle users, your menus will appear there as well. The *DockPanel* is the standard base panel for windows that have menus, toolbars, or status bars. You can get the same effect with a *StackPanel* or *Grid*, but the *DockPanel* is standard.

The *UnimplementedOnClick* method handles the *Click* events for the New, Open, and Save items. The *Click* handler for the Exit item calls *Close* on the *Window* object to end the program. A *Separator* separates the Exit item from the others, as is common.

The four items on the Window menu all have their *IsCheckable* property set to *true* to enable automatic toggling of the checkmark. The *IsChecked* property indicates whether a checkmark currently appears or not. For the first and third items, the program installs a handler for the *Click* event. For the second and fourth items, the program installs the same handler for the *Checked* and *Unchecked* events. It doesn't really matter which approach you use. Potentially installing separate *Checked* and *Unchecked* handlers lets you perform actions without explicitly examining the *IsChecked* property. In this program, the *Click*, *Checked*, and *Unchecked* event handlers all merely use the *IsChecked* property to set certain properties of the window.

Try commenting out the following two lines of code:

```
itemTaskbar.IsChecked = ShowInTaskbar;
itemTaskbar.Click += TaskbarOnClick;
```

And replace them with these:

```
itemTaskbar.SetBinding(MenuItem.IsCheckedProperty, "ShowInTaskbar");
itemTaskbar.DataContext = this;
```

You can also get rid of the entire *TaskbarOnClick* method, and the program will work the same. This is a little taste of data binding. Basically you're telling the *MenuItem* that the *IsChecked* property should always be the same value as the *ShowInTaskbar* property of the *this* object (the window). You can set such a data binding with the *Topmost* menu item as well, but you can't do it with the other two because the properties aren't Booleans.

If you want a menu item to have a checkmark, and you don't set the *IsCheckable* property to *true*, you'll need to handle *Click* events and manually check and uncheck the item using the *IsChecked* property. This approach is necessary when you use the menu to display a group of mutually exclusive items, where checking any item causes the previously checked item to become unchecked (much like a group of radio buttons). With a group of mutually exclusive checked items, you'll leave the *IsCheckable* property in its default *false* setting so that there's no automatic toggling, and you'll probably set the *IsChecked* property on one of the items when initially creating the items. You'll also handle all the checking and unchecking logic yourself in the *Click* event handlers. Often it's easiest to share a single *Click* handler for all the items in a mutually exclusive group.

To keep track of the currently checked item in any mutually exclusive group, you'll probably maintain a field of type *MenuItem* named (for example) *itemChecked*:

```
MenuItem itemChecked;
```

You initialize this field in the window's constructor. If you use a single *Click* event handler for the whole group of mutually exclusive items, the *Click* event handler begins by unchecking the currently checked item:

```
itemChecked.IsChecked = false;
```

The handler then saves the item being clicked as the new value of *itemChecked*, and checks that item:

```
itemChecked = args.Source as MenuItem;
itemChecked.IsChecked = true;
```

The event handler can then do whatever specifically needs to be done for these items. Here's a program similar to the TuneTheRadio program in Chapter 5 that lets you change the *WindowStyle* property of the window:

CheckTheWindowStyle.cs

```
//-----------------------------------------------------
// CheckTheWindowStyle.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
```

```
namespace Petzold.CheckTheWindowStyle
{
    public class CheckTheWindowStyle : Window
    {
        MenuItem itemChecked;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CheckTheWindowStyle());
        }
        public CheckTheWindowStyle()
        {
            Title = "Check the Window Style";

            // Create DockPanel.
            DockPanel dock = new DockPanel();
            Content = dock;

            // Create Menu docked at top.
            Menu menu = new Menu();
            dock.Children.Add(menu);
            DockPanel.SetDock(menu, Dock.Top);

            // Create TextBlock filling the rest.
            TextBlock text = new TextBlock();
            text.Text = Title;
            text.FontSize = 32;
            text.TextAlignment = TextAlignment.Center;
            dock.Children.Add(text);

            // Create MenuItem objects to change WindowStyle.
            MenuItem itemStyle = new MenuItem();
            itemStyle.Header = "_Style";
            menu.Items.Add(itemStyle);

            itemStyle.Items.Add(
                CreateMenuItem("_No border or caption", WindowStyle.None));

            itemStyle.Items.Add(
                CreateMenuItem("_Single-border window",
                               WindowStyle.SingleBorderWindow));

            itemStyle.Items.Add(
                CreateMenuItem("3_D-border window",
                               WindowStyle.ThreeDBorderWindow));
            itemStyle.Items.Add(
                CreateMenuItem("_Tool window",
                               WindowStyle.ToolWindow));
        }
        MenuItem CreateMenuItem(string str, WindowStyle style)
        {
            MenuItem item = new MenuItem();
            item.Header = str;
            item.Tag = style;
```

```
            item.IsChecked = (style == WindowStyle);
            item.Click += StyleOnClick;

            if (item.IsChecked)
                itemChecked = item;

            return item;
        }
        void StyleOnClick(object sender, RoutedEventArgs args)
        {
            itemChecked.IsChecked = false;
            itemChecked = args.Source as MenuItem;
            itemChecked.IsChecked = true;

            WindowStyle = (WindowStyle)itemChecked.Tag;
        }
    }
}
```

