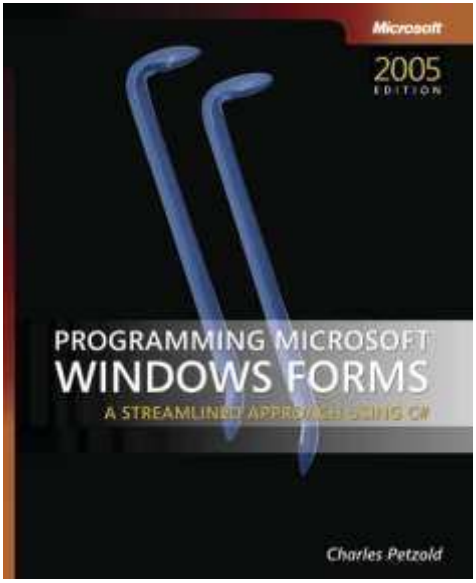


Programming Microsoft® Windows® Forms

Charles Petzold

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/7824.aspx

9780735621534
Publication Date: November 2005

Microsoft Press

Table of Contents

Chapter 4

Custom Controls

Some people are never satisfied, and that probably goes double for programmers. Even with the many controls available under Windows Forms, it is still sometimes desirable to ascend a step (or perhaps several) beyond the customary controls into the realms of the custom control.

From a programming perspective, a custom control is a class that you define that derives—either directly or indirectly—from *Control*. A custom control can be an enhancement of an existing control or an entirely new control. Although you can perform a significant amount of customization just by installing event handlers on existing controls and processing those events, a new class is required if you need to *entirely* override default event handling. For example, you can add new visuals to a *Button* control by installing a *Paint* event handler, but you can't prevent the *Button* from displaying its own visuals as well unless you create a new class and override *OnPaint*.

You also need to create a new class if you want to add fields or properties to an existing control. However, if you just need to attach some arbitrary data to a control, you might consider using the *Tag* property provided specifically for this purpose. The property is defined as type *object*, so some casting will be required when you access it, but it's ideal for simple control identification and attaching arbitrary data.

As with most any programming job, the real benefit of custom controls comes with reusing them in multiple applications or in making them available to other programmers, either for cash or just glory.

Enhancing Existing Controls

A control is basically a filter through which user input is interpreted and consolidated into actions such as events. In most cases, a control must perform three crucial jobs. First, it displays something on a screen to identify itself to the user. Second, it handles user input, generally from the keyboard and mouse. (A control might also be specially programmed to handle stylus input on a Tablet PC, or even to respond to voice input.) Third, the control fires events to notify the application using the control of certain changes.

Because existing controls have already been designed and tested to perform these three functions, it is much easier to enhance an existing control (perhaps by deriving from *Button*, for example) rather than to start from scratch by deriving from *Control*.

Overriding Methods

Here's an extremely simple example, which is a button that beeps when it is clicked with the mouse:

BeepButton.cs

```
//-----------------------------------------
// BeepButton.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Media;
using System.Windows.Forms;

class BeepButton : Button
{
    protected override void OnClick(EventArgs args)
    {
        SystemSounds.Exclamation.Play();
        base.OnClick(args);
    }
}
```

BeepButton simply inherits from *Button* and overrides the protected *OnClick* method. The overriding method makes a sound by using two of the three classes from the *System.Media* namespace, which is new in the .NET Framework 2.0. The *SystemsSounds* class contains several static properties—named *Asterisk*, *Beep*, *Exclamation*, *Hand*, and *Question*—associated with different sounds. These properties return objects of type *SystemSound*. Notice the difference in class names: the static methods in *SystemsSounds* (plural) return an object of type *SystemSound* (singular). The *SystemSound* class has a single method named *Play* to play the sound.

It is *imperative* that the *OnClick* method in your new class call the *OnClick* method in the base class:

```
base.OnClick(args);
```

Without this call, any program using the *BeepButton* control will not have access to the *Click* event. Here's why: *Button* inherits the *Click* event and the *OnClick* method from *Control*. The code in *Control* that defines the *Click* event probably looks something like this:

```
public event EventHandler Click;
```

The *EventHandler* part of that statement indicates that any event handlers installed for the *Click* event be defined in accordance with the *EventHandler* delegate.

The *Control* class doesn't have to worry about the mechanism for letting the program attach and detach event handlers. That happens behind the scenes. But *Control* is responsible for fir-

ing the *Click* event, and that happens in its *OnClick* method, which probably looks something like this:

```
protected virtual void OnClick(EventArgs args)
{
    ...
    if (Click != null)
        Click(this, args);
    ...
}
```

This *OnClick* method probably gets called from at least two places. The control's *OnMouseDown* method—which occurs when the mouse cursor is positioned on top of the control and a mouse button is depressed—undoubtedly makes a call to *OnClick*. For buttons, *OnClick* is also called when the button has input focus and the user presses the spacebar or Enter key.

I've indicated with ellipses that *OnClick* might or might not perform some other duties, but it definitely triggers the *Click* event. The code I've shown can be translated like this: "If there are any *Click* event handlers installed, call those event handlers with the current object as the first argument and an *EventArgs* object as the second argument." If you define a class that inherits from *Control* (either directly or indirectly) and you override the *OnClick* method without calling the method in the base class, the code to call all the *Click* event handlers does not get executed. (Of course, if disabling the *Click* event is part of your nefarious strategy in creating a new control, don't bother calling the *OnClick* method in the base class.)

Code examples showing how an *OnClick* method calls the method in the base class usually put this call at the very beginning of the method:

```
protected override void OnClick(EventArgs args)
{
    base.OnClick(args);
    ...
}
```

Sometimes you might want the code in the base method executed first, but in this particular example, it didn't work well at all (and you'll see why shortly), so I put the call to *base.OnClick* at the end of *OnClick* in *ButtonBeep*.

Here's a simple "demo" program that creates an object of type *BeepButton* and installs a *Click* event handler on the button to display a message box:

BeepButtonDemo.cs

```
//-----------------------------------------------
// BeepButtonDemo.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
class BeepButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new BeepButtonDemo());
    }
    public BeepButtonDemo()
    {
        Text = "BeepButton Demonstration";

        BeepButton btn = new BeepButton();
        btn.Parent = this;
        btn.Location = new Point(100, 100);
        btn.AutoSize = true;
        btn.Text = "Click the BeepButton";
        btn.Click += ButtonOnClick;
    }
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        SilentMsgBox.Show("The BeepButton has been clicked", Text);
    }
}
```

Actually, I couldn't use the regular *MessageBox* class because that class itself makes a sound as the message box is displayed. Instead, I duplicated some of the functionality of *MessageBox* in this class:

SilentMsgBox.cs

```
//--------------------------------------------
// SilentMsgBox.cs (c) 2005 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SilentMsgBox
{
    public static DialogResult Show(string strMessage, string strCaption)
    {
        Form frm = new Form();
        frm.StartPosition = FormStartPosition.CenterScreen;
        frm.FormBorderStyle = FormBorderStyle.FixedDialog;
        frm.MinimizeBox = frm.MaximizeBox = frm.ShowInTaskbar = false;
        frm.AutoSize = true;
        frm.AutoSizeMode = AutoSizeMode.GrowAndShrink;
        frm.Text = strCaption;

        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = frm;
        pnl.AutoSize = true;
        pnl.FlowDirection = FlowDirection.TopDown;
```

```
            pnl.WrapContents = false;
            pnl.Padding = new Padding(pnl.Font.Height);

            Label lbl = new Label();
            lbl.Parent = pnl;
            lbl.AutoSize = true;
            lbl.Anchor = AnchorStyles.None;
            lbl.Margin = new Padding(lbl.Font.Height);
            lbl.Text = strMessage;

            Button btn = new Button();
            btn.Parent = pnl;
            btn.AutoSize = true;
            btn.Anchor = AnchorStyles.None;
            btn.Margin = new Padding(btn.Font.Height);
            btn.Text = "OK";
            btn.DialogResult = DialogResult.OK;

            return frm.ShowDialog();
        }
    }
```

The BeepButton.cs, BeepButtonDemo.cs, and SilentMsgBox.cs files are all part of the BeepButtonDemo project.

Just to assure yourself that everything I said about the *Click* event handler is true, comment out the call to the *base.OnClick* method in *BeepButton*, recompile, and take careful note that *BeepButtonDemo* no longer gets notified of the *Click* event.

Now try this: in the *OnClick* method in *BeepButton*, swap the order of the two statements so that the *OnClick* method in the base class is called before the sound is played:

```
protected override void OnClick(EventArgs args)
{
    base.OnClick(args);
    SystemSounds.Exclamation.Play();
}
```

When you click the button with the mouse, this *OnClick* method is called—probably from the *OnMouseDown* method in *Control*. In this altered code, *BeepButton* first makes a call to the *OnClick* method in its base class. That base class is *Button*, but the *OnClick* method in *Button* calls the *OnClick* method in *its* base class, and so on, and eventually the *OnClick* method in *Control* is called. That *OnClick* method is responsible for executing the code that calls all the event handlers installed for *Click*. The *BeepButtonDemo* has installed such an event handler, so the *ButtonOnClick* method in *BeepButtonDemo* is called. That method calls the static *Show* method in *SilentMsgBox*, which (like *MessageBox*) displays a modal dialog box and waits for the user to dismiss it. When the user ends the message box, the *Show* call returns control back to *ButtonOnClick*, which in turn returns control back to the *OnClick* method in *BeepButton*,

which finally (in the altered code) plays the sound—unfortunately, long after the user clicked the button.

The lesson is: You can choose when an *On* method calls the method in the base class. Choose wisely.

Adding New Properties

Besides demonstrating how to enhance existing controls, the *BeepButton* class also demonstrated the use of two of the three classes in the *System.Media* namespace. The third class in that namespace is *SoundPlayer*, which is used by the next program to create a new control called *SoundButton*. The *SoundButton* control is similar to *BeepButton* except that it plays a MicrosoftWindows waveform (.wav) file rather than a simple beep. This feature allows your buttons to be accompanied by the soothing voice of Bart Simpson, for example.

This class also inherits from *Button*. The first thing it does is create and store a *SoundPlayer* object as a field:

SoundButton.cs

```
//-------------------------------------------
// SoundButton.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Media;
using System.Windows.Forms;

class SoundButton : Button
{
    SoundPlayer sndplay = new SoundPlayer();

    public string WaveFile
    {
        set
        {
            sndplay.SoundLocation = value;
            sndplay.LoadAsync();
        }
        get
        {
            return sndplay.SoundLocation;
        }
    }
    public Stream WaveStream
    {
        set
        {
            sndplay.Stream = value;
            sndplay.LoadAsync();
        }
```

```
        get
        {
            return sndplay.Stream;
        }
    }
    protected override void OnClick(EventArgs args)
    {
        if (sndplay.IsLoadCompleted)
            sndplay.Play();

        base.OnClick(args);
    }
}
```

The class defines two new properties named *WaveFile* and *WaveStream*. These correspond to the *SoundPlayer* properties *SoundLocation* and *Stream*, which are the two ways that a program specifies to *SoundPlayer* the source of a waveform file. The *SoundLocation* property of *SoundPlayer* (and the *WaveFile* property of *SoundButton*) is a string that indicates a local file or a URL. *Stream* (and *WaveStream*) is an object of type *Stream*, an abstract class defined in the *System.IO* namespace. The descendents of *Stream* include *FileStream*, which generally refers to an open file, and *MemoryStream*, which is a block of memory accessed as if it were a file. The *Stream* option is particularly useful for embedding waveform files into your executable file and accessing them as resources (as I'll demonstrate shortly). Whichever property is specified most recently is the one that *SoundPlayer* uses to access the waveform file.

Normally, *SoundPlayer* will not load a waveform file into memory until it needs to play it, which in this case occurs during the *OnClick* method. I was afraid that this loading process might slow down processing of *OnClick*, so the two properties load in the waveforms immediately in a separate thread by calling *LoadAsync*. (Using the *Load* method of *SoundPlayer* instead loads the waveform synchronously—that is, in the same thread—which might slow down initialization of the program.)

The normal *Play* command of *SoundPlayer* is asynchronous. The *OnClick* method starts the sound going and then does its other processing, which is a call to the base *OnClick* method and whatever else might happen as a result of that. (The alternative *PlaySync* method of *SoundPlayer* doesn't return until a sound is finished. The *PlayLooping* method is asynchronous and plays the sound repeatedly until *Stop* is called.)

Here's a program that demonstrates three ways to use *SoundButton*: loading a local file (in this case the "ta-da" sound from the Windows Media directory), loading a file from the Internet (a lion's roar from the Oakland Zoo Web site), and using a resource. The SoundButtonDemo project also includes the SoundButton.cs file and a waveform file named MakeItSo.wav, which is my voice expressing the user's wish to "make it so."

