## Contents

<table>
<thead>
<tr>
<th>Section 1: Using Graphics, Pens, and Brushes</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting a Graphics Object</td>
<td>3</td>
</tr>
<tr>
<td>Using a Graphics Object</td>
<td>5</td>
</tr>
<tr>
<td>Creating Pens</td>
<td>8</td>
</tr>
<tr>
<td>Creating Brushes</td>
<td>11</td>
</tr>
<tr>
<td>Section 1 Wrap-up</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 2: Using Advanced Pens and Brushes</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Dash Patterns</td>
<td>15</td>
</tr>
<tr>
<td>Longitudinal Stripes</td>
<td>16</td>
</tr>
<tr>
<td>Custom End Caps</td>
<td>18</td>
</tr>
<tr>
<td>Linear Gradient Brushes</td>
<td>19</td>
</tr>
<tr>
<td>Path Gradient Brushes</td>
<td>21</td>
</tr>
<tr>
<td>Section 2 Wrap-up</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 3: Drawing Text</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drawing Simple Text</td>
<td>23</td>
</tr>
<tr>
<td>Using Layout Rectangles</td>
<td>24</td>
</tr>
<tr>
<td>Section 3 Wrap-up</td>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 4: Manipulating Images</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating and Loading Bitmaps</td>
<td>29</td>
</tr>
<tr>
<td>Manipulating Bitmaps</td>
<td>31</td>
</tr>
<tr>
<td>Saving Image Files</td>
<td>33</td>
</tr>
<tr>
<td>Section 4 Wrap-up</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 5: Using Transformations</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Transformations</td>
<td>35</td>
</tr>
<tr>
<td>World Coordinate Mapping</td>
<td>38</td>
</tr>
<tr>
<td>Section 5 Wrap-up</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 6: Printing</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using PrintPage</td>
<td>43</td>
</tr>
<tr>
<td>Using Other Event Handlers</td>
<td>46</td>
</tr>
<tr>
<td>Printing Transformations</td>
<td>48</td>
</tr>
<tr>
<td>Section 6 Wrap-up</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 7: Using WPF Graphics</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decorative Controls</td>
<td>51</td>
</tr>
<tr>
<td>Shape Controls</td>
<td>52</td>
</tr>
<tr>
<td>Brushes</td>
<td>56</td>
</tr>
<tr>
<td>Section 7 Wrap-up</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 8: Building FlowDocuments</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockUIContainer</td>
<td>61</td>
</tr>
<tr>
<td>List</td>
<td>61</td>
</tr>
<tr>
<td>Paragraph</td>
<td>61</td>
</tr>
<tr>
<td>Section</td>
<td>61</td>
</tr>
<tr>
<td>Table</td>
<td>61</td>
</tr>
<tr>
<td>Section 8 Wrap-up</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conclusion</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>About Rod Stephens</td>
<td>65</td>
</tr>
</tbody>
</table>
C# Graphics Programming

When my editor Bob Elliott asked me if I wanted to write a Wrox Blox about .NET graphics, I felt like a kid in a candy shop! Where should I start? Should I gorge myself on image processing? Or pace myself and work slowly through an assortment of chocolate-covered pens and brushes? Perhaps the metaphor is a bit strained, but .NET provides so many graphics tools that it’s hard to decide where to start and where to stop. There are so many interesting tools and techniques that there’s no way to cover everything in depth in a single Wrox Blox.

To make this as useful as possible to you, I’ll cover an assortment of the topics that are most useful in Visual Studio applications. I’ll provide enough information to get you going, and I’ll even describe some of the more advanced tools such as PathGradientBrushes and custom line end caps. Since I won’t be able to cover every tool and technique in as much depth as I would like, at some point you’ll need to strike out on your own and do some additional research on the Web. If you get stuck on an advanced topic, or if you have questions or comments about the material in this Wrox Blox, e-mail me at RodStephens@vb-helper.com, and I’ll try to give you some pointers.