Because the four items that appear on the Style menu are rather similar and share the same *Click* event handler, the program defines a little method named *CreateMenuItem* specifically to create these items. Each item has a text string describing a particular member of the *WindowStyle* enumeration. The ever-handy *Tag* property of the *MenuItem* object gets the enumeration member itself. If the particular *WindowStyle* enumeration member is the same as the window's *WindowStyle* property, the method sets the *IsChecked* property to *true* and also sets the *itemChecked* field to that item.

The *Click* event handler unchecks the *itemChecked* item, sets *itemChecked* to the clicked item, and concludes by setting the *WindowStyle* property of the window based on the *Tag* property of the clicked item.

If you think about it, the CheckTheWindowStyle program doesn't really need to maintain the *itemChecked* field. That's simply for convenience. The program can always determine which item is currently checked by searching through the *Items* collection of the Style item, or even by examining the *WindowStyle* property of the window.

It's not even necessary to check and uncheck the items in the *Click* event handler. Instead, the program can prepare the submenu for viewing during the *SubmenuOpened* event and check the correct item at that time. The following program demonstrates an alternative approach to checking and unchecking menu items. Two menu items let you change the foreground and background brushes of a *TextBlock* element.

CheckTheColor.cs

```
//-----------------------------------------------
// CheckTheColor.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Reflection;
using System.Windows;
```

```
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Petzold.CheckTheColor
{
    public class CheckTheColor : Window
    {
        TextBlock text;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CheckTheColor());
        }
        public CheckTheColor()
        {
            Title = "Check the Color";

            // Create DockPanel.
            DockPanel dock = new DockPanel();
            Content = dock;

            // Create Menu docked at top.
            Menu menu = new Menu();
            dock.Children.Add(menu);
            DockPanel.SetDock(menu, Dock.Top);

            // Create TextBlock filling the rest.
            text = new TextBlock();
            text.Text = Title;
            text.TextAlignment = TextAlignment.Center;
            text.FontSize = 32;
            text.Background = SystemColors.WindowBrush;
            text.Foreground = SystemColors.WindowTextBrush;
            dock.Children.Add(text);

            // Create menu items.
            MenuItem itemColor = new MenuItem();
            itemColor.Header = "_Color";
            menu.Items.Add(itemColor);

            MenuItem itemForeground = new MenuItem();
            itemForeground.Header = "_Foreground";
            itemForeground.SubmenuOpened += ForegroundOnOpened;
            itemColor.Items.Add(itemForeground);

            FillWithColors(itemForeground, ForegroundOnClick);

            MenuItem itemBackground = new MenuItem();
            itemBackground.Header = "_Background";
            itemBackground.SubmenuOpened += BackgroundOnOpened;
            itemColor.Items.Add(itemBackground);
```

```
            FillWithColors(itemBackground, BackgroundOnClick);
        }
        void FillWithColors(MenuItem itemParent, RoutedEventHandler handler)
        {
            foreach (PropertyInfo prop in typeof(Colors).GetProperties())
            {
                Color clr = (Color)prop.GetValue(null, null);
                int iCount = 0;

                iCount += clr.R == 0 || clr.R == 255 ? 1 : 0;
                iCount += clr.G == 0 || clr.G == 255 ? 1 : 0;
                iCount += clr.B == 0 || clr.B == 255 ? 1 : 0;

                if (clr.A == 255 && iCount > 1)
                {
                    MenuItem item = new MenuItem();
                    item.Header = "_" + prop.Name;
                    item.Tag = clr;
                    item.Click += handler;
                    itemParent.Items.Add(item);
                }
            }
        }
        void ForegroundOnOpened(object sender, RoutedEventArgs args)
        {
            MenuItem itemParent = sender as MenuItem;

            foreach (MenuItem item in itemParent.Items)
                item.IsChecked =
                    ((text.Foreground as SolidColorBrush).Color == (Color)item.Tag);
        }
        void BackgroundOnOpened(object sender, RoutedEventArgs args)
        {
            MenuItem itemParent = sender as MenuItem;

            foreach (MenuItem item in itemParent.Items)
                item.IsChecked =
                    ((text.Background as SolidColorBrush).Color == (Color)item.Tag);
        }
        void ForegroundOnClick(object sender, RoutedEventArgs args)
        {
            MenuItem item = sender as MenuItem;
            Color clr = (Color)item.Tag;
            text.Foreground = new SolidColorBrush(clr);
        }
        void BackgroundOnClick(object sender, RoutedEventArgs args)
        {
            MenuItem item = sender as MenuItem;
            Color clr = (Color)item.Tag;
            text.Background = new SolidColorBrush(clr);
        }
    }
}
```

The program creates a top-level menu item of Color and a submenu containing the two items Foreground and Background. For each of these items, the *FillWithColors* method adds individual color items to the nested submenu. The logic is a little elaborate because it restricts the menu to only those colors where at least two of the Red, Green, and Blue primaries are either 0 or 255. (Remove that logic if you want to see how a large menu is handled under the Windows Presentation Foundation.)