SoundButtonDemo.cs

```
//---------------------------------------------
// SoundButtonDemo.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class SoundButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new SoundButtonDemo());
    }
    public SoundButtonDemo()
    {
        Text = "SoundButton Demonstration";

        SoundButton btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 25);
        btn.AutoSize = true;
        btn.Text = "SoundButton with File";
        btn.Click += ButtonOnClick;
        btn.WaveFile = Path.Combine(
            Environment.GetEnvironmentVariable("windir"),
            "Media\\tada.wav");

        btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 125);
        btn.AutoSize = true;
        btn.Text = "SoundButton with URI";
        btn.Click += ButtonOnClick;
        btn.WaveFile = "http://www.oaklandzoo.org/atoz/azlinsnd.wav";

        btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 225);
        btn.AutoSize = true;
        btn.Text = "SoundButton with Resource";
        btn.Click += ButtonOnClick;
        btn.WaveStream = GetType().Assembly.GetManifestResourceStream(
                "SoundButtonDemo.MakeItSo.wav");
    }
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        Button btn = objSrc as Button;
        SilentMsgBox.Show("The SoundButton has been clicked", btn.Text);
    }
}
```

None of these three approaches to obtaining waveform files is trivial. The tada.wav file is located in the *Media* subdirectory of your Windows directory, but that directory might be WINDOWS or WINNT. The program uses the *GetEnvironmentVariable* of the *Environment* class to obtain the directory name and then combines that with the *Media* subdirectory and tada.wav filename.

Specifying a URL is much easier of course, but only if you have complete confidence that the URL will not change over the commercial lifetime of your program, and that the program will have access to a live Internet connection.

The foolproof method for getting access to binary files is to make them part of your executable as resources. You do this by first adding the file to your project. When you click the file in the Solution Explorer in Microsoft Visual Studio, a Properties box will open at the bottom right. It is *very important* to change the Build Action to Embedded Resource. Otherwise, your program will not be able to load the resource at runtime, and you will go mad trying to figure out why.

The last statement of the constructor in *SoundButtonDemo* shows the code to load that binary resource into your program as a *Stream* object. The filename must be preceded with a "resource namespace," which Visual Studio normally sets to the name of the project. You can change that name through the Project Properties dialog box. In the Application section of that dialog box, it is identified by the label "Default namespace."

Control Paint Jobs

One of the most dramatic ways you can modify an existing control is by changing its entire visual appearance. At the very least, this requires that you override the *OnPaint* method. If your futuristic vision requires that the control be a different size than it would normally be, you'll also want to override *GetPreferredSize* and (perhaps) *OnResize*. With any luck, you might be able to leave the entire keyboard and mouse processing logic intact.

Windows Forms provides some assistance to programmers who implement control-drawing logic. Before you reinvent the button, you'll want to take a close look at *ControlPaint*. This class contains a number of static methods that perform various control-painting jobs and convert system colors into light and dark variations. System colors, pens, and brushes, by the way, are located in three classes in the *System.Drawing* namespace: *SystemColors*, *SystemPens*, and *SystemBrushes*. The *ProfessionalColors* and *ProfessionalColorTable* classes in *System.Windows.Forms* provide colors that are purportedly similar to those in Microsoft Office. *ProfessionalColors* is a collection of static properties, and *ProfessionalColorTable* contains identical instance properties that you can access after creating an instance of *ProfessionalColorTable*.

For the most part, I decided to forge my own drawing logic and choose my own colors in the *RoundButton* class. As the name suggests, *RoundButton* inherits from *Button* but creates a button that is round. This code also demonstrates how to make nonrectangular controls.

RoundButton.cs

```
//---------------------------------------
// RoundButton.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RoundButton : Button
{
    public RoundButton()
    {
        SetStyle(ControlStyles.UserPaint, true);
        SetStyle(ControlStyles.AllPaintingInWmPaint, true);
    }
    public override Size GetPreferredSize(Size szProposed)
    {
        // Base size on text string to be displayed.
        Graphics grfx = CreateGraphics();
        SizeF szf = grfx.MeasureString(Text, Font);
        int iRadius = (int)Math.Sqrt(Math.Pow(szf.Width / 2, 2) +
                                     Math.Pow(szf.Height / 2, 2));
        return new Size(2 * iRadius, 2 * iRadius);
    }
    protected override void OnResize(EventArgs args)
    {
        base.OnResize(args);

        // Circular region makes button non-rectangular.
        GraphicsPath path = new GraphicsPath();
        path.AddEllipse(ClientRectangle);
        Region = new Region(path);
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;
        grfx.SmoothingMode = SmoothingMode.AntiAlias;
        Rectangle rect = ClientRectangle;

        // Draw interior (darker if pressed).
        bool bPressed = Capture & ((MouseButtons & MouseButtons.Left) != 0) &
            ClientRectangle.Contains(PointToClient(MousePosition));

        GraphicsPath path = new GraphicsPath();
        path.AddEllipse(rect);
        PathGradientBrush pgbr = new PathGradientBrush(path);
        int k = bPressed ? 2 : 1;
        pgbr.CenterPoint = new PointF(k * (rect.Left + rect.Right) / 3,
                                      k * (rect.Top + rect.Bottom) / 3);
        pgbr.CenterColor = bPressed ? Color.Blue : Color.White;
        pgbr.SurroundColors = new Color[] { Color.SkyBlue };
        grfx.FillRectangle(pgbr, rect);
```

```
        // Display border (thicker for default button)
        Brush br = new LinearGradientBrush(rect,
                        Color.FromArgb(0, 0, 255), Color.FromArgb(0, 0, 128),
                        LinearGradientMode.ForwardDiagonal);
        Pen pn = new Pen(br, (IsDefault ? 4 : 2) * grfx.DpiX / 72);
        grfx.DrawEllipse(pn, rect);

        // Draw the text centered in the rectangle (grayed if disabled).
        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
        br = Enabled ? SystemBrushes.WindowText : SystemBrushes.GrayText;
        grfx.DrawString(Text, Font, br, rect, strfmt);

        // Draw dotted line around text if button has input focus.
        if (Focused)
        {
            SizeF szf = grfx.MeasureString(Text, Font, PointF.Empty,
                                           StringFormat.GenericTypographic);
            pn = new Pen(ForeColor);
            pn.DashStyle = DashStyle.Dash;
            grfx.DrawRectangle(pn,
                rect.Left + rect.Width / 2 - szf.Width / 2,
                rect.Top + rect.Height / 2 - szf.Height / 2,
                szf.Width, szf.Height);
        }
    }
}
```

The constructor sets two *ControlStyles* flags. For *Button*, these two flags happen to be *true* by default, but that can't be assumed for all controls. Setting the flags to *true* ensures that all painting logic is performed in the *OnPaint* method, so that overriding *OnPaint* is sufficient to replace all that drawing logic. (Alternatively, if the *AllPaintingInWmPaint* flag is *false*, you can paint the background of the control by overriding *OnPaintBackground*.)

GetPreferredSize is an important method in connection with autosizing. When *AutoSize* is set to *true*, a layout manager calls this method to obtain the size desired by the control. Whenever something happens that could affect the size—such as changes in the control's *Font* or *Text* properties—*GetPreferredSize* is called again to obtain an updated size. The *GetPreferredSize* method defined in *RoundButton* obtains the pixel dimensions of its *Text* property by calling *MeasureString*. It then calculates the distance from the center of that rectangle to a corner. This value is the desired radius of the round button.

The *OnResize* method is called whenever the size of the control changes, whether by autosizing or by explicit resizing. The code in the round button's *OnResize* method is also responsible for making the control nonrectangular by setting the *Region* property defined by the *Control* class. You set the *Region* property to an object of type *Region*, which is a class defined in *System.Drawing*. A graphical region defines an irregular area as a series of discontinuous scan lines. After setting the *Region* property, the control still has a rectangular dimension, but any

part of the control lying outside the area defined by the region will be transparent—both visually and in regard to mouse activity.

If you work solely within the confines of the *Region* class, it is only possible to construct regions from Boolean combinations of multiple rectangles. A more general approach is to first construct a path using the *GraphicsPath* class. A path is a collection of lines and curves that might or might not be connected, and which might or might not enclose areas. (A full discussion of paths and regions can be found in Chapter 15 of my book *Programming Microsoft Windows with C#* [Microsoft Press, 2001].) The path that *RoundButton* creates is composed simply of an ellipse the size of the button's client area. This ellipse—or rather, the interior of an ellipse—is converted into a region, and then that region is set to the *Region* property of the button.

The only other method in *RoundButton* is *OnPaint*, and this method does *not* call the *OnPaint* method in the base class because it doesn't want that method to do anything. A call to *OnPaint* occurs whenever something about the button needs repainting.

The method begins by drawing the interior area of the button. I decided to use a *PathGradientBrush* to give the button a hemisphere-like appearance. (Brushes—gradient and otherwise—are discussed in Chapter 17 of *Programming Microsoft Windows with C#*.) However, the button should also provide visual feedback whenever it's "pressed" with the mouse. The code chooses a darker color if the *Capture* property is *true* (that is, mouse input is going to the control) and the left mouse button is down and the mouse pointer is over the control.

OnPaint next displays the border. This is a pen based on a *LinearGradientBrush*. A regular button generally draws a heavier border when the button is the default (that is, when it responds to the Enter key). *RoundButton* uses the *IsDefault* property to vary the width of the border.

OnPaint then displays the text. Once again, it's not quite as simple as it might at first seem. If the button is disabled, the text should appear in gray. The method chooses between the system brushes *SystemBrushes.WindowText* and *SystemBrushes.GrayText* for displaying the text. The *StringFormat* object helps position the text in the center of the button.

If the button has the input focus, a dashed line should appear around the text. That's the job of the final section of *OnPaint*. The method again calls *MeasureString* to obtain the width and height of the text string, and then constructs a rectangle from that to display the dashed line.

You'll notice that the calls to *MeasureString* in *GetPreferredSize* and *OnPaint* are a little different. In the latter method, an argument of *StringFormat.GenericTypographic* was passed to the method along with the text string and font. By default, *MeasureString* returns dimensions a little larger than the string. I thought that behavior was appropriate for determining the size of the button because it provides a little padding around the text. When I originally used the same *MeasureString* call to draw the dashed outline of the text, however, the left and right sides of the dashed rectangle were obscured by the border of the button. Passing *StringFormat.GenericTypographic* to *MeasureString* causes the dimensions to more closely approximate

the actual text size, thus making a snugger dashed rectangle. (See Chapter 9 of *Programming Microsoft Windows with C#* for a fuller explanation of *GenericTypographic*.)

Here's a little program to test out the buttons. The RoundButtonDemo project includes both RoundButton.cs and this file.

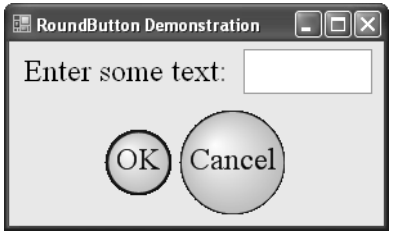
RoundButtonDemo.cs

```
//-----------------------------------------------
// RoundButtonDemo.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RoundButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new RoundButtonDemo());
    }
    public RoundButtonDemo()
    {
        Text = "RoundButton Demonstration";
        Font = new Font("Times New Roman", 18);
        AutoSize = true;
        AutoSizeMode = AutoSizeMode.GrowAndShrink;

        FlowLayoutPanel flow = new FlowLayoutPanel();
        flow.Parent = this;
        flow.AutoSize = true;
        flow.FlowDirection = FlowDirection.TopDown;

        FlowLayoutPanel flowTop = new FlowLayoutPanel();
        flowTop.Parent = flow;
        flowTop.AutoSize = true;
        flowTop.Anchor = AnchorStyles.None;

        Label lbl = new Label();
        lbl.Parent = flowTop;
        lbl.AutoSize = true;
        lbl.Text = "Enter some text:";
        lbl.Anchor = AnchorStyles.None;

        TextBox txtbox = new TextBox();
        txtbox.Parent = flowTop;
        txtbox.AutoSize = true;

        FlowLayoutPanel flowBottom = new FlowLayoutPanel();
        flowBottom.Parent = flow;
        flowBottom.AutoSize = true;
        flowBottom.Anchor = AnchorStyles.None;
```

```
        RoundButton btnOk = new RoundButton();
        btnOk.Parent = flowBottom;
        btnOk.Text = "OK";
        btnOk.Anchor = AnchorStyles.None;
        btnOk.DialogResult = DialogResult.OK;
        AcceptButton = btnOk;

        RoundButton btnCancel = new RoundButton();
        btnCancel.Parent = flowBottom;
        btnCancel.AutoSize = true;
        btnCancel.Text = "Cancel";
        btnCancel.Anchor = AnchorStyles.None;
        btnCancel.DialogResult = DialogResult.Cancel;
        CancelButton = btnCancel;

        btnOk.Size = btnCancel.Size;
    }
}
```

This program simulates a little dialog box with a label, a *TextBox*, and two *RoundButton* controls for OK and Cancel. Three *FlowPanel* controls perform dynamic layout.