Section 1 starts the discussion by describing the basic building blocks of graphics programming in C#: the Graphics, Pen, and Brush classes. You can use these classes to draw and fill lines, curves, ellipses — just about everything except images and text.

Section 2 describes advanced Pen and Brush objects that let you produce more advanced effects such as lines with custom end caps and areas filled with color gradients that follow a path.

Section 3 finishes the discussion of fundamental graphical techniques by explaining how to draw text.

Section 4 tells how to draw and manipulate bitmapped images. It shows how to modify or draw images and save the results in a variety of formats such as BMP, GIF, and JPG files.
Section 5 explains an extremely powerful feature provided by .NET graphics: transformations. With a little practice, you can use transformations to build extremely complex graphics relatively simply using the coordinate systems that are easiest for you.

Section 6 shows how to add printouts and print previews to C# applications. Many books don’t cover this topic very well, but printing plays an important role in many applications.

Section 7 describes the new Graphics objects that you can use in Windows Presentation Foundation (WPF) applications. By using these objects, you can create graphics declaratively at design time in addition to procedural run time.

Section 8 describes a particularly useful object provided by WPF (Windows Presentation Foundation) graphics: FlowDocument. A FlowDocument object can contain text, images, graphical objects, controls, and other visible objects. It then automatically flows those objects across whatever space is available. The result is a newsletter-like document that is extremely easy to build and view.

It’s a lot of material to cover in not so many pages, so grab your favorite caffeinated beverage and let’s get started!

**Section 1: Using Graphics, Pens, and Brushes**

Whenever you draw in C#, you draw on a Graphics object. You can think of this object as representing the canvas or piece of paper on which you will draw. There are several occasions when you may want to draw (when a form refreshes, to update an image, when generating a printout), but they all use the same kind of Graphics object. That’s great news because it means you only need to learn how to draw in one way and you can use the same code to draw on just about anything.

The Pen class determines the characteristics of line-like graphics. This includes straight lines as well as the edges of ellipses, rectangles, arcs, Bezier curves, and other drawn lines. The Pen object determines the line’s color, thickness, dash style, lengthwise style (e.g., the line may consist of thin stripes), end caps (the shape of the line’s end points), dash caps (the shape of the ends of dashes), and join style.

The Brush class determines the characteristics of filled areas. This includes such items as filled ellipses, rectangles, arcs, and other closed curves. The Brush class also determines how text is filled. The Brush object determines the filled area’s color, hatch pattern, color gradient, or fill pattern.

For example, the following code draws on a Graphics object named gr. First the code uses a solid, light blue brush to fill an ellipse that has its upper-left corner at (10, 10), width 200, and height 100. (All units are pixels.) It then draws the same ellipse with a 1-pixel-wide, solid, stock red pen.

```csharp
gr.FillEllipse(Brushes.LightBlue, 10, 10, 200, 100)
gr.DrawEllipse(Pens.Red, 10, 10, 200, 100)
```
The following sections describe simple uses of the Graphics, Pen, and Brush classes. Section 2, “Using Advanced Pens and Brushes,” explains how to use more complicated Pen and Brush objects to produce more sophisticated images.

**Getting a Graphics Object**

The most common methods for obtaining a Graphics object on which to draw are by:

- Calling a form or other control’s CreateGraphics method.
- Calling the Graphics class’s FromImage method.
- Using the e.Graphics parameter in a Paint event handler.
- Using the e.Graphics parameter in a PrintDocument object’s PrintPage event handler.

The following sections describe each of these methods in some detail.

**Using CreateGraphics**

The CreateGraphics method returns a Graphics object that you can use to draw immediately on a form or control. Unfortunately, whatever you draw is only safe until that object is refreshed. The UseCreateGraphics example program uses the following code to draw an ellipse on a form:

```csharp
Graphics gr = this.CreateGraphics();
gr.FillEllipse(Brushes.Yellow, 10, 30, 200, 150);
gr.DrawEllipse(Pens.Orange, 10, 30, 200, 150);
```

If you cover part of the form with another form and then expose that part again, the part of the ellipse that was covered disappears. Similarly, if you minimize the form and restore it, the entire ellipse disappears. To produce graphics that don’t go away like this, you should use some other drawing method such as the Paint event handler described shortly.