For the Foreground menu item, the *ForegroundOnOpened* method handles the *SubmenuOpened* event, while the *ForegroundOnClick* method handles the *Click* events for each of the colors on the Foreground submenu. (The Background menu works similarly.) The *ForegroundOnOpened* handler loops through the items in the *Items* property and sets the value of the *IsChecked* property to *true* if the item corresponds to the current foreground color of the *TextBlock*, and *false* otherwise. The *ForegroundOnClick* method doesn't have to bother with checking and unchecking and needs only to create a new brush for the *TextBlock*.

Can we get the actual colors into the menu? Yes, of course, and the *MenuItem* class has an *Icon* property intended for little pictures (or whatever) at the left side of the item. In the *if* block in the *FillWithColors* method, any time after the *MenuItem* has been created, add the following code. I've already provided the *using* directive for *System.Windows.Shapes*.

```
Rectangle rect = new Rectangle();
rect.Fill = new SolidColorBrush(clr);
rect.Width = 2 * (rect.Height = 12);
item.Icon = rect;
```

The checkmarks share the space with any *Icon* property that may be set.

You'll probably recall the *ColorGrid* control from Chapter 11 and the similar (but simpler) *ColorGridBox* control from Chapter 13. You can use these controls as menu items, and the following program shows how. This project requires a link to the ColorGridBox.cs file. Notice that this program contains a *using* directive for that project.

SelectColorFromMenuGrid.cs

```
//-----------------------------------------------------
// SelectColorFromMenuGrid.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using Petzold.SelectColorFromGrid;
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.SelectColorFromMenuGrid
{
    public class SelectColorFromMenuGrid : Window
    {
        [STAThread]
        public static void Main()
```

```
        {
            Application app = new Application();
            app.Run(new SelectColorFromMenuGrid());
        }
        public SelectColorFromMenuGrid()
        {
            Title = "Select Color from Menu Grid";

            // Create DockPanel.
            DockPanel dock = new DockPanel();
            Content = dock;

            // Create Menu docked at top.
            Menu menu = new Menu();
            dock.Children.Add(menu);
            DockPanel.SetDock(menu, Dock.Top);

            // Create TextBlock filling the rest.
            TextBlock text = new TextBlock();
            text.Text = Title;
            text.FontSize = 32;
            text.TextAlignment = TextAlignment.Center;
            dock.Children.Add(text);

            // Add items to menu.
            MenuItem itemColor = new MenuItem();
            itemColor.Header = "_Color";
            menu.Items.Add(itemColor);

            MenuItem itemForeground = new MenuItem();
            itemForeground.Header = "_Foreground";
            itemColor.Items.Add(itemForeground);

            // Create ColorGridBox and bind with Foreground of window.
            ColorGridBox clrbox = new ColorGridBox();
            clrbox.SetBinding(ColorGridBox.SelectedValueProperty, "Foreground");
            clrbox.DataContext = this;
            itemForeground.Items.Add(clrbox);

            MenuItem itemBackground = new MenuItem();
            itemBackground.Header = "_Background";
            itemColor.Items.Add(itemBackground);

            // Create ColorGridBox and bind with Background of window.
            clrbox = new ColorGridBox();
            clrbox.SetBinding(ColorGridBox.SelectedValueProperty, "Background");
            clrbox.DataContext = this;
            itemBackground.Items.Add(clrbox);
        }
    }
}
```

As in the previous program, this program creates a top-level item of Color with Foreground and Background items in the Color submenu. Instead of adding multiple items to the

Foreground and Background submenus, the program adds just one item to each—an object of type *ColorGridBox*. Because *ColorGridBox* was written so that the *SelectedValue* property is an object of type *Brush*, it is possible to avoid event handlers entirely and simply provide bindings between the *SelectedValueProperty* dependency property of the *ColorGridBox* and the *Foreground* and *Background* properties of the window.

You've seen how handling the *SubmenuOpen* event can be useful for checking items on the menu. Handling this event is particularly common in programs that need to disable certain menu items. The Edit menu is one common example. A program that has the ability to transfer text in and out of the clipboard shouldn't enable the Paste item unless the clipboard actually contains text. The Cut, Copy, and Delete options should be enabled only if the program is able to copy something to the clipboard.

The following program does nothing but implement an Edit menu that lets you Cut, Copy, Paste, and Delete the text in a *TextBlock* element. The project also includes four bitmaps as application resources that it uses to create *Image* elements that it sets to the *Icon* property of each menu item. (I obtained these images from the library shipped with Microsoft Visual Studio; however, these images originally had resolutions of 72 dots per inch so I changed them to 96 DPI.)