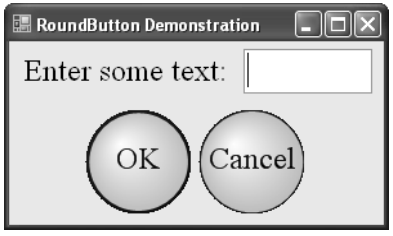
At first, I set the *AutoSize* property for both *RoundButton* controls to *true*, just as if they were normal buttons. The result looked very, very wrong:



This *had* to be fixed. I decided to set *AutoSize* to *true* for only the Cancel button. After both buttons are created, the OK button is set equal in size to Cancel:

```
btnOk.Size = btnCancel.Size;
```

The appearance is much better:



And, I guess I discovered why buttons aren't round by default.

Suppose a program wants to use the *RoundButton* control and also wants to add a little something to the visuals. It dutifully installs an event handler for *Paint* and . . . gets nothing. The problem is that *RoundButton* overrides *OnPaint*, but it doesn't call *OnPaint* in the base class because it doesn't want the base class doing anything. But that base class is responsible for firing the *Paint* event. Yet *RoundButton* can't fire the *Paint* event because only the class that defines the event can fire it.

The solution—if one is desired—is for *RoundButton* to define its own *Paint* event using the *new* keyword, and for the *OnPaint* method in *RoundButton* to fire that event.

Combining Existing Controls

One popular solution to creating new controls is by combining existing controls. To take advantage of much of the support of existing control logic, it is recommended that you derive such controls from the class *UserControl*. In particular, this class supports keyboard navigation among multiple controls.

As our first example, let's create a control not so different from a popular existing control. By whatever name they've gone under—spin buttons, up-down controls, the *NumericUpDown* class—I've always had a fundamental problem with controls that let you change numeric values using buttons sporting up and down arrows. On the one hand, the up-arrow might mean "higher," suggesting larger values, and the down-arrow might mean "lower." But when I visualize a list of numbers that I might be selecting from:

0

1

2

3

...

the up-arrow obviously means "toward 0" and the down-arrow means "toward infinity."

This ambiguity might be eliminated if the scroll bar were horizontal rather than vertical. Then, clearly, the left-arrow would signify smaller values and the right-arrow would move toward larger values, at least in cultures that read from left to right.

Because I am convinced that I am right about this issue and that the rest of the world need only see my example before universally adopting my design, I have decided to put my *NumericScan* control—as I've called it—into a dynamic-link library (DLL) named NumericScan.dll. I have also added some support so that this control will be more compliant with the Visual Studio designer.

Whether a collection of source code files becomes an executable (.exe) or a dynamic-link library (.dll) ultimately depends on the /target switch of the C# compiler. Set it to /target:exe

or /target:winexe to create an executable, or set it to /target:library to create a DLL. Within Visual Studio, you specify whether you want an executable or a DLL in the project properties.

It's easy to make a dynamic-link library project rather than a program project. If you're using the predefined project types in Visual Studio, select a template of either Class Library or Windows Control Library. If you're using the Empty Project option, create the project normally, and then display the project properties. Under Output Type, select Class Library. Selecting this option will create a dynamic-link library (.dll) rather than an executable file (.exe). Add at least one source code file to the project, and begin.

The only problem is that a DLL can't be directly executed, so you might get to the point where your code compiles fine but you don't know whether the control is actually working. You need an actual program to test it out.

For that reason, it's common when creating a DLL to also have a test program handy. And the easiest way to do that is to put the DLL project *and* the test program in the same Visual Studio solution. Here's how I did it.

In Visual Studio, I selected New Project from the File menu to invoke the New Project dialog box, as usual. I chose a project name of NumericScan, but I also checked the Create Directory For Solution check box. Checking this box results in the creation of a solution directory named NumericScan and, within that directory, a project directory also named NumericScan. Then select New Project from the File menu again. In the New Project dialog box, type a project name of TestProgram and make sure the Solution combo box is Add To Solution. Now the NumericScan solution has two projects named NumericScan and TestProgram.

In the Project Properties for the NumericScan project, make sure the Output Type is Class Library. In the Solution Explorer, right-click TestProgram and select Set As Startup Project. That means that when Visual Studio recompiles the entire solution, it will then launch TestProgram. There's still a little more overhead involved in creating a DLL and testing it, but I'll get to that.

The *NumericScan* control that I'll present here will comprise a *TextBox* control and two buttons. However, the buttons aren't quite normal. If you click one of the arrows on a normal *NumericUpDown* control and hold down the mouse button, you'll find that the buttons have a repeating action much like a scroll bar. It's also similar to the typematic action of the keyboard, so I decided to call a button that exhibited this characteristic a *ClickmaticButton* control. This is the first file in the NumericScan project:

ClickmaticButton.cs

```
//-------------------------------------------------
// ClickmaticButton.cs (c) 2005 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
namespace Petzold.ProgrammingWindowsForms
{
    class ClickmaticButton : Button
    {
        Timer tmr = new Timer();

        int iDelay = 250 * (1 + SystemInformation.KeyboardDelay);
        int iSpeed = 405 - 12 * SystemInformation.KeyboardSpeed;

        protected override void OnMouseDown(MouseEventArgs args)
        {
            base.OnMouseDown(args);

            if ((args.Button & MouseButtons.Left) != 0)
            {
                tmr.Interval = iDelay;
                tmr.Tick += TimerOnTick;
                tmr.Start();
            }
        }
        void TimerOnTick(object objSrc, EventArgs args)
        {
            OnClick(EventArgs.Empty);
            tmr.Interval = iSpeed;
        }
        protected override void OnMouseMove(MouseEventArgs args)
        {
            base.OnMouseMove(args);
            tmr.Enabled = Capture & ClientRectangle.Contains(args.Location);
        }
        protected override void OnMouseUp(MouseEventArgs args)
        {
            base.OnMouseUp(args);
            tmr.Stop();
        }
    }
}
```

Notice that this class is defined in a namespace. If you're going to be making DLLs, it's important to define the classes with a namespace so that they don't clash with class names used in any program using the DLL. I've chosen this namespace in rough accordance with common practice: company name first, followed by product name.

Regardless of the namespace, this particular class will *not* be visible from outside the DLL because the class definition does not include the *public* keyword. We haven't been worrying much about making classes public. It's really only an issue when you're putting classes in a DLL.

The class is fairly straightforward: it inherits from *Button* and overrides the *OnMouseDown* method to detect mouse clicks. Calling the method in the base class causes the normal call to *OnClick*, which then fires the *Click* event. *ClickmaticButton* continues processing *OnMouse-*

Down by starting a timer. The timer event handler makes additional calls to *OnClick* for repeated *Click* events. If the mouse button is down when the mouse moves away from the button, the timer should temporarily stop. The timer should also stop when the mouse button is released.

I was stuck for a little while about the proper *Interval* settings for the timer. There should be an initial delay when the button is clicked before the "clickmatic" action kicks in. Thereafter, the time interval should be shorter. Fortunately, I discovered some new .NET Framework 2.0 additions to the *SystemInformation* class. *SystemInformation.KeyboardDelay* is defined as "The keyboard repeat-delay setting, from 0 (approximately 250- millisecond delay) through 3 (approximately 1- second delay)." *SystemInformation.KeyboardSpeed* is defined as, "The keyboard repeat-speed setting, from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second)." These values worked just fine.

The *ArrowButton* class inherits from *ClickmaticButton* and overrides the *OnPaint* method to draw arrows using the *ControlPaint.DrawScrollButton* method. The direction of the arrow is specified through a public property.

ArrowButton.cs

```
//-------------------------------------------
// ArrowButton.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsForms
{
    class ArrowButton : ClickmaticButton
    {
        ScrollButton scrbtn = ScrollButton.Right;

        public ArrowButton()
        {
            SetStyle(ControlStyles.Selectable, false);
        }
        public ScrollButton ScrollButton
        {
            set
            {
                scrbtn = value;
                Invalidate();
            }
            get { return scrbtn; }
        }
        protected override void OnPaint(PaintEventArgs args)
        {
            Graphics grfx = args.Graphics;
            ControlPaint.DrawScrollButton(grfx, ClientRectangle, scrbtn,
                !Enabled ? ButtonState.Inactive :
```

```
                    (Capture & ClientRectangle.Contains(
                                        PointToClient(MousePosition))) ?
                ButtonState.Pushed : ButtonState.Normal);
        }
        protected override void OnMouseCaptureChanged(EventArgs args)
        {
            base.OnMouseCaptureChanged(args);
            Invalidate();
        }
    }
}
```

The *ArrowButton* class has a constructor that sets the *ControlStyles.Selectable* flag to *false*. When used in the *NumericScan* control, the buttons should not be selectable, which means they should not be able to get the input focus. The input focus should remain in the edit field.

Here's the public class *NumericScan* that inherits from *UserControl* and builds a control from a *TextBox* and two *ArrowButton* controls.

NumericScan.cs

```
//-----------------------------------------
// NumericScan.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.ComponentModel;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;

[assembly: AssemblyTitle("NumericScan")]
[assembly: AssemblyDescription("NumericScan Control")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("www.charlespetzold.com")]
[assembly: AssemblyProduct("NumericScan")]
[assembly: AssemblyCopyright("(c) Charles Petzold, 2005")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyVersion("1.0.*")]

namespace Petzold.ProgrammingWindowsForms
{
    [DefaultEvent("ValueChanged")]
    public class NumericScan : UserControl
    {
        public event EventHandler ValueChanged;

        TextBox txtbox;
        ArrowButton btn1, btn2;

        // These private fields have corresponding public properties.
        int iDecimalPlaces = 0;
        decimal mValue = 0;
        decimal mIncrement = 1;
```

```
        decimal mMinimum = 0;
        decimal mMaximum = 100;

        public NumericScan()
        {
            txtbox = new TextBox();
            txtbox.Parent = this;
            txtbox.TextAlign = HorizontalAlignment.Right;
            txtbox.Text = ValueToText(mValue);
            txtbox.TextChanged += TextBoxOnTextChanged;
            txtbox.KeyDown += TextBoxOnKeyDown;

            btn1 = new ArrowButton();
            btn1.Parent = this;
            btn1.Text = "btn1";
            btn1.ScrollButton = ScrollButton.Left;
            btn1.Click += ButtonOnClick;

            btn2 = new ArrowButton();
            btn2.Parent = this;
            btn2.Text = "btn2";
            btn2.ScrollButton = ScrollButton.Right;
            btn2.Click += ButtonOnClick;

            Width = 4 * Font.Height;
            Height = txtbox.PreferredHeight +
                                SystemInformation.HorizontalScrollBarHeight;
        }
        string ValueToText(decimal mValue)
        {
            return mValue.ToString("F" + DecimalPlaces);
        }

        [Category("Data"), Description("Value displayed in the control")]
        public decimal Value
        {
            set
            {
                txtbox.Text = ValueToText(mValue = value);
            }
            get
            {
                return mValue;
            }
        }

        [Category("Data"),
        Description("The amount to increment or decrement on a button click")]
        public decimal Increment
        {
            set { mIncrement = value; }
            get { return mIncrement; }
        }
```

```
[Category("Data"), Description("Minimum allowed value")]
public decimal Minimum
{
    set
    {
        if ((mMinimum = value) > Value)
            Value = mMinimum;
    }
    get { return mMinimum; }
}

[Category("Data"), Description("Maximum allowed value")]
public decimal Maximum
{
    set
    {
        if ((mMaximum = value) < Value)
            Value = mMaximum;
    }
    get { return mMaximum; }
}

[Category("Data"), Description("Number of decimal places to display")]
public int DecimalPlaces
{
    set { iDecimalPlaces = value; }
    get { return iDecimalPlaces; }
}
public override Size GetPreferredSize(Size szProposed)
{
    return new Size(4 * Font.Height, txtbox.PreferredHeight +
                        SystemInformation.HorizontalScrollBarHeight);
}
protected override void OnResize(EventArgs args)
{
    base.OnResize(args);

    txtbox.Height = txtbox.PreferredHeight;
    txtbox.Width = Width;
    btn1.Location = new Point(0, txtbox.Height);
    btn2.Location = new Point(Width / 2, txtbox.Height);
    btn1.Size = btn2.Size = new Size(Width / 2, Height - txtbox.Height);
}
void TextBoxOnTextChanged(object objSrc, EventArgs args)
{
    if (txtbox.Text.Length == 0)
        return;

    try
    {
        mValue = Decimal.Parse(txtbox.Text);
    }
    catch
    {
    }
```

```
            txtbox.Text = ValueToText(mValue);
        }
        void TextBoxOnKeyDown(object objSrc, KeyEventArgs args)
        {
            switch (args.KeyCode)
            {
                case Keys.Enter:
                    OnValueChanged(EventArgs.Empty);
                    break;
            }
        }
        void ButtonOnClick(object objSrc, EventArgs args)
        {
            ArrowButton btn = objSrc as ArrowButton;
            decimal mNewValue = Value;

            if (btn == btn1)
                if ((mNewValue -= Increment) < Minimum)
                    return;

            if (btn == btn2)
                if ((mNewValue += Increment) > Maximum)
                    return;

            Value = mNewValue;
            OnValueChanged(EventArgs.Empty);
        }
        protected override void OnLeave(EventArgs args)
        {
            base.OnLeave(args);
            OnValueChanged(EventArgs.Empty);
        }
        protected virtual void OnValueChanged(EventArgs args)
        {
            Value = Math.Max(Minimum, Value);
            Value = Math.Min(Maximum, Value);
            Value = Decimal.Round(Value, DecimalPlaces);

            if (ValueChanged != null)
                ValueChanged(this, args);
        }
    }
}
```

Everything in square brackets in the file is an attribute. The attributes at the beginning of the source code file are those I discussed in Chapter 1. The others are defined in the *System.ComponentModel* namespace. Preceding each of the public properties are *Category* and *Description* attributes. These are used by the *PropertyGrid* control to group related properties and to give a little description of them. Immediately preceding the class definition is a *DefaultEvent* attribute. Visual Studio uses this attribute in the designer to determine what event to use when you double-click a control to set up an event handler.