**Using Graphics.FromImage**

The Graphics class provides a static FromImage method that lets you create a Graphics object associated with an image, typically a bitmap. You can then use that object to draw on the bitmap.

The EllipseOnBitmap example program uses the following code. It creates a bitmap and then gets a Graphics object associated with it. It uses that object to draw an ellipse and then displays the bitmap in a PictureBox.

```csharp
// Create the Bitmap.
Bitmap bm = new Bitmap(200, 100);

// Get a Graphics object associated with the Bitmap.
Graphics gr = Graphics.FromImage(bm);

// Draw on the Bitmap.
gr.FillEllipse(Brushes.Pink, 10, 10, 180, 80);
```
gr.DrawEllipse(Pens.Black, 10, 10, 180, 80);

// Display the Bitmap in the PictureBox.
picEllipse.Image = bm;

The PictureBox automatically re-displays its image whenever it is refreshed. That means the image
doesn’t disappear when the control is refreshed as it would be if you used the CreateGraphics method
described above. Similarly, you could draw on a bitmap and assign it to a form’s BackgroundImage
property to tile the form with the image.

Building a bitmap by using the FromImage method is usually the best method for drawing permanent
images.

**Using e.Graphics in Paint Event Handlers**

When a form or other control needs to be refreshed, it raises a Paint event. The event handler takes a
parameter e that has a Graphics property. The event handler’s code can use that property to draw on
the control.

The PaintDrawX example program uses the following code to draw an X on its form:

```csharp
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.Clear(this.BackColor);
    e.Graphics.DrawLine(Pens.Blue, 0, 0,
        this.ClientSize.Width, this.ClientSize.Height);
    e.Graphics.DrawLine(Pens.Blue,
        this.ClientSize.Width, 0, 0, this.ClientSize.Height);
}
```

This approach is simple and works well if the drawing appears quickly. If it takes several seconds to
produce the image, the user may see annoying flickering and re-drawing when the form is refreshed or
resized. In that situation, it would be better to draw the image onto a bitmap and display the result in a
PictureBox as described in the above section.

Using a Paint event handler leads to two fairly confusing situations. First, if the drawing appears at different
sizes based on the form size, as is the case in the above code, you need to handle form resizing specially.

When a form resizes, it raises its Paint event to re-draw any new areas that have been exposed. That
much makes sense, but the e.Graphics object passed into the Paint event handler only represents the
new areas and not any areas that were already visible. For example, suppose a form’s drawing area is
100 pixels wide and you then widen the form so that the area is 200 pixels wide. In that case, the
e.Graphics parameter only represents the new area on the right half of the form. Any drawing that the
code performs on the left part of the form is ignored. The result can be strange, with pieces of older
images still on the form until the next total refresh.
The easiest way to fix this problem is to flag the form so that it knows that you will be drawing in the `Paint` event handler and that resizing the form should cause a complete re-draw. The program `PaintDrawX` does this by placing the following code in its `Load` event handler:

```csharp
private void Form1_Load(object sender, EventArgs e)
{
    this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.ResizeRedraw, true);
}
```

Now when the form is resized, it raises a `Paint` event with an `e.Graphics` parameter that represents the whole form. Comment out the call to `SetStyle` and resize the form a few times to see the difference.

The second confusing situation that can arise when drawing in the `Paint` event handler occurs if you use `CreateGraphics` to create a `Graphics` object. This code works almost as you would expect, and you can even draw on that `Graphics` object. However, when the `Paint` event handler completes, the form re-draws itself with the graphics produced by the `e.Graphics` object so that whatever you drew on the other `Graphics` object is immediately lost. You can easily avoid this problem if you don’t use `CreateGraphics` inside `Paint` event handlers. (In fact, you should rarely need to use `CreateGraphics`.)