CutCopyAndPaste.cs

```
//-----------------------------------------------
// CutCopyAndPaste.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Petzold.CutCopyAndPaste
{
    public class CutCopyAndPaste : Window
    {
        TextBlock text;
        protected MenuItem itemCut, itemCopy, itemPaste, itemDelete;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CutCopyAndPaste());
        }
        public CutCopyAndPaste()
        {
            Title = "Cut, Copy, and Paste";
```

```
        // Create DockPanel.
        DockPanel dock = new DockPanel();
        Content = dock;

        // Create Menu docked at top.
        Menu menu = new Menu();
        dock.Children.Add(menu);
        DockPanel.SetDock(menu, Dock.Top);

        // Create TextBlock filling the rest.
        text = new TextBlock();
        text.Text = "Sample clipboard text";
        text.HorizontalAlignment = HorizontalAlignment.Center;
        text.VerticalAlignment = VerticalAlignment.Center;
        text.FontSize = 32;
        text.TextWrapping = TextWrapping.Wrap;
        dock.Children.Add(text);

        // Create Edit menu.
        MenuItem itemEdit = new MenuItem();
        itemEdit.Header = "_Edit";
        itemEdit.SubmenuOpened += EditOnOpened;
        menu.Items.Add(itemEdit);

        // Create items on Edit menu.
        itemCut = new MenuItem();
        itemCut.Header = "Cu_t";
        itemCut.Click += CutOnClick;
        Image img = new Image();
        img.Source = new BitmapImage(
                new Uri("pack://application:,,/Images/CutHS.png"));
        itemCut.Icon = img;
        itemEdit.Items.Add(itemCut);

        itemCopy = new MenuItem();
        itemCopy.Header = "_Copy";
        itemCopy.Click += CopyOnClick;
        img = new Image();
        img.Source = new BitmapImage(
                new Uri("pack://application:,,/Images/CopyHS.png"));
        itemCopy.Icon = img;
        itemEdit.Items.Add(itemCopy);

        itemPaste = new MenuItem();
        itemPaste.Header = "_Paste";
        itemPaste.Click += PasteOnClick;
        img = new Image();
        img.Source = new BitmapImage(
                new Uri("pack://application:,,/Images/PasteHS.png"));
        itemPaste.Icon = img;
        itemEdit.Items.Add(itemPaste);
```

```
            itemDelete = new MenuItem();
            itemDelete.Header = "_Delete";
            itemDelete.Click += DeleteOnClick;
            img = new Image();
            img.Source = new BitmapImage(
                            new Uri("pack://application:,,/Images/DeleteHS.png"));
            itemDelete.Icon = img;
            itemEdit.Items.Add(itemDelete);
        }
        void EditOnOpened(object sender, RoutedEventArgs args)
        {
            itemCut.IsEnabled =
            itemCopy.IsEnabled =
            itemDelete.IsEnabled = text.Text != null && text.Text.Length > 0;
            itemPaste.IsEnabled = Clipboard.ContainsText();
        }
        protected void CutOnClick(object sender, RoutedEventArgs args)
        {
            CopyOnClick(sender, args);
            DeleteOnClick(sender, args);
        }
        protected void CopyOnClick(object sender, RoutedEventArgs args)
        {
            if (text.Text != null && text.Text.Length > 0)
                Clipboard.SetText(text.Text);
        }
        protected void PasteOnClick(object sender, RoutedEventArgs args)
        {
            if (Clipboard.ContainsText())
                text.Text = Clipboard.GetText();
        }
        protected void DeleteOnClick(object sender, RoutedEventArgs args)
        {
            text.Text = null;
        }
    }
}
```

The program stores the Cut, Copy, Paste, and Delete *MenuItem* objects as fields and accesses them during the *EditOnOpened* event handler. The handler enables Cut, Copy, and Delete only if the *TextBlock* contains at least one character of text. To enable the Paste item, the handler uses the return value from the static *Clipboard.ContainsText* method.

The *PasteOnClick* method uses the static *Clipboard.GetText* method to copy text from the clipboard. Similarly, *CopyOnClick* calls *Clipboard.SetText*. The Delete command doesn't need to access the clipboard and simply sets the *Text* property of the *TextBlock* element to *null*. The *CutOnClick* event handler take advantage of the fact that a Cut is simply a Copy followed by a Delete by calling *CopyOnClick* and *DeleteOnClick*.

The CutCopyAndPaste program has the standard underlined characters for the Edit menu. A user can trigger the Paste command by pressing Alt, E, P for example. However, these edit commands also have standard keyboard shortcuts called *accelerators*: Ctrl+X for Cut, Ctrl+C for Copy, Ctrl+V for Paste, and the Delete key for Delete. You'll notice that these aren't implemented in the CutCopyAndPaste program.

It's fairly easy to get the text "Ctrl+X" displayed alongside the Cut item. Just set the *InputGestureText* property of the menu item:

```
itemCut.InputGestureText = "Ctrl+X";
```

However, actually triggering a Cut command when the user types Ctrl+X is something else entirely. It does not happen automatically and you have two options to make it happen: You can handle the keyboard input on your own (which I'll show shortly) or you can use command bindings (which I'll show after that).