Notice also that the class is defined as *public*, so it will be visible from outside the DLL. The first member defined in the class is the public event *ValueChanged*.

NumericScan assembles one *TextBox* control and two *ArrowButton* controls on its surface. As in *NumericUpDown*, the control exposes public properties named *Value*, *Minimum*, *Maximum*, *Increment*, and *DecimalPlaces*. It's possible to implement more consistency checking than what I show here. (The .NET Framework *NumericUpDown* control throws some exceptions if a program sets *Value* outside the range of *Minimum* and *Maximum*, for example.) But what you should try to avoid are consistency checks that fail if a program sets properties in a particular order. For example, the default *Minimum* and *Maximum* properties are 0 and 100. A program might reset these two properties like this:

```
numscan.Minimum = 200;
numscan.Maximum = 300;
```

If the control threw an exception whenever *Minimum* is less than *Maximum*, the first statement would fail, and it really shouldn't.

Much of the remainder of the class is devoted to handling events from the *TextBox* and *ArrowButton* controls. Whenever the text changes, the *TextBoxOnTextChanged* event handler determines whether it's still a number. If not, it changes it back to its previous value. This event handler does not try to enforce minimum and maximum boundaries. (Neither does the *NumericUpDown* control. You can type whatever number you want in the control.)

As with the *NumericUpDown* control, the *ValueChanged* event should be fired whenever the value is altered by the buttons, when the user presses the Enter key, or when the control loses input focus. This is when the minimum and maximum boundaries are imposed.

To determine when the Enter key is pressed, the control installs a *KeyDown* event handler for the *TextBox* control. At this time, I recognized the flaw in having the arrow buttons point to the left and right: the buttons should be mimicked by the left and right cursor keys, yet these are the same keys used to move within the *TextBox*. Perhaps the up and down arrows are the right way to implement a spin button after all!

At the very bottom of the class is the *OnValueChanged* method. It's defined as *virtual* so that programs wishing to derive from *NumericScan* can easily override the method. The method imposes the minimum and maximum boundaries, rounds it to the desired number of decimal places, and fires the *ValueChanged* event.

I mentioned earlier that the NumericScan solution has two projects: NumericScan, which creates the NumericScan.dll file, and TestProgram, which creates TestProgram.exe from the following source code file.

TestProgram.cs

```
//-----------------------------------------
// TestProgram.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using Petzold.ProgrammingWindowsForms;
using System;
using System.Drawing;
using System.Windows.Forms;

class TestProgram : Form
{
    Label lbl;
    NumericScan numscan1, numscan2;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new TestProgram());
    }
    public TestProgram()
    {
        Text = "Test Program";

        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = this;
        pnl.Dock = DockStyle.Fill;

        numscan1 = new NumericScan();
        numscan1.Parent = pnl;
        numscan1.AutoSize = true;
        numscan1.ValueChanged += NumericScanOnValueChanged;

        numscan2 = new NumericScan();
        numscan2.Parent = pnl;
        numscan2.AutoSize = true;
        numscan2.ValueChanged += NumericScanOnValueChanged;

        lbl = new Label();
        lbl.Parent = pnl;
        lbl.AutoSize = true;
    }
    void NumericScanOnValueChanged(object objSrc, EventArgs args)
    {
        lbl.Text = "First: " + numscan1.Value + ", Second: " + numscan2.Value;
    }
}
```

The test program is simply two *NumericScan* controls with event handlers installed and a *Label* that displays the two values.

When defining the references for the TestProgram project, you need the normal System, System.Drawing, and System.Windows.Forms dynamic-link libraries, but you also need to

include NumericScan.dll. In the Add Reference dialog box, click the Projects tab and select NumericScan. As you'll note, TestProgram.cs also includes a *using* directive for the *Petzold.ProgrammingWindowsForms* namespace.

To add this control to Visual Studio's Toolbox, first right-click one of the tabs in the Toolbox and select Add Tab. You can name it More Controls, for example. Right-click that new tab and select Choose Items. The Choose Toolbox Items dialog box has a Browse button that lets you navigate to the NumericScan.dll file.

The following program gives the NumericScan control a more extensive workout. The first part of the program is a class that derives from *TableLayoutPanel* to display six *NumericScan* controls for setting the six fields of a .NET Framework matrix transform object. This class is quite similar to the MatrixElements program from the last chapter except that it's more generalized. The panel is given a public property named *Matrix* that allows setting the *NumericScan* controls and obtaining their values in the form of a *Matrix* object. The panel also defines a public event named *Change* that is fired whenever one of the *NumericScan* controls fires a *ValueChanged* event. Notice the first *using* directive for the namespace of the *NumericScan* control.

MatrixPanel.cs

```
//-------------------------------------------
// MatrixPanel.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using Petzold.ProgrammingWindowsForms;
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixPanel: TableLayoutPanel
{
    public event EventHandler Change;

    NumericScan[] numscan = new NumericScan[6];

    public Matrix Matrix
    {
        set
        {
            for (int i = 0; i < 6; i++)
                numscan[i].Value = (decimal)value.Elements[i];
        }
        get
        {
            return new Matrix((float)numscan[0].Value, (float)numscan[1].Value,
                              (float)numscan[2].Value, (float)numscan[3].Value,
                              (float)numscan[4].Value, (float)numscan[5].Value);
        }
    }
    public MatrixPanel()
    {
```

```
        AutoSize = true;
        Padding = new Padding(Font.Height);
        ColumnCount = 2;

        SuspendLayout();

        for (int i = 0; i < 6; i++)
        {
            Label lbl = new Label();
            lbl.Parent = this;
            lbl.AutoSize = true;
            lbl.Anchor = AnchorStyles.Left;
            lbl.Text = new string[] { "X Scale:", "Y Shear:", "X Shear:",
                                      "Y Scale:", "X Translate:",
                                      "Y Translate:" }[i];

            numscan[i] = new NumericScan();
            numscan[i].Parent = this;
            numscan[i].AutoSize = true;
            numscan[i].Anchor = AnchorStyles.Right;
            numscan[i].Minimum = -1000;
            numscan[i].Maximum = 1000;
            numscan[i].DecimalPlaces = 2;
            numscan[i].ValueChanged += NumericScanOnValueChanged;
        }
        ResumeLayout();

        Matrix = new Matrix();
    }
    void NumericScanOnValueChanged(object objSrc, EventArgs args)
    {
        OnChange(EventArgs.Empty);
    }
    protected virtual void OnChange(EventArgs args)
    {
        if (Change != null)
            Change(this, args);
    }
}
```

Because *TableLayoutPanel* derives from *Control*, and because this new class derives from *TableLayoutPanel*, does that make this new class a custom control? Sure! Any class you derive directly or indirectly from *Control* can be treated as a custom control. Adding properties and events to the control certainly makes it more customized and more useful, and actually reusing the control in other applications is the crowning achievement.

Here's another "custom control" of sorts. This is a derivative of *Panel* that displays its *Text* property after setting a matrix transform delivered to it in its public *Transform* property. Some matrix transforms raise exceptions when the *Graphics* object is set to the transform. For invalid matrix transforms, the *Graphics* object raises an exception, and the panel displays the exception message.

DisplayPanel.cs

```
//---------------------------------------------
// DisplayPanel.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class DisplayPanel : Panel
{
    Matrix matx = new Matrix();

    public DisplayPanel()
    {
        ResizeRedraw = true;
    }
    public Matrix Transform
    {
        set
        {
            matx = value;
            Invalidate();
        }
        get
        {
            return matx;
        }
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;
        Brush brsh = new SolidBrush(ForeColor);

        try
        {
            grfx.Transform = matx;
            grfx.DrawString(Text, Font, brsh, Point.Empty);
        }
        catch (Exception exc)
        {
            StringFormat strfmt = new StringFormat();
            strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
            grfx.DrawString(exc.Message, Font, brsh, ClientRectangle, strfmt);
        }
        brsh.Dispose();
    }
}
```

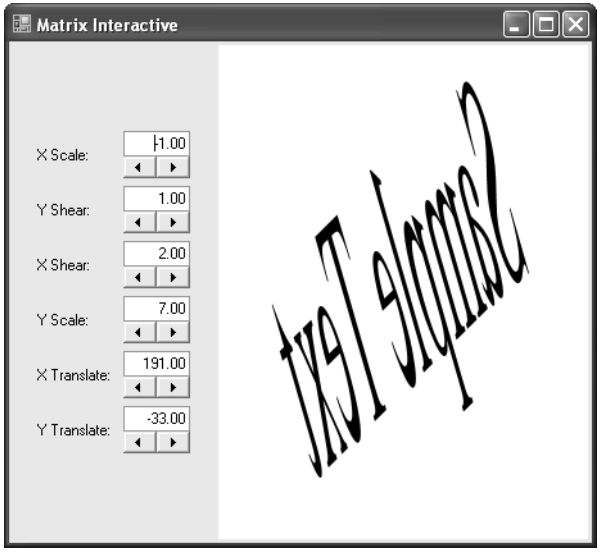
Normally, panels don't display their *Text* properties, so nobody bothers setting a panel's *Text* property. It is the responsibility of a program using this *DisplayPanel* control to assign a valid *Text* property and perhaps a different *Font* property.

MatrixInteractive is not the name of a new movie (let's hope not, anyway) but a project that includes MatrixPanel.cs, DisplayPanel.cs, and the next file.

MatrixInteractive.cs

```
//-------------------------------------------------
// MatrixInteractive.cs (c) 2005 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class MatrixInteractive : Form
{
    MatrixPanel matxpnl;
    DisplayPanel disppnl;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new MatrixInteractive());
    }
    public MatrixInteractive()
    {
        Text = "Matrix Interactive";

        TableLayoutPanel pnl = new TableLayoutPanel();
        pnl.Parent = this;
        pnl.Dock = DockStyle.Fill;
        pnl.ColumnCount = 2;

        matxpnl = new MatrixPanel();
        matxpnl.Parent = pnl;
        matxpnl.Anchor = AnchorStyles.Left | AnchorStyles.Right;
        matxpnl.Change += MatrixPanelOnChange;

        disppnl = new DisplayPanel();
        disppnl.Parent = pnl;
        disppnl.Dock = DockStyle.Fill;
        disppnl.BackColor = Color.White;
        disppnl.ForeColor = Color.Black;
        disppnl.Text = "Sample Text";
        disppnl.Font = new Font(FontFamily.GenericSerif, 24);

        Width = 3 * matxpnl.Width;
        Height = 3 * matxpnl.Height / 2;
    }
    void MatrixPanelOnChange(object objSrc, EventArgs args)
    {
        disppnl.Transform = matxpnl.Matrix;
    }
}
```

This program splits its client area into two parts using a *TableLayoutPanel*. At the left, centered vertically, is the *MatrixPanel*. At the right is a *DisplayPanel*, which is given a white background color, black foreground color, a *Text* property of "Sample Text," and a 24-point font. Whenever the *MatrixPanel* fires a *Change* event, this class obtains the matrix transform from the *MatrixPanel* and sets it to the *DisplayPanel*. Here's a sample screen shot:



The Sheer Pleasure of Autoscroll

The *Form* class and various *Panel* classes have an interesting feature that's not used much because it's not quite acceptable as a general-purpose user interface technique. This feature is called "autoscroll," and you enable it simply by setting the *AutoScroll* property of these classes to *true*. Then, if the *Form* or *Panel* is not large enough to display all its child controls, scrollbars magically appear and the missing controls can be scrolled into view. Autoscroll is implemented in *ScrollableControl* and available in every control that derives from *ScrollableControl*. Not every control that has scrollbars derives from *ScrollableControl*, however. *TextBox*, *ListBox*, and *ScrollBar* do *not* derive from *ScrollableControl*, but *Form* and *Panel* do.

I don't think autoscroll is the best approach to fitting a lot of controls in a tiny dialog box. However, it can certainly make some custom controls a whole lot easier. For example, consider a control that displays an indeterminate number of thumbnails of image files. If you just put these thumbnails on a panel and enable autoscrolling, all the scrolling logic is handled for you.