**Using e.Graphics in PrintPage Event Handlers**

When you use a `PrintDocument` to generate a printout, that object raises `PrintPage` events to generate the pages that it must draw. (This is described in greater detail in Section 6, “Printing.”) Like the `Paint` event handler, `PrintPage` provides an `e.Graphics` parameter that you can use to draw graphics.

The `SimplePrint` example program uses the following code to draw an ellipse on a single printout page. It uses the `e.Graphics` object’s `DrawEllipse` method, passing it the page’s margin bounds so that the ellipse fills the page. It finishes by setting `e.HasMorePages` to `false` so that the `PrintDocument` doesn’t try to generate any more pages.

```csharp
{
    e.Graphics.DrawEllipse(Pens.Green, e.MarginBounds);
    e.HasMorePages = false;
}
```

**Using a Graphics Object**

The `Graphics` class provides many methods for drawing and filling shapes. These methods fall into two categories: those that draw shapes and those that fill areas. Methods that draw shapes all take a `Pen` as a parameter to determine the shape’s line characteristics. Methods that fill areas all take a `Brush` as a parameter to determine how the area is filled.

Many of the filling methods have corresponding drawing methods. In those cases, the two methods’ parameters are the same except the drawing method takes a `Pen` and the filling method takes a `Brush`. Their names are even the same except “Draw” is replaced with “Fill.”
Table 1 summarizes the Graphics class’s drawing and filling methods.

### Table 1: Graphics Class’s Drawing and Filling Methods

<table>
<thead>
<tr>
<th>Drawing Method</th>
<th>Filling Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawArc</td>
<td>N/A</td>
<td>Draws an elliptical arc</td>
</tr>
<tr>
<td>DrawBezier</td>
<td>N/A</td>
<td>Draws a Bezier curve</td>
</tr>
<tr>
<td>DrawBeziers</td>
<td>N/A</td>
<td>Draws a series of connected Bezier curves</td>
</tr>
<tr>
<td>DrawClosedCurve</td>
<td>FillClosedCurve</td>
<td>Draws/fills a smooth closed curve</td>
</tr>
<tr>
<td>DrawCurve</td>
<td>N/A</td>
<td>Draws a smooth curve that is not closed</td>
</tr>
<tr>
<td>DrawEllipse</td>
<td>FillEllipse</td>
<td>Draws/fills an ellipse</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawIcon</td>
<td>Draws an icon image</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawIconUnstretched</td>
<td>Draws an icon image with its original size</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawImage</td>
<td>Draws some or all of an image at a given location, possibly resized</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawImageUnscaled</td>
<td>Draws an image at a given location at its original size</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawImageUnscaledAndClipped</td>
<td>Draws an image at a given location at its scale but possibly clipped to fit an output area</td>
</tr>
<tr>
<td>DrawLine</td>
<td>N/A</td>
<td>Draws a line segment</td>
</tr>
<tr>
<td>DrawLines</td>
<td>N/A</td>
<td>Draws a series of connected line segments</td>
</tr>
<tr>
<td>DrawPath</td>
<td>FillPath</td>
<td>Draws/fills a path defined by a GraphicsPath object</td>
</tr>
<tr>
<td>DrawPie</td>
<td>FillPie</td>
<td>Draws/fills an elliptical pie slice</td>
</tr>
<tr>
<td>DrawPolygon</td>
<td>FillPolygon</td>
<td>Draws/fills a series of connected line segments and connects the last to the first to make a closed shape</td>
</tr>
<tr>
<td>DrawRectangle</td>
<td>FillRectangle</td>
<td>Draws/fills a rectangle</td>
</tr>
<tr>
<td>DrawRectangles</td>
<td>FillRectangles</td>
<td>Draws/fills a series of rectangles</td>
</tr>
<tr>
<td>N/A</td>
<td>FillRegion</td>
<td>Fills an area defined by a Region object</td>
</tr>
<tr>
<td>N/A</td>
<td>DrawString</td>
<td>Draws a text string</td>
</tr>
</tbody>
</table>