If you decide to handle the keyboard input on your own, you should treat that input as high priority. In other words, you want to examine keyboard input for possible menu accelerators before anybody else gets hold of that input, and that means you'll probably override the *OnPreviewKeyDown* method of the window. If a keystroke corresponds to an enabled menu item, carry out the command and set the *Handled* property of the event arguments to *true*.

The job of handling keyboard input to trigger menu items is eased somewhat by the *KeyGesture* class. You can define an object of type *KeyGesture* for Ctrl+X like this:

```
KeyGesture gestCut = new KeyGesture(Key.X, ModifierKeys.Control);
```

This class doesn't include much, and the only reason to use it is to make use of the *Matches* method that accepts an *InputEventArgs* argument. You can call the *Matches* method during the *OnPreviewKeyDown* override using the *KeyEventArgs* argument delivered with that event. (*KeyEventArgs* derives from *InputEventArgs*.) The *Matches* method will recognize that its argument is actually a *KeyEventArgs*, and returns *true* if the key being pressed is the same as the key defined in the *KeyGesture* object. The processing in your *OnPreviewKeyDown* override might look like this:

```
if (gestCut.Matches(null, args))
{
    CutOnClick(this, args);
    args.Handled = true;
}
```

You can pass the *KeyEventArgs* object directly to *CutOnClick* because *KeyEventArgs* derives from *RoutedEventArgs*. However, this code doesn't check whether a Cut item is actually valid before calling the Click handler. One simple approach you might consider is checking whether the *itemCut* menu item is enabled. But that won't work because *itemCut* is enabled and disabled only when the drop-down menu is displayed.

Fortunately, you'll notice that the *CopyOnClick* and *PasteOnClick* methods in the CutCopyAndPaste program don't actually perform the Copy and Paste operations unless the commands are valid. Those checks allow the following program to inherit from CutCopyAndPaste to implement the standard keyboard accelerators for the Edit menu. This project requires a link to the CutCopyAndPaste.cs source code file. There's no *using* directive for that project's namespace; instead, the *class* definition refers to the fully qualified name of the *CutCopyAndPaste* class.

ControlXCV.cs

```
//-----------------------------------------
// ControlXCV.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.ControlXCV
{
    public class ControlXCV : Petzold.CutCopyAndPaste.CutCopyAndPaste
    {
        KeyGesture gestCut = new KeyGesture(Key.X, ModifierKeys.Control);
        KeyGesture gestCopy = new KeyGesture(Key.C, ModifierKeys.Control);
        KeyGesture gestPaste = new KeyGesture(Key.V, ModifierKeys.Control);
        KeyGesture gestDelete = new KeyGesture(Key.Delete);

        [STAThread]
        public new static void Main()
        {
            Application app = new Application();
            app.Run(new ControlXCV());
        }
        public ControlXCV()
        {
            Title = "Control X, C, and V";

            itemCut.InputGestureText = "Ctrl+X";
            itemCopy.InputGestureText = "Ctrl+C";
            itemPaste.InputGestureText = "Ctrl+V";
            itemDelete.InputGestureText = "Delete";
        }
        protected override void OnPreviewKeyDown(KeyEventArgs args)
        {
            base.OnKeyDown(args);
            args.Handled = true;

            if (gestCut.Matches(null, args))
                CutOnClick(this, args);

            else if (gestCopy.Matches(null, args))
                CopyOnClick(this, args);

            else if (gestPaste.Matches(null, args))
                PasteOnClick(this, args);
```

```
            else if (gestDelete.Matches(null, args))
                DeleteOnClick(this, args);

            else
                args.Handled = false;
        }
    }
}
```

Whenever you inherit from a class that defines a *Main* method (as the *CutCopyAndPaste* class does) and you supply a new *Main* method (as the *ControlXCV* class does) you need to tell Visual Studio which *Main* method is the true entry point to the program. Select Project Properties and change Startup Object to the class with the *Main* method you want to use.

The program defines and sets the four *KeyGesture* objects as fields, and also needs to set the *InputGestureText* property of each *MenuItem* to the corresponding string. (Unfortunately, *KeyGesture* itself doesn't provide that information through its *ToString* method or otherwise.) The *OnPreviewKeyDown* method begins by setting the *Handled* property of its event arguments to *true*, and then resets it to *false* if the key doesn't match one of the define gestures.