The next custom control I'll show is called *ImageScan*, and it displays 1-inch-square thumbnails of all the image files in a particular disk directory in a single scrollable row. *ImageScan* itself derives from *FlowLayoutControl*, and the thumbnails are implemented using the *PictureBox* control.

Once I started putting this control together, however, I ran into a basic problem. I wanted to navigate through the thumbnails using the Tab or arrow keys. The impediment was the *PictureBox* control itself, which is obstinately nonselectable. It cannot receive input focus, and (consequently) it cannot provide any feedback that it has input focus. The first step was to make a selectable picture box control and give it a simple keyboard interface.

SelectablePictureBox.cs

```
//-----------------------------------------------------
// SelectablePictureBox.cs (c) 2005 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SelectablePictureBox : PictureBox
{
    public SelectablePictureBox()
    {
        SetStyle(ControlStyles.Selectable, true);
        TabStop = true;
    }
    protected override void OnMouseDown(MouseEventArgs args)
    {
        base.OnMouseDown(args);
        Focus();
    }
    protected override void OnKeyPress(KeyPressEventArgs args)
    {
        if (args.KeyChar == '\r')
            OnClick(EventArgs.Empty);
        else
            base.OnKeyPress(args);
    }
    protected override void OnEnter(EventArgs args)
    {
        base.OnEnter(args);
        Invalidate();
    }
    protected override void OnLeave(EventArgs e)
    {
        base.OnLeave(e);
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        base.OnPaint(args);

        if (Focused)
        {
```

```
            Graphics grfx = args.Graphics;
            grfx.DrawRectangle(new Pen(Brushes.Black, grfx.DpiX / 12),
                ClientRectangle);
        }
    }
}
```

The constructor sets the *ControlStyles.Selectable* flag and the *TabStop* property to *true*. An override of the *OnPaint* method lets the *OnPaint* method of the base class do its stuff, and then prints a black border around the control if the control has input focus. Several other *On* methods also needed some enhancements. When the control is clicked, the control gives itself input focus. When the Enter key is pressed, the control simulates an *OnClick* call. The *OnEnter* and *OnLeave* methods are called when the control gains input focus and loses it. My overrides simply invalidate the control to generate a call to *OnPaint* and ensure the control is painted correctly.

Here's the *ImageScan* control that inherits from *FlowLayoutPanel*. Notice the constructor that sets *WrapContents* to *false* and *AutoScroll* to *true*.

ImageScan.cs

```
//---------------------------------------
// ImageScan.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.ComponentModel;    // for AsyncCompletedEventArgs
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImageScan : FlowLayoutPanel
{
    Size szImage;
    string strImageLocation;
    ToolTip tips = new ToolTip();

    public ImageScan()
    {
        FlowDirection = FlowDirection.LeftToRight;
        WrapContents = false;
        AutoScroll = true;

        // Create Size object of one square inch.
        Graphics grfx = CreateGraphics();
        szImage = new Size((int)grfx.DpiX, (int)grfx.DpiY); // 1" square
        grfx.Dispose();

        Height = szImage.Height + Font.Height +
                        SystemInformation.HorizontalScrollBarHeight;
    }
    public string Directory
    {
```

```
        set
        {
            Controls.Clear();
            tips.RemoveAll();

            string[] astrFiles = System.IO.Directory.GetFiles(value, "*.*");

            foreach (string strFile in astrFiles)
            {
                PictureBox picbox = new SelectablePictureBox();
                picbox.Parent = this;
                picbox.Size = szImage;
                picbox.SizeMode = PictureBoxSizeMode.Zoom;
                picbox.Click += PictureBoxOnClick;
                picbox.LoadCompleted += PictureBoxOnLoadCompleted;
                picbox.LoadAsync(strFile);
            }
        }
    }
    public string SelectedImageFile
    {
        get
        {
            return strImageLocation;
        }
    }
    void PictureBoxOnClick(object objSrc, EventArgs args)
    {
        PictureBox picbox = objSrc as PictureBox;
        strImageLocation = picbox.ImageLocation;

        OnClick(args);
    }

    // Don't generate Click events when user clicks the panel.
    protected override void OnMouseDown(MouseEventArgs args)
    {
    }
    void PictureBoxOnLoadCompleted(object objSrc, AsyncCompletedEventArgs args)
    {
        PictureBox picbox = objSrc as PictureBox;

        if (args.Error == null)
            tips.SetToolTip(picbox, Path.GetFileName(picbox.ImageLocation));
        else
            Controls.Remove(picbox);
    }
}
```

The control basically hosts a collection of *SelectablePictureBox* controls, each displaying a particular image in a directory. Also included is a *ToolTip* component for displaying file names as ToolTips. *ImageScan* implements a public property named *Directory* that specifies a disk directory. When this property is set, the control first clears out its existing collection of child

controls and all the ToolTips. It then obtains all the files in the directory and creates a *SelectablePictureBox* object for each.

But wait: this control is supposed to display only *image* files, but a *SelectablePictureBox* is created for every file, whether it's an image file or not. Notice two things about these *SelectablePictureBox* controls: First, an event handler is installed for the *LoadCompleted* event, and second, the *LoadAsync* method of *PictureBox* is called for the file. This method loads the file in a secondary thread and fires the *LoadCompleted* event when it is completed. The event is accompanied by an object of type *AsyncCompletedEventArgs*. If the file loaded fine, the *Error* property of that object is *null*. If the file did not load correctly—and in this program that will happen a lot because the directory might contain a lot of nonimage files—the control removes that file from its child control collection.

And here's a program that uses *ImageScan*. The ImageDirectory project includes SelectablePictureBox.cs, ImageScan.cs, and this file.

ImageDirectory.cs

```
//-----------------------------------------------
// ImageDirectory.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageDirectory: Form
{
    PictureBox picbox;
    ImageScan imgscan;
    Label lblDirectory;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new ImageDirectory());
    }
    public ImageDirectory()
    {
        Text = "Image Directory";

        picbox = new PictureBox();
        picbox.Parent = this;
        picbox.Dock = DockStyle.Fill;
        picbox.SizeMode = PictureBoxSizeMode.Zoom;

        imgscan = new ImageScan();
        imgscan.Parent = this;
        imgscan.Dock = DockStyle.Top;
        imgscan.Click += ImageScanOnClick;
```

```
        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = this;
        pnl.AutoSize = true;
        pnl.Dock = DockStyle.Top;

        Button btn = new Button();
        btn.Parent = pnl;
        btn.AutoSize = true;
        btn.Anchor = AnchorStyles.Left;
        btn.Text = "Directory...";
        btn.Click += ButtonOnClick;

        lblDirectory = new Label();
        lblDirectory.Parent = pnl;
        lblDirectory.AutoSize = true;
        lblDirectory.Anchor = AnchorStyles.Right;

        // Initialize.
        imgscan.Directory = lblDirectory.Text =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
    }
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        FolderBrowserDialog dlg = new FolderBrowserDialog();
        dlg.SelectedPath = lblDirectory.Text;
        dlg.ShowNewFolderButton = false;

        if (dlg.ShowDialog() == DialogResult.OK)
            imgscan.Directory = lblDirectory.Text = dlg.SelectedPath;
    }
    void ImageScanOnClick(object objSrc, EventArgs args)
    {
        picbox.ImageLocation = imgscan.SelectedImageFile;
    }
}
```

The program really just organizes controls into panels and hooks them together. A *Button* invokes a *FolderBrowserDialog* to let the user select a directory. A *Label* displays that directory. The *ImageScan* control shows the images in that directory, and the selected image occupies the remainder of the space in the program's client area. Here's a screen shot showing a selected image from the WINDOWS directory:

When selecting a directory containing a lot of nonimage files, the autoscroll facility of *ImageScan* kicks in almost immediately in accordance with all the *SelectablePictureBox* controls added as children. As certain of these files are determined to be nonimage files, the scroll bar registers those changes and might disappear altogether! It's a little peculiar, but it seems to work.

Controls from Scratch

Some controls, of course, are so unlike any of the standard controls that they require significant amounts of custom drawing and processing of user input. For such controls, you'll probably just derive from *Control* and start coding. (Better get an early start.) As examples, I present two case studies: a ruler like the one that appears above documents in word-processing programs such as Windows WordPad and a simple color-selection grid.

An Interactive Ruler

Let's take a look at Windows WordPad. WordPad is built around the control known in the Windows API as the *RichEdit* control, and which is accessible in a Windows Forms program through the *RichTextBox* class. The *RichEdit* control saves documents in the Rich Text Format (RTF), which allows paragraph and character formatting to be applied to different parts of the document.

The ruler displayed by WordPad allows the user to interactively set a left indentation, right indentation, first-line indentation, and tabs that apply to the currently selected paragraph. (Alternatively, you can change these same formatting items in the Paragraph and Tabs dialog boxes accessible from the menu in WordPad Format.)

Now let's examine the ruler in a more sophisticated word-processing program. We'll pick one at random, say, Microsoft Office Word. Ostensibly, the ruler in Microsoft Word looks a lot like the ruler in WordPad, but it also lets you change the left and right *margins*. These margins

apply to the entire document and can alternatively be altered through the Page Setup dialog box accessible through the File menu. The margins really only make sense in connection with a particular paper width. The margins denote areas on the right and left of the page—generally an inch or so wide—where text does not appear. The width of the text on the page is the page width minus the left and right margins. In Word, left and right paragraph indentation is normally 0, meaning that the paragraphs occupy the full width between the left and right margins. Indentation can be made larger for narrower paragraphs or made to be even less than zero to occupy space in the margin. Itis also possible to set a different indentation for the first line of a paragraph, so that the line is either indented relative to the rest of the paragraph or "hanging" to the left beyond the paragraph.

By convention, the left margin is measured from the left edge of the page, the left indentation is relative to the left margin, and the first-line indentation is relative to the left indentation: positive for a normal indentation, negative for a hanging effect, and 0 for a flush first line. The right margin is measured from the right edge of the page, and the right indentation is measured from the right margin. Both numbers are normally non-negative.

My original intention was to create a *DocumentRuler* control (as I called it) similar to the one in Microsoft Word that lets you set margins and indentations. However, the *RichTextBox* control has a much weaker concept of margins than Word. The only thing in *RichTextBox* that comes close is a property named *RightMargin* that lets a program specify a pixel width of the text displayed by the control. (In conventional terminology, this property actually specifies something more akin to a page width minus the left and right margins.) By default, this property is 0, which means that the width of the text displayed by the control is governed by the width of the control itself. For awhile, I toyed with the idea of imposing traditional concepts of page widths and margins on the *RichTextBox*, but I eventually decided to go for a simpler approach. My ruler is similar to the WordPad implementation, but it also allows the text width to be changed.

Although *DocumentRuler* doesn't specifically *require* that it be used in conjunction with a *RichTextBox* control, it doesn't go beyond the capabilities of *RichTextBox* in any way. It would need some enhancements to be used with a real word-processing application such as Word.

Given a page width of 8 1/2 inches and margins of 1 1/4 inches, I figured that an initial default value of 6 inches for the *RightMargin* property seemed about right. It's customary to think of margins, indents, and tabs in terms of inches, particularly if there's a ruler sitting at the top of the document. But that's not the way *RichTextBox* does business. The *RightMargin* property and all the indentation properties in *RichTextBox* are instead specified in pixels.

I decided that the programming interface to the ruler must be in terms of inches. Thus, the ruler would have a set of properties with names such as *LeftIndent*, *RightIndent*, and *FirstLineIndent*, and these would be *float* values in inches. This decision resulted in a lot of conversion between inches and pixels throughout both the ruler control and the program using the control. Of course, because the ruler is displayed on the screen, the conversion between

inches and pixels is based on the user's screen resolution, which is available from the *DpiX* and *DpiY* properties of the *Graphics* object. (Even more conversions go on behind the scenes. Rich Text Format maintains measurements in terms of "twips," which are 1/20 of a printer's point and equal to 1/1440 of an inch.)

Although the inch markings on *DocumentRuler* are based on the screen resolution, I was less successful in making other aspects of the ruler as device independent as I would have preferred. In particular, the little moveable markers showing the indentations are too tiny and precisely constructed to react well to sizes calculated based on screen resolution.

Inches and pixels weren't the only conversions required in this job. The three properties of *RichTextBox* connected with indentations are called *SelectionIndent*, *SelectionRightIndent*, and *SelectionHangingIndent*. Unfortunately, the first two of these properties work a little differently from the familiar conventions of paragraph formatting. *SelectionIndent* is the indentation of the first line of the paragraph from the left side of the text box. *SelectionHangingIndent* is the indentation of the remainder of the paragraph relative to the first line. A paragraph indented 100 pixels from the left side of the text box with the first line indented another 50 pixels would have a *SelectionIndent* of 150 and a *SelectionHangingIndent* of −50. I decided that *DocumentRuler* would implement indentations in the more familiar way. It is the responsibility of a program that creates both a *RichTextBox* and a *DocumentRuler* to convert between the two.