If you have more than just a few *KeyGesture* objects floating around, you'll probably want to store them in a collection. You can define a field that creates a generic *Dictionary* like this:

```
Dictionary<KeyGesture, RoutedEventHandler> gests =
                                new Dictionary<KeyGesture, RoutedEventHandler>();
```

The constructor of your window can fill it up with the *KeyGesture* objects and their associated event handlers:

```
gests.Add(new KeyGesture(Key.X, ModifierKeys.Control), CutOnClick);
gests.Add(new KeyGesture(Key.C, ModifierKeys.Control), CopyOnClick);
gests.Add(new KeyGesture(Key.V, ModifierKeys.Control), PasteOnClick);
gests.Add(new KeyGesture(Key.Delete), DeleteOnClick);
```

The *OnPreviewKeyDown* method can then search for a match and call the corresponding event handler by looping through the dictionary:

```
foreach (KeyGesture gest in gests.Keys)
    if (gest.Matches(null, args))
    {
        gests[gest](this, args);
        args.Handled = true;
    }
```

The first statement in the *if* block line indexes the *Dictionary* object named *gests* with the matching *KeyGesture* object named *gest*. The result is the *RoutedEventHandler* object, which the statement calls by passing arguments of *this* and the *KeyEventArgs* object.

If you'd rather not call the *Click* event handlers directly, you could instead define a *Dictionary* with the *MenuItem* as the *Value*:

```
Dictionary<KeyGesture, MenuItem> gests =
                                new Dictionary<KeyGesture, MenuItem>();
```

You add entries to this dictionary like so:

```
gests.Add(new KeyGesture(Key.X, ModifierKeys.Control), itemCut);
```

And now the *OnKeyDown* processing looks like this:

```
foreach (KeyGesture gest in gests.Keys)
    if (gest.Matches(null, args))
        gests[gest].RaiseEvent(
            new RoutedEventArgs(MenuItem.ClickEvent, gests[gest]));
```

By this time you may have concluded that command bindings probably provide a simpler approach, and they certainly do. The CommandTheButton program in Chapter 4 showed how to use command bindings with a button. Using them with menu items is quite similar. Generally you'll be using static properties of type *RoutedUICommand* from the *ApplicationCommands* class and (for more esoteric applications) from the *ComponentCommands*, *EditingCommands*, *MediaCommands*, and *NavigationCommands* classes, but you can also make your own, as I'll demonstrate.

To use one of the predefined static properties, you set the *Command* property of the *MenuItem* like this:

```
itemCut.Command = ApplicationCommands.Cut;
```

If you don't set the *Header* property of the *MenuItem*, it will use the *Text* property of the *RoutedUICommand*, which is *almost* OK except there's no preceding underline. Regardless, the *MenuItem* automatically adds the "Ctrl+X" text to the menu item.

The other crucial step is creating a command binding based on the *RoutedUICommand* object, and adding it to the *CommandBindings* collection of the window:

```
CommandBindings.Add(new CommandBinding(ApplicationCommands.Cut,
                                        CutOnExecute, CutCanExecute));
```

This command binding automatically provides for keyboard handling of the standard accelerators associated with the commands. As you'll see, the accelerator is defined within the *RoutedUICommand* object. The command binding associates the command with the *CommandBinding* events *CanExecute* and *Executed*. When using *RoutedUICommand* objects, there is no need to provide an event handler specifically to enable and disable the items on the Edit menu. The enabling and disabling occurs via the *CanExecute* handlers by setting the *CanExecute* property of the *CanExecuteRoutedEventArgs* to *true* or *false*. You can share *CanExecute* handlers among several menu items if appropriate.

Here's a program that implements command bindings for the four standard items on the Edit menu.

CommandTheMenu.cs

```
//----------------------------------------------
// CommandTheMenu.cs (c) 2006 by Charles Petzold
//----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.CommandTheMenu
{
    public class CommandTheMenu : Window
    {
        TextBlock text;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CommandTheMenu());
        }
        public CommandTheMenu()
        {
            Title = "Command the Menu";

            // Create DockPanel.
            DockPanel dock = new DockPanel();
            Content = dock;

            // Create Menu docked at top.
            Menu menu = new Menu();
            dock.Children.Add(menu);
            DockPanel.SetDock(menu, Dock.Top);

            // Create TextBlock filling the rest.
            text = new TextBlock();
            text.Text = "Sample clipboard text";
            text.HorizontalAlignment = HorizontalAlignment.Center;
            text.VerticalAlignment = VerticalAlignment.Center;
            text.FontSize = 32;     // ie, 24 points
            text.TextWrapping = TextWrapping.Wrap;
            dock.Children.Add(text);

            // Create Edit menu.
            MenuItem itemEdit = new MenuItem();
            itemEdit.Header = "_Edit";
            menu.Items.Add(itemEdit);
```

```
            // Create items on Edit menu.
            MenuItem itemCut = new MenuItem();
            itemCut.Header = "Cu_t";
            itemCut.Command = ApplicationCommands.Cut;
            itemEdit.Items.Add(itemCut);

            MenuItem itemCopy = new MenuItem();
            itemCopy.Header = "_Copy";
            itemCopy.Command = ApplicationCommands.Copy;
            itemEdit.Items.Add(itemCopy);

            MenuItem itemPaste = new MenuItem();
            itemPaste.Header = "_Paste";
            itemPaste.Command = ApplicationCommands.Paste;
            itemEdit.Items.Add(itemPaste);

            MenuItem itemDelete = new MenuItem();
            itemDelete.Header = "_Delete";
            itemDelete.Command = ApplicationCommands.Delete;
            itemEdit.Items.Add(itemDelete);

            // Add command bindings to window collection.
            CommandBindings.Add(new CommandBinding(ApplicationCommands.Cut,
                                        CutOnExecute, CutCanExecute));
            CommandBindings.Add(new CommandBinding(ApplicationCommands.Copy,
                                        CopyOnExecute, CutCanExecute));
            CommandBindings.Add(new CommandBinding(ApplicationCommands.Paste,
                                        PasteOnExecute, PasteCanExecute));
            CommandBindings.Add(new CommandBinding(ApplicationCommands.Delete,
                                        DeleteOnExecute, CutCanExecute));
        }
        void CutCanExecute(object sender, CanExecuteRoutedEventArgs args)
        {
            args.CanExecute = text.Text != null && text.Text.Length > 0;
        }
        void PasteCanExecute(object sender, CanExecuteRoutedEventArgs args)
        {
            args.CanExecute = Clipboard.ContainsText();
        }
        void CutOnExecute(object sender, ExecutedRoutedEventArgs args)
        {
            ApplicationCommands.Copy.Execute(null, this);
            ApplicationCommands.Delete.Execute(null, this);
        }
        void CopyOnExecute(object sender, ExecutedRoutedEventArgs args)
        {
            Clipboard.SetText(text.Text);
        }
        void PasteOnExecute(object sender, ExecutedRoutedEventArgs args)
        {
            text.Text = Clipboard.GetText();
        }
        void DeleteOnExecute(object sender, ExecutedRoutedEventArgs args)
```

```
        {
            text.Text = null;
        }
    }
}
```

Notice that Cut, Copy, and Delete all share the same *CanExecute* handler.

Although it's nice to implement the standard Edit items with command bindings, it's even more fun to create new ones. You can add this code to the CommandTheMenu program at the end of the constructor. The object here is to create a new command called Restore that restores the *TextBlock* to its original text. The Restore command has a keyboard shortcut of Ctrl+R.

Because a particular *RoutedUICommand* can be associated with multiple key gestures, a collection must be defined even if you want only one gesture:

```
InputGestureCollection collGestures = new InputGestureCollection();
```

Add the appropriate *KeyGesture* to this collection:

```
collGestures.Add(new KeyGesture(Key.R, ModifierKeys.Control));
```

And then create a *RoutedUICommand*:

```
RoutedUICommand commRestore =
    new RoutedUICommand("_Restore", "Restore", GetType(), collGestures);
```

The first argument to the constructor becomes the *Text* property and the second is the *Name* property. (Notice that I've added an underline to the *Text* property.) The third argument is the owner (which can be simply the *Window* object) and the fourth argument is a collection of keyboard gestures.

Now the *MenuItem* can be defined and added to the menu:

```
MenuItem itemRestore = new MenuItem();
itemRestore.Header = "_Restore";
itemRestore.Command = commRestore;
itemEdit.Items.Add(itemRestore);
```

Setting the *Header* property isn't required because it picks up the *Text* property from the *RoutedUICommand*. The command must also be added to the window's command collection. Here's where event handlers are specified:

```
CommandBindings.Add(new CommandBinding(commRestore, RestoreOnExecute));
```

The *RestoreOnExecute* handler simply restores the *TextBlock* text to its original value:

```
void RestoreOnExecute(object sender, ExecutedRoutedEventArgs args)
{
    text.Text = "Sample clipboard text";
}
```

The programs so far in this chapter have dealt with the *Menu* control that normally sits near the top of the window. The Windows Presentation Foundation also includes a *ContextMenu* control, customarily invoked in response to a click of the right mouse button.

Like *ToolTip*, *ContextMenu* is a property as well as a class. And like *ToolTip* again, a *ContextMenu* property is defined by both *FrameworkElement* and *FrameworkContentElement*. If you'd like, you can define a *ContextMenu* object that is associated with a particular element, and then assign that *ContextMenu* object to the *ContextMenu* property of the element. The context menu then opens whenever the user right-clicks that element. You can install event handlers to initialize the menu when it opens, and to be notified of clicks and checks.

If you don't set the *ContextMenu* object to the *ContextMenu* property of some element, you need to open the context menu "manually"—probably in response to a *MouseRightButtonUp* event. Fortunately, opening the context menu is as easy as setting the *IsOpen* property to *true*. By default, the context menu appears at the location of the mouse pointer.

The following program is similar to the ToggleBoldAndItalic program from Chapter 3. It displays a famous quotation and lets you right-click each word with the mouse. A context menu is displayed that lists formatting options Bold, Italic, Underline, Overline, Strikethrough, and Baseline. The program creates only one *ContextMenu* object for use with all the words of the text, and doesn't attempt to keep track of the formatting of each word. Instead, whenever the context menu is displayed, it is initialized with the formatting of the particular word being clicked.