We are now ready to start looking at some code. The *DocumentRuler* control implements just one event that I called simply *Change*. The event is triggered whenever the user changes a margin, indentation, or tab on the ruler. I wanted the program using the ruler to know what had changed (for example, the left indentation), which implies that the event must deliver that information. The event could not be based on the standard *EventHandler* delegate but required a custom delegate. First, an enumeration is defined with fields for each of the items settable through the ruler.

RulerProperty.cs

```
//-----------------------------------------------
// RulerProperty.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
public enum RulerProperty
{
    TextWidth,
    LeftIndent,
    RightIndent,
    FirstLineIndent,
    Tabs
}
```

The *Change* event will be accompanied by an object of type *RulerEventArgs*. This class derives from *EventArgs* but implements an additional property of type *RulerProperty*.

RulerEventArgs.cs

```
//---------------------------------------------
// RulerEventArgs.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;

public class RulerEventArgs : EventArgs
{
    RulerProperty rlrprop;

    public RulerEventArgs(RulerProperty rlrprop)
    {
        this.rlrprop = rlrprop;
    }
    public RulerProperty RulerChange
    {
        get { return rlrprop; }
        set { rlrprop = value; }
    }
}
```

The class also defines a constructor to create a new *RulerEventArgs* object by specifying a member of *RulerProperty*.

Whenever you define a new class that will be used to deliver information to an event handler, you must also define a new delegate for the event handler. The code is simple.

RulerEventHandler.cs

```
//-------------------------------------------------
// RulerEventHandler.cs (c) 2005 by Charles Petzold
//-------------------------------------------------
public delegate void RulerEventHandler(object objSrc, RulerEventArgs args);
```

The ruler contains four little types of graphical pictures that denote the left indent, right indent, first-line indent, and tabs. These little items must be drawn, of course, but the ruler must also detect when the user clicks one of them with the mouse. I decided to implement these objects in separate classes, but all are based on one abstract class called *RulerSlider*.

RulerSlider.cs

```
//-----------------------------------------
// RulerSlider.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

abstract class RulerSlider
{
```

```
// Private fields.
RulerProperty rlrprop;
float fValue;
int x, y;
Bitmap bm;
Region rgn;

// Public properties.
public RulerProperty RulerProperty
{
    get { return rlrprop; }
    set { rlrprop = value; }
}
public float Value
{
    get { return fValue; }
    set { fValue = value; }
}
public int X
{
    get { return x; }
    set { x = value; }
}
public virtual Rectangle Rectangle
{
    get
    {
        return new Rectangle(X - bm.Width / 2, Y, bm.Width, bm.Height);
    }
}

// Protected property.
protected int Y
{
    get { return y; }
    set { y = value; }
}

// Public methods.
public virtual void Draw(Graphics grfx)
{
    grfx.DrawImage(bm, X - bm.Width / 2, Y);
}
public virtual bool HitTest(Point pt)
{
    return rgn.IsVisible(pt.X - X + bm.Width / 2, pt.Y - Y);
}
protected void CreateBitmap(int cx, int cy, Point[] apt)
{
    bm = new Bitmap(cx, cy);

    GraphicsPath path = new GraphicsPath();
    path.AddLines(apt);
    rgn = new Region(path);
```

```
        Graphics grfx = Graphics.FromImage(bm);
        grfx.FillPolygon(Brushes.LightGray, apt);
        grfx.Clip = rgn;

        Shading(grfx, Pens.White, 1, apt);
        Shading(grfx, Pens.Gray, -1, apt);

        grfx.ResetClip();
        grfx.DrawPolygon(Pens.Black, apt);
        grfx.Dispose();
    }
    void Shading(Graphics grfx, Pen pn, int iOffset, Point[] apt)
    {
        grfx.TranslateTransform(iOffset, 0);
        grfx.DrawPolygon(pn, apt);
        grfx.TranslateTransform(-iOffset, iOffset);
        grfx.DrawPolygon(pn, apt);
        grfx.TranslateTransform(0, -iOffset);
    }
}
```

As you'll see shortly, the various classes that derive from *RulerSlider* themselves set the *RulerProperty* property to an appropriate member of the *RulerProperty* enumeration. They also set the *Y* property to a fixed position of the slider relative to the top of the control. The *X* property varies as the user moves the slider from one place to another.

I also decided that these sliders should maintain a floating-point *Value* property that saves the current value in inches. In theory, *Value* and *X* are convertible between each other, but I wanted to keep the floating-point value intact to avoid rounding differences during conversions. I wanted to avoid situations in which a property such as *LeftIndent* might be set to one value but then return a slightly different value.

The *Draw* method of *RulerSlider* draws the slider at the *X* and *Y* coordinates. In *RulerSlider*, *Draw* is implemented to simply draw a bitmap—the same bitmap created in the protected *CreateBitmap* method. The *CreateBitmap* method requires a width and a height of the bitmap and an array of *Point* objects. This array defines a closed area on the bitmap that is filled with various colors and shading. In the process, a *Region* object is also created to be used in the *HitTest* method. If a mouse coordinate is passed to *HitTest*, the method returns *true* if the mouse is over the object.

The read-only *Rectangle* property returns a rectangle that encompasses the bitmap as displayed on the ruler. This is useful to invalidate areas of the ruler when the user is moving the slider.

Here's the *RightIndent* class that derives from *RulerSlider*.

RightIndent.cs

```
//-----------------------------------------
// RightIndent.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RightIndent : RulerSlider
{
    public RightIndent()
    {
        RulerProperty = RulerProperty.RightIndent;
        Y = 9;
        CreateBitmap(9, 8, new Point[]
        {
            new Point(0, 7), new Point(0, 4), new Point(4, 0),
            new Point(8, 4), new Point(8, 7), new Point(0, 7)
        });
    }
}
```

The class simply sets two properties of *RulerSlider* and calls *CreateBitmap* to create an image that looks like a little house. The *LeftIndent* is similar except the image is a bit more elaborate.

LeftIndent.cs

```
//----------------------------------------
// LeftIndent.cs (c) 2005 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class LeftIndent : RulerSlider
{
    public LeftIndent()
    {
        RulerProperty = RulerProperty.LeftIndent;
        Y = 9;
        CreateBitmap(9, 14, new Point[]
        {
            new Point(0, 7), new Point(0, 4), new Point(4, 0), new Point(8, 4),
            new Point(8, 7), new Point(0, 7), new Point(0, 13), new Point(8, 13),
            new Point(8, 7), new Point(0, 7)
        });
    }
}
```

The *FirstLineIndent* is similar to *RightIndent* except that it's upside-down and positioned near the top of the control.

FirstLineIndent.cs

```
//-----------------------------------------------
// FirstLineIndent.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FirstLineIndent : RulerSlider
{
    public FirstLineIndent()
    {
        RulerProperty = RulerProperty.FirstLineIndent;
        Y = 1;
        CreateBitmap(9, 8, new Point[]
        {
            new Point(0, 0), new Point(8, 0), new Point(8, 3),
            new Point(4, 7), new Point(0, 3), new Point(0, 0)
        });
    }
}
```

The *Tab* character (an L shape) is better suited for a simple line. The class overrides the *Draw* and *HitTest* methods, as well as the *Rectangle* property.

Tab.cs

```
//-----------------------------------
// Tab.cs (c) 2005 by Charles Petzold
//-----------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Tab : RulerSlider
{
    public Tab()
    {
        RulerProperty = RulerProperty.Tabs;
        Y = 9;
    }
    public override void Draw(Graphics grfx)
    {
        Pen pn = new Pen(Color.Black, 2);

        grfx.DrawLine(pn, X, Y, X, Y + 4);
        grfx.DrawLine(pn, X, Y + 4, X + 4, Y + 4);
    }
    public override bool HitTest(Point pt)
    {
        return pt.X >= X - 1 && pt.X <= X + 1 && pt.Y >= Y - 1 && pt.Y <= Y + 6;
    }
    public override Rectangle Rectangle
```

```
    {
        get
        {
            return new Rectangle(X - 1, Y - 1, 6, 6);
        }
    }
}
```

The final class that inherits from *RulerSlider* is *TextWidth*. Changing the text width is a little different than moving little markers. The drawing logic needs to be handled in the main *OnPaint* method of the control, so the class doesn't do much.

TextWidth.cs

```
//---------------------------------------
// TextWidth.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TextWidth : RulerSlider
{
    public TextWidth()
    {
        RulerProperty = RulerProperty.TextWidth;
    }
    public override void Draw(Graphics grfx)
    {
    }
    public override bool HitTest(Point pt)
    {
        return (pt.X >= X - 2) && (pt.X <= X + 2);
    }
    public override Rectangle Rectangle
    {
        get { return Rectangle.Empty; }
    }
}
```

And now we're ready for the *RulerDocument* class itself. So that it won't be too overwhelming, I divided the class into two source code files using the *partial* keyword. The first installment has the constructor and all the properties.

DocumentRuler1.cs

```
//-----------------------------------------------
// DocumentRuler1.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Collections.Generic;
using System.Drawing;
```

```
using System.Windows.Forms;

public partial class DocumentRuler : Control
{
    // Private fields.
    int iLeftMargin;
    float fDpi;
    Control ctrlDocument;

    // RulerSlider objects.
    LeftIndent rsLeftIndent = new LeftIndent();
    RightIndent rsRightIndent = new RightIndent();
    FirstLineIndent rsFirstIndent = new FirstLineIndent();
    TextWidth rsTextWidth = new TextWidth();
    List<RulerSlider> rsCollection = new List<RulerSlider>();

    // Constructor.
    public DocumentRuler()
    {
        Dock = DockStyle.Top;
        ResizeRedraw = true;
        TabStop = false;
        Height = 23;
        Font = new Font(Font.Name, 14, GraphicsUnit.Pixel);

        Graphics grfx = CreateGraphics();
        fDpi = grfx.DpiX;
        grfx.Dispose();

        rsCollection.Add(rsLeftIndent);
        rsCollection.Add(rsRightIndent);
        rsCollection.Add(rsFirstIndent);
        rsCollection.Add(rsTextWidth);
    }

    // Public properties.
    public float TextWidth
    {
        get { return rsTextWidth.Value; }
        set
        {
            rsTextWidth.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float LeftIndent
    {
        get { return rsLeftIndent.Value; }
        set
        {
            rsLeftIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float RightIndent
```

```
    {
        get { return rsRightIndent.Value; }
        set
        {
            rsRightIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float FirstLineIndent
    {
        get { return rsFirstIndent.Value; }
        set
        {
            rsFirstIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float[] Tabs
    {
        get
        {
            List<float> fTabs = new List<float>();

            foreach (RulerSlider rs in rsCollection)
                if (rs is Tab)
                    fTabs.Add(rs.Value);

            // RichTextBox wants tabs in numeric order
            float[] afTabs = fTabs.ToArray();
            Array.Sort(afTabs);
            return afTabs;
        }
        set
        {
            // First, delete tabs that aren't in value array.
            List<Tab> rsTabsDelete = new List<Tab>();

            foreach (RulerSlider rs in rsCollection)
                if (rs is Tab && (Array.IndexOf(value, rs.Value) == -1))
                    rsTabsDelete.Add(rs as Tab);

            foreach (Tab tab in rsTabsDelete)
            {
                rsCollection.Remove(tab);
                Invalidate(tab.Rectangle);
            }

            // Second, add tabs that aren't in rsCollection.
            foreach (float fTab in value)
            {
                bool bAdd = true;

                foreach (RulerSlider rs in rsCollection)
                    if (rs is Tab && rs.Value == fTab)
                        bAdd = false;
```

```
            if (bAdd)
            {
                Tab tab = new Tab();
                tab.Value = fTab;
                tab.X = LeftMargin + InchesToPixels(fTab);
                rsCollection.Add(tab);
                Invalidate(tab.Rectangle);
            }
        }
    }
}
public int LeftMargin
{
    get { return iLeftMargin; }
    set
    {
        iLeftMargin = value;
        CalculateDisplayOffsets();
    }
}

// For displaying a line when sliders are slid.
public Control DocumentControl
{
    get { return ctrlDocument; }
    set { ctrlDocument = value; }
}

// These two methods calculate X values for the four types of sliders
//  (excluding tabs).  If the X values changes, invalidate the rectangle
//  at the previous position and the new position.
void CalculateDisplayOffsets()
{
    CalculateDisplayOffsets2(rsTextWidth, LeftMargin +
                    InchesToPixels(rsTextWidth.Value));
    CalculateDisplayOffsets2(rsLeftIndent, LeftMargin +
                    InchesToPixels(rsLeftIndent.Value));
    CalculateDisplayOffsets2(rsRightIndent, LeftMargin +
                    InchesToPixels(TextWidth - rsRightIndent.Value));
    CalculateDisplayOffsets2(rsFirstIndent, LeftMargin +
                    InchesToPixels(LeftIndent + rsFirstIndent.Value));
}
void CalculateDisplayOffsets2(RulerSlider rs, int xNew)
{
    if (rs.X != xNew)
    {
        Invalidate(rs.Rectangle);
        rs.X = xNew;
        Invalidate(rs.Rectangle);
    }
}
float PixelsToInches(int i)
{
    return i / fDpi;
}
```

```
    int InchesToPixels(float f)
    {
        return (int)Math.Round(f * fDpi);
    }
}
```

Fields define the various *RulerSlider* objects so that *rsLeftIndent* is an object of type *LeftIndent*, *rsRightIndent* is an object of type *RightIndent*, and so forth. Also created is a *List* collection of type *RulerSlider* named *rsCollection*. The constructor makes all these individual *RulerSlider* objects members of the collection. The collection must also eventually include multiple objects of type *Tab*.

Next in the file is a series of public properties named *TextWidth*, *LeftIndent*, and so forth. These provide access to the objects *rsTextWidth*, *rsLeftIndent*, and so on. For each of these properties, after the *Value* property is set for the *RulerSlider* object, a *CalculateDisplayOffsets* calculates the *X* properties.

The property named *Tabs* is more extensive than all the rest simply because it's plural rather than singular. Each paragraph may have its own series of tabs. In general, as the ruler displays information for different parts of the document, all the tabs in the *rsCollection* must be deleted and new ones added. My original code did literally that, and the constant erasing and redrawing during typing in the *RichTextBox* caused the markers to flicker. The current code doesn't delete and re-create tabs at the same location.

You'll also notice a public property named *LeftMargin* of type *int* that provides access to the *iLeftMargin* field, and you'll also notice that just about every calculation in the program uses this *LeftMargin* property. This property might better be called *Kludge*. Visually, the ruler needs a little space to the left of the 0 position to properly display the left indent slider. So, in the program coming up that puts the *RichTextBox* and *DocumentRuler* controls together, I did what WordPad does: I set the *ShowSelectionMargin* property of *RichTextBox* to *true*. This property opens up a little space to the left so that the user can select whole lines of text. But how much space? It seemed to be about 10 pixels, so that's what the program coming up sets as the *LeftMargin* property of *DocumentRuler*. I wasn't very happy, and I fear this is a decision that will haunt me for the rest of my days, but I wasn't sure of a better approach.

The final public property in *DocumentRuler* is named *DocumentControl*. This is set to the word-processing control associated with the *DocumentRuler*. The only reason *DocumentRuler* needs to know this is to draw a vertical line down the control when the user is sliding one of the sliders. All other interaction between the *DocumentRuler* and the *RichTextBox* takes place external to the two controls.

Although the first part of the *DocumentRuler* class is devoted to input and output for *programs* using the control, the second part of the *DocumentRuler* class is devoted to input and output for the *user*. It includes overrides of the *OnPaint*, *OnMouseDown*, *OnMouseMove*, and *OnMouseUp* methods.

DocumentRuler2.cs

```
//---------------------------------------------
// DocumentRuler2.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

public partial class DocumentRuler : Control
{
    // Public event.
    public event RulerEventHandler Change;

    // Private fields used during mouse dragging.
    RulerSlider rsDragging;
    Point ptDown;
    int xOriginal;
    int xLineOverTextBox;

    // OnPaint method handles virtually all drawing.
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;

        Rectangle rect = new Rectangle(LeftMargin, 0,
                               rsTextWidth.X - LeftMargin, Height - 4);
        grfx.FillRectangle(Brushes.White, rect);
        ControlPaint.DrawBorder3D(grfx, rect);

        for (int i = 1; i < 8 * PixelsToInches(Width); i++)
        {
            int x = LeftMargin + InchesToPixels(i / 8f);

            if (i % 8 == 0)
            {
                StringFormat strfmt = new StringFormat();
                strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
                grfx.DrawString((i / 8).ToString(), Font,
                                Brushes.Black, x, 9, strfmt);
            }
            else if (i % 4 == 0)
            {
                grfx.DrawLine(Pens.Black, x, 7, x, 10);
            }
            else
            {
                grfx.DrawLine(Pens.Black, x, 8, x, 9);
            }
        }

        // Display all sliders.
        foreach (RulerSlider rs in rsCollection)
            rs.Draw(grfx);
```

```
        return;
    }

    // OnMouseDown for moving sliders and creating tabs.
    protected override void OnMouseDown(MouseEventArgs args)
    {
        // Ignore if it's not the left button.
        if ((args.Button & MouseButtons.Left) == 0)
            return;

        // Loop through existing sliders looking for positive hit test.
        foreach (RulerSlider rs in rsCollection)
            if (rs.HitTest(args.Location))
            {
                rsDragging = rs;
                ptDown = args.Location;
                xOriginal = rsDragging.X;

                if (rsDragging is TextWidth)
                    Cursor.Current = Cursors.SizeWE;

                DrawReversibleLine(xLineOverTextBox = args.X);
                return;
            }
        // If no hit, create a new tab.
        rsDragging = new Tab();
        rsCollection.Add(rsDragging);
        ptDown = args.Location;
        xOriginal = rsDragging.X = ptDown.X;

        Invalidate(rsDragging.Rectangle);
        DrawReversibleLine(xLineOverTextBox = args.X);
        return;
    }

    // OnMouseMove for moving sliders.
    protected override void OnMouseMove(MouseEventArgs args)
    {
        if (!Capture)   // i.e., mouse button not down.
        {
            // If over TextWidth end, change cursor.
            if (!rsRightIndent.HitTest(args.Location) &&
                 rsTextWidth.HitTest(args.Location))
                    Cursor.Current = Cursors.SizeWE;
            return;
        }

        // If rsDragging not null, we're in a drag operation.
        if (rsDragging != null)
        {
            if (rsDragging is TextWidth)
                Cursor.Current = Cursors.SizeWE;

            int xNow = xOriginal - ptDown.X + args.X;
```

```
            // Don't let the sliders go out of bounds!
            if (rsDragging is Tab && (xNow < LeftMargin || xNow > rsTextWidth.X))
                return;

            if ((rsDragging == rsLeftIndent || rsDragging == rsFirstIndent) &&
                    (xNow < LeftMargin || xNow > rsRightIndent.X))
                return;

            if (rsDragging == rsRightIndent && (xNow > rsTextWidth.X ||
                    xNow < rsLeftIndent.X || xNow < rsFirstIndent.X))
                return;

            if (rsDragging == rsTextWidth && xNow < rsRightIndent.X)
                return;

            if (rsDragging == rsTextWidth)
            {
                Invalidate(new Rectangle(Math.Min(rsDragging.X, xOriginal) - 1, 0,
                                Math.Abs(rsDragging.X - xOriginal) + 2, Height));
                rsDragging.X = xNow;
            }
            else
            {
                // Update the slider X property and invalidate old and new.
                Invalidate(rsDragging.Rectangle);
                rsDragging.X = xNow;
                Invalidate(rsDragging.Rectangle);
            }
            // Move line over text box.
            DrawReversibleLine(xLineOverTextBox);
            DrawReversibleLine(xLineOverTextBox = args.X);
        }
    }

    // OnMouseUp is new position of slider.
    protected override void OnMouseUp(MouseEventArgs args)
    {
        if (rsDragging != null)
        {
            // Calculate new Value properties and trigger the event.
            if (rsDragging == rsLeftIndent || rsDragging == rsFirstIndent)
            {
                rsLeftIndent.Value = PixelsToInches(rsLeftIndent.X - LeftMargin);
                rsFirstIndent.Value =
                        PixelsToInches(rsFirstIndent.X - rsLeftIndent.X);
                OnChange(new RulerEventArgs(rsDragging.RulerProperty));
            }
            else if (rsDragging == rsRightIndent || rsDragging == rsTextWidth)
            {
                rsTextWidth.Value = PixelsToInches(rsTextWidth.X - LeftMargin);
                rsRightIndent.Value =
                        PixelsToInches(rsTextWidth.X - rsRightIndent.X);
                OnChange(new RulerEventArgs(rsTextWidth.RulerProperty));
                OnChange(new RulerEventArgs(rsRightIndent.RulerProperty));
            }
```

```
            else if (rsDragging is Tab)
            {
                rsDragging.Value = PixelsToInches(rsDragging.X - LeftMargin);
                OnChange(new RulerEventArgs(rsDragging.RulerProperty));
            }
            // Cease drag operation.
            rsDragging = null;
            DrawReversibleLine(xLineOverTextBox);
        }
    }

    // Draw line down text box in screen coordinates.
    void DrawReversibleLine(int x)
    {
        if (ctrlDocument != null)
        {
            Point pt1 = ctrlDocument.PointToScreen(new Point(x, 0));
            Point pt2 = ctrlDocument.PointToScreen(
                                new Point(x, ctrlDocument.Height));
            ControlPaint.DrawReversibleLine(pt1, pt2, ctrlDocument.BackColor);
        }
    }

    // OnChange method triggers Change event.
    protected virtual void OnChange(RulerEventArgs args)
    {
        if (Change != null)
            Change(this, args);
    }
}
```

The *OnPaint* method is simpler than it might be because of the two lines of code near the bottom of the method:

```
foreach (RulerSlider rs in rsCollection)
    rs.Draw(grfx);
```

The *RulerSlider* objects draw themselves, so the *OnPaint* method doesn't need to bother.

Similarly, the *RulerSlider* objects perform their own hit-testing, and the *OnMouseDown* method uses those methods to determine whether the user is clicking an existing slider. If not, the user wants a new tab. In either case, *OnMouseDown* sets several private fields—*rsDragging* (the particular slider being dragged), *ptDown* (the original point where the mouse button was clicked), and *xOriginal* (the original position of the slider)—to assist in moving the sliders. The *OnMouseMove* method is mostly devoted to making sure the sliders don't get moved to illegal positions. The *SelectionRightIndent* property of *RichTextBox* cannot be negative, for example, which means that the right margin cannot be moved to the left of the right indent. *OnMouseUp* completes the dragging operation.

And now, here's the class that lets us actually look at the ruler and try it out. The *RichTextWithRuler* class inherits from *Form* and creates a *RichTextBox* control and a *DocumentRuler* control.

The RichTextWithRuler project includes this file and all other files starting with RulerProperty.cs.

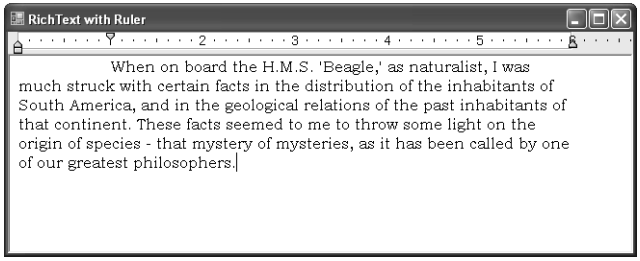
RichTextWithRuler.cs

```
//-----------------------------------------------
// RichTextWithRuler.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RichTextWithRuler : Form
{
    DocumentRuler ruler;
    RichTextBox txtbox;
    float fDpi;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new RichTextWithRuler());
    }
    public RichTextWithRuler()
    {
        Text = "RichText with Ruler";

        Graphics grfx = CreateGraphics();
        fDpi = grfx.DpiX;
        grfx.Dispose();

        txtbox = new RichTextBox();
        txtbox.Parent = this;
        txtbox.AcceptsTab = true;
        txtbox.Dock = DockStyle.Fill;
        txtbox.RightMargin = InchesToPixels(6);
        txtbox.ShowSelectionMargin = true;
        txtbox.SelectionChanged += TextBoxOnSelectionChanged;

        ruler = new DocumentRuler();
        ruler.Parent = this;
        ruler.LeftMargin = 10;
        ruler.TextWidth = PixelsToInches(txtbox.RightMargin);
        ruler.DocumentControl = txtbox;
        ruler.Change += RulerOnChange;

        // Initialize the ruler with text box values.
        TextBoxOnSelectionChanged(txtbox, EventArgs.Empty);
    }
    void TextBoxOnSelectionChanged(object objSrc, EventArgs args)
    {
        ruler.LeftIndent = PixelsToInches(txtbox.SelectionIndent +
                                          txtbox.SelectionHangingIndent);
        ruler.RightIndent = PixelsToInches(txtbox.SelectionRightIndent);
```

```
        ruler.FirstLineIndent = PixelsToInches(-txtbox.SelectionHangingIndent);

        float[] fTabs = new float[txtbox.SelectionTabs.Length];

        for (int i = 0; i < txtbox.SelectionTabs.Length; i++)
            fTabs[i] = PixelsToInches(txtbox.SelectionTabs[i]);

        ruler.Tabs = fTabs;
    }
    void RulerOnChange(object objSrc, RulerEventArgs args)
    {
        switch (args.RulerChange)
        {
            case RulerProperty.TextWidth:
                txtbox.RightMargin = InchesToPixels(ruler.TextWidth);
                break;

            case RulerProperty.LeftIndent:
            case RulerProperty.FirstLineIndent:
                txtbox.SelectionIndent = InchesToPixels(ruler.LeftIndent +
                                                        ruler.FirstLineIndent);
                txtbox.SelectionHangingIndent =
                    InchesToPixels(-ruler.FirstLineIndent);
                break;

            case RulerProperty.RightIndent:
                txtbox.SelectionRightIndent = InchesToPixels(ruler.RightIndent);
                break;

            case RulerProperty.Tabs:
                int[] iTabs = new int[ruler.Tabs.Length];

                for (int i = 0; i < ruler.Tabs.Length; i++)
                    iTabs[i] = InchesToPixels(ruler.Tabs[i]);

                txtbox.SelectionTabs = iTabs;
                break;
        }
    }
    float PixelsToInches(int i)
    {
        return i / fDpi;
    }
    int InchesToPixels(float f)
    {
        return (int)Math.Round(f * fDpi);
    }
}
```

As I promised, this file again contains a lot of converting between inches and pixels. The program must provide the interface between *RichTextBox* and *DocumentRuler*, and it does this largely in event handlers for the *SelectionChanged* event of the text box and the *Change* event of

the ruler. Here's the program showing left and right indents of zero and a first-line indent of one inch:



Color Selection

There are six million color-selection controls in .NET city, and this is one of them. *ColorGrid* displays an array of 40 colors in a grid. You can click one of the colors (of course), but you can also move through the color grid using the keyboard arrow keys. This control has the most extensive keyboard processing in this chapter.