PopupContextMenu.cs

```
//-------------------------------------------------
// PopupContextMenu.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.PopupContextMenu
{
    public class PopupContextMenu : Window
    {
        ContextMenu menu;
        MenuItem itemBold, itemItalic;
        MenuItem[] itemDecor;
        Inline inlClicked;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new PopupContextMenu());
        }
```

```
        public PopupContextMenu()
        {
            Title = "Popup Context Menu";

            // Create ContextMenu.
            menu = new ContextMenu();

            // Add an item for "Bold".
            itemBold = new MenuItem();
            itemBold.Header = "Bold";
            menu.Items.Add(itemBold);

            // Add an item for "Italic".
            itemItalic = new MenuItem();
            itemItalic.Header = "Italic";
            menu.Items.Add(itemItalic);

            // Get all the TextDecorationLocation members.
            TextDecorationLocation[] locs =
                (TextDecorationLocation[])
                    Enum.GetValues(typeof(TextDecorationLocation));

            // Create an array of MenuItem objects and fill them up.
            itemDecor = new MenuItem[locs.Length];

            for (int i = 0; i < locs.Length; i++)
            {
                TextDecoration decor = new TextDecoration();
                decor.Location = locs[i];

                itemDecor[i] = new MenuItem();
                itemDecor[i].Header = locs[i].ToString();
                itemDecor[i].Tag = decor;
                menu.Items.Add(itemDecor[i]);
            }

            // Use one handler for the entire context menu.
            menu.AddHandler(MenuItem.ClickEvent,
                            new RoutedEventHandler(MenuOnClick));

            // Create a TextBlock as content of the window.
            TextBlock text = new TextBlock();
            text.FontSize = 32;
            text.HorizontalAlignment = HorizontalAlignment.Center;
            text.VerticalAlignment = VerticalAlignment.Center;
            Content = text;

            // Break a famous quotation up into words.
            string strQuote = "To be, or not to be, that is the question";
            string[] strWords = strQuote.Split();

            // Make each word a Run, and add to the TextBlock.
            foreach (string str in strWords)
            {
                Run run = new Run(str);
```

```
                // Make sure that TextDecorations is an actual collection!
                run.TextDecorations = new TextDecorationCollection();
                text.Inlines.Add(run);
                text.Inlines.Add(" ");
            }
        }
        protected override void OnMouseRightButtonUp(MouseButtonEventArgs args)
        {
            base.OnMouseRightButtonUp(args);

            if ((inlClicked = args.Source as Inline) != null)
            {
                // Check the menu items according to properties of the InLine.
                itemBold.IsChecked = (inlClicked.FontWeight == FontWeights.Bold);
                itemItalic.IsChecked = (inlClicked.FontStyle == FontStyles.Italic);

                foreach (MenuItem item in itemDecor)
                    item.IsChecked = (inlClicked.TextDecorations.Contains
                            (item.Tag as TextDecoration));

                // Display context menu.
                menu.IsOpen = true;
                args.Handled = true;
            }
        }
        void MenuOnClick(object sender, RoutedEventArgs args)
        {
            MenuItem item = args.Source as MenuItem;

            item.IsChecked ^= true;

            // Change the Inline based on the checked or unchecked item.
            if (item == itemBold)
                inlClicked.FontWeight =
                    (item.IsChecked ? FontWeights.Bold : FontWeights.Normal);

            else if (item == itemItalic)
                inlClicked.FontStyle =
                    (item.IsChecked ? FontStyles.Italic : FontStyles.Normal);

            else
            {
                if (item.IsChecked)
                    inlClicked.TextDecorations.Add(item.Tag as TextDecoration);
                else
                    inlClicked.TextDecorations.Remove(item.Tag as TextDecoration);
            }
            (inlClicked.Parent as TextBlock).InvalidateVisual();
        }
    }
}
```

The first part of the window constructor is devoted to creating the *ContextMenu* object. After adding Bold and Italic items to the menu, the window constructor obtains the members of the *TextDecorationLocation* enumeration. These members are *Underline*, *Overline*, *Strikethrough*, and *Baseline*. The constructor uses the *AddHandler* method of the *ContextMenu* to assign a single *Click* handler for all the menu items.

The *Split* method of the *String* class divides the quotation into words. These are made into objects of type *Run* and patched together into a single *TextBlock* object. Notice that a *TextDecorationCollection* is explicitly created for each *Run* object. This collection does not exist by default and the *TextDecorations* property is normally *null*.

Although the *OnMouseRightButtonUp* method seemingly obtains mouse events to the window, event routing provides that if an *Inline* object is clicked, the *Source* property of the event arguments will indicate that object. (Recall that *Run* derives from *Inline*.) The event handler can then initialize the menu based on the properties of the clicked word.

The *MenuOnClick* manually toggles the *IsChecked* property of the clicked item. This isn't really necessary because the menu disappears when it's clicked, but the event handler uses the new value of this *IsChecked* property to determine how to change the formatting of the clicked *Inline* object.

I began this chapter by noting that the menu occupies a regal position near the top of the window. As you know, directly below the menu is often a toolbar, and (sometimes almost as important) a status bar often sits at the bottom of a window. These are the subjects of the next chapter.