ColorGrid.cs

```
//-----------------------------------------
// ColorGrid.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorGrid : Control
{
    // Number of colors horizontally and vertically.
    const int xNum = 8;
    const int yNum = 5;

    // The colors.
    Color[,] aclr = new Color[yNum, xNum]
    {
        { Color.Black, Color.Brown, Color.DarkGreen, Color.MidnightBlue,
            Color.Navy, Color.DarkBlue, Color.Indigo, Color.DimGray },

        { Color.DarkRed, Color.OrangeRed, Color.Olive, Color.Green,
            Color.Teal, Color.Blue, Color.SlateGray, Color.Gray },

        { Color.Red, Color.Orange, Color.YellowGreen, Color.SeaGreen,
            Color.Aqua, Color.LightBlue, Color.Violet, Color.DarkGray },

        { Color.Pink, Color.Gold, Color.Yellow, Color.Lime,
            Color.Turquoise, Color.SkyBlue, Color.Plum, Color.LightGray },

        { Color.LightPink, Color.Tan, Color.LightYellow, Color.LightGreen,
            Color.LightCyan, Color.LightSkyBlue, Color.Lavender, Color.White }
    };
```

```
// Selected color as a private field.
Color clrSelected = Color.Black;

// Rectangles for displaying colors and borders.
Rectangle rectTotal, rectGray, rectBorder, rectColor;

// The coordinate currently highlighted by keyboard or mouse.
int xHighlight = -1;
int yHighlight = -1;

// Constructor.
public ColorGrid()
{
    AutoSize = true;

    // Obtain the resolution of the screen
    Graphics grfx = CreateGraphics();
    int xDpi = (int)grfx.DpiX;
    int yDpi = (int)grfx.DpiY;
    grfx.Dispose();

    // Calculate rectangles for color displays
    rectTotal = new Rectangle(0, 0, xDpi / 5, yDpi / 5);
    rectGray = Rectangle.Inflate(rectTotal, -xDpi / 72, -yDpi / 72);
    rectBorder = Rectangle.Inflate(rectGray, -xDpi / 48, -yDpi / 48);
    rectColor = Rectangle.Inflate(rectBorder, -xDpi / 72, -yDpi / 72);
}

// SelectedColor property -- access to clrSelected field
public Color SelectedColor
{
    get
    {
        return clrSelected;
    }
    set
    {
        clrSelected = value;
        Invalidate();
    }
}

// Required for autosizing.
public override Size GetPreferredSize(Size sz)
{
    return new Size(xNum * rectTotal.Width, yNum * rectTotal.Height);
}

// Draw all colors in the grid.
protected override void OnPaint(PaintEventArgs args)
{
    Graphics grfx = args.Graphics;

    for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
```

```
            DrawColor(grfx, x, y, false);
    }

    // Draw an individual color. (grfx can be null)
    void DrawColor(Graphics grfx, int x, int y, bool bHighlight)
    {
        bool bDisposeGraphics = false;

        if (x < 0 || y < 0 || x >= xNum || y >= yNum)
            return;

        if (grfx == null)
        {
            grfx = CreateGraphics();
            bDisposeGraphics = true;
        }

        // Determine if the color is currently selected.
        bool bSelect = aclr[y, x].ToArgb() == SelectedColor.ToArgb();

        Brush br = (bHighlight | bSelect) ? SystemBrushes.HotTrack :
                        SystemBrushes.Menu;

        // Start draw rectangles.
        Rectangle rect = rectTotal;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(br, rect);

        if (bHighlight || bSelect)
        {
            br = bHighlight ? SystemBrushes.ControlDark :
                              SystemBrushes.ControlLight;
            rect = rectGray;
            rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
            grfx.FillRectangle(br, rect);
        }

        rect = rectBorder;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(SystemBrushes.ControlDark, rect);

        rect = rectColor;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(new SolidBrush(aclr[y, x]), rect);

        if (bDisposeGraphics)
            grfx.Dispose();
    }

    // Methods for mouse movement and clicks.
    protected override void OnMouseEnter(EventArgs args)
    {
        xHighlight = -1;
        yHighlight = -1;
    }
```

```
protected override void OnMouseMove(MouseEventArgs args)
{
    int x = args.X / rectTotal.Width;
    int y = args.Y / rectTotal.Height;

    if (x != xHighlight || y != yHighlight)
    {
        DrawColor(null, xHighlight, yHighlight, false);
        DrawColor(null, x, y, true);

        xHighlight = x;
        yHighlight = y;
    }
}
protected override void OnMouseLeave(EventArgs args)
{
    DrawColor(null, xHighlight, yHighlight, false);

    xHighlight = -1;
    yHighlight = -1;
}
protected override void OnMouseDown(MouseEventArgs args)
{
    int x = args.X / rectTotal.Width;
    int y = args.Y / rectTotal.Height;
    SelectedColor = aclr[y, x];
    base.OnMouseDown(args);         // Generates Click event.
    Focus();
}

// Methods for keyboard interface.
protected override void OnEnter(EventArgs args)
{
    if (xHighlight < 0 || yHighlight < 0)
        for (yHighlight = 0; yHighlight < yNum; yHighlight++)
        {
            for (xHighlight = 0; xHighlight < xNum; xHighlight++)
            {
                if (aclr[yHighlight, xHighlight].ToArgb() ==
                                SelectedColor.ToArgb())
                    break;
            }
            if (xHighlight < xNum)
                break;
        }

    if (xHighlight == xNum && yHighlight == yNum)
        xHighlight = yHighlight = 0;

    DrawColor(null, xHighlight, yHighlight, true);
}
protected override void OnLeave(EventArgs args)
{
    DrawColor(null, xHighlight, yHighlight, false);
    xHighlight = yHighlight = -1;
```

```
    }
    protected override bool IsInputKey(Keys keyData)
    {
        return keyData == Keys.Home || keyData == Keys.End ||
               keyData == Keys.Up || keyData == Keys.Down ||
               keyData == Keys.Left || keyData == Keys.Right;
    }
    protected override void OnKeyDown(KeyEventArgs args)
    {
        DrawColor(null, xHighlight, yHighlight, false);
        int x = xHighlight, y = yHighlight;

        switch (args.KeyCode)
        {
            case Keys.Home:
                x = y = 0;
                break;

            case Keys.End:
                x = xNum - 1;
                y = yNum - 1;
                break;

            case Keys.Right:
                if (++x == xNum)
                {
                    x = 0;
                    if (++y == yNum)
                    {
                        Parent.GetNextControl(this, true).Focus();
                    }
                }
                break;

            case Keys.Left:
                if (--x == -1)
                {
                    x = xNum - 1;
                    if (--y == -1)
                    {
                        Parent.GetNextControl(this, false).Focus();
                    }
                }
                break;

            case Keys.Down:
                if (++y == yNum)
                {
                    y = 0;
                    if (++x == xNum)
                    {
                        Parent.GetNextControl(this, true).Focus();
                    }
                }
                break;
```

```
            case Keys.Up:
                if (--y == -1)
                {
                    y = 0;
                    if (--x == -1)
                    {
                        Parent.GetNextControl(this, false).Focus();
                    }
                }
                break;

            case Keys.Enter:
            case Keys.Space:
                SelectedColor = aclr[y, x];
                OnClick(EventArgs.Empty);
                break;

            default:
                base.OnKeyDown(args);
                return;
        }
        DrawColor(null, x, y, true);

        xHighlight = x;
        yHighlight = y;
    }
}
```

The control defines one new public property named *SelectedColor*. A program using the control is notified when the *SelectedColor* has changed by the normal *Click* event.

The constructor defines four rectangles used for displaying each of the colors in the grid. The dimensions of these rectangles are based entirely upon the vertical and horizontal resolution of the display. *ColorGrid* is one control (at least) that won't need recoding when people begin using 300-DPI displays. The *OnPaint* method simply calls *DrawColor* for each of the 40 colors in the grid; the *DrawColor* method does the real work. The drawing logic distinguishes between the *selected* color, which is the color available from the *SelectedColor* property, and the *highlighted* color. The highlighted control changes when the mouse pointer passes over the control, or when the control has input focus and the arrow keys are pressed. A color is converted from highlighted to selected when the mouse button is depressed or when the Enter key or spacebar is pressed.

Changing the highlight based on the mouse pointer is the responsibility of the *OnMouseEnter*, *OnMouseMove*, and *OnMouseLeave* overrides. The *OnMouseDown* method sets a new selected color and calls the base method to generate a *Click* event.

The keyboard processing is more extensive. First, the control must determine when it gets and loses input focus. If the mouse pointer is not over the control, a color is highlighted only when the control has the input focus. *ColorGrid* uses the *OnEnter* and *OnLeave* methods for this job.

Another problem: many of the keys that *ColorGrid* wants—the arrow keys in particular—are used by the parent control to shift input focus among its children. *ColorGrid* must override the *IsInputKey* and return *true* for any key that it wants exclusive use of. These keys are then processed in the *OnKeyDown* method. The arrow keys move through the rows and columns of the grid until the color is reached at the upper-left or lower-right corner. At that point, input focus passes to the previous or next sibling control. Notice also the processing of the Enter and spacebar keys to change the selection.

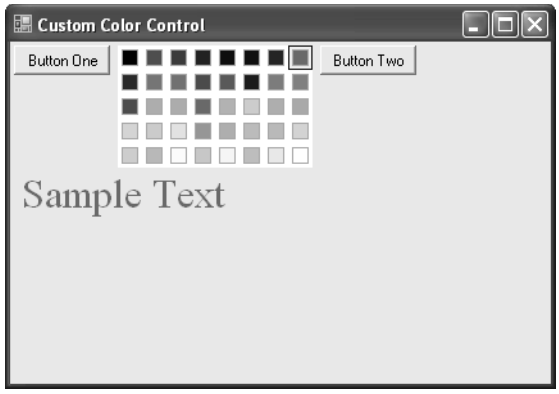
The ColorGridDemo project includes ColorGrid.cs and the next file.

ColorGridDemo.cs

```
//--------------------------------------------
// ColorGridDemo.cs (c) 2005 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorGridDemo : Form
{
    Label lbl;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new ColorGridDemo());
    }
    public ColorGridDemo()
    {
        Text = "Custom Color Control";
        AutoSize = true;

        TableLayoutPanel table = new TableLayoutPanel();
        table.Parent = this;
        table.AutoSize = true;
        table.ColumnCount = 3;

        Button btn = new Button();
        btn.Parent = table;
        btn.AutoSize = true;
        btn.Text = "Button One";

        ColorGrid clrgrid = new ColorGrid();
        clrgrid.Parent = table;
        clrgrid.Click += ColorGridOnClick;

        btn = new Button();
        btn.Parent = table;
        btn.AutoSize = true;
        btn.Text = "Button Two";
```

```
        lbl = new Label();
        lbl.Parent = table;
        lbl.AutoSize = true;
        lbl.Font = new Font("Times New Roman", 24);
        lbl.Text = "Sample Text";

        table.SetColumnSpan(lbl, 3);
        clrgrid.SelectedColor = lbl.ForeColor;
    }
    void ColorGridOnClick(object objSrc, EventArgs args)
    {
        ColorGrid clrgrid = (ColorGrid) objSrc;
        lbl.ForeColor = clrgrid.SelectedColor;
    }
}
```

The *ColorGridDemo* class creates a *ColorGrid* control, and it puts the control between two *Button* controls just to test the transfer of input focus. When a new color is selected, that color is used as the foreground color of a *Label* control:



If you think that color grid resembles something you've seen in the Format | Background menu in a Microsoft Office application, I must confess that was the source of my inspiration. Putting that control into a menu and toolbar will be one of the challenges facing us in the next chapter.