Most of the Graphics class’s drawing and filling methods are straightforward. You can learn more about them in the Help or online at http://msdn2.microsoft.com/library/system.drawing.graphics_methods.aspx.
There are several different versions of the online Help for different versions of the .NET Framework. Unfortunately, some of those versions are incomplete. You can find information for the .NET Framework 3.5 version of the Graphics class at http://msdn2.microsoft.com/library/system.drawing.graphics_methods(VS.90).aspx, but you may have to look at older versions to get details for many of the methods.

The image drawing methods are a bit different because, despite the fact that their names begin with “draw,” they fill an area. They also fill with some sort of image rather than a brush.

The DrawString method’s name also begins with “draw,” but it uses a brush to fill text.

Three other important Graphics methods are Clear, CopyFromScreen, and Dispose. The Clear method clears the drawing surface with a specified color, and CopyFromScreen copies part of the screen image onto the drawing surface.

The Dispose method frees resources used by the Graphics object. To make resource recycling more efficient, you should always call the Graphics object’s Dispose method when you are done drawing with it. Alternatively, you can build a using block for the Graphics object.

The UseCopyFromScreen example program uses the following code to copy a piece of the screen onto a bitmap:

```csharp
int x = int.Parse(txtXmin.Text);
int y = int.Parse(txtYmin.Text);
int wid = int.Parse(txtXmax.Text) - x + 1;
int hgt = int.Parse(txtYmax.Text) - y + 1;

Bitmap bm = new Bitmap(wid, hgt);
using (Graphics gr = Graphics.FromImage(bm))
{
    gr.CopyFromScreen(x, y, 0, 0, new Size(wid, hgt));
}
picResult.Image = bm;
```

After getting the location and size of the area to copy from textboxes, the code creates a Bitmap of the correct size.

It then makes a Graphics object to draw on the Bitmap. It makes the Graphics object in a using statement so that C# automatically calls the object’s Dispose method when it reaches the end of the block.

Inside the using block, the program calls the CopyFromScreen method to copy part of the screen’s image into the Bitmap.

The code finishes by displaying the Bitmap in the picture box named picResult.

While it is important to call the Dispose method when you are finished with a Graphics object, don’t do the same thing for a Bitmap as long as the Bitmap is still in use. In this example, the picture box needs to use the Bitmap to re-draw itself. If you call bm.Dispose, the picture box will fail to re-draw and the program will crash.
The Graphics object provides three properties that determine the quality of the graphics produced and that deserve special mention.

First, InterpolationMode determines the quality of image operations. When the Graphics object draws an image at a new scale, this property determines the quality of the new image. Setting this property to a high-quality value such as High or Bilinear makes drawing take slightly longer but produces a better result.

Second, TextRenderingHint determines the quality of drawn text. Setting this to a high-quality value such as AntiAliasGridFit again slows performance somewhat, but the results can be much nicer than those given by faster settings.

Finally, the SmoothingMode property determines whether the Graphics object smoothes the edges of drawn lines and shapes.

Check the online Help or just experiment with different values for these properties to see what effects they have. Unless you are drawing an extremely complicated picture, the higher-quality settings provide a much better result without a noticeable delay.

**Creating Pens**

The Pens class defines more than a hundred stock pens that you can use with very little effort. These are solid pens of various colors with width 0. (A pen with width 0 is always drawn as thinly as possible even if the Graphics object has been stretched. Section 5, “Using Transformations,” explains transformations including stretching.)

The following code uses the Pens.Green pen to draw a rectangle with a 1-pixel-wide green border:

```csharp
e.Graphics.DrawRectangle(Pens.Green, 10, 10, 100, 50)
```

Stock pens are sufficient for many tasks, and they keep your code simple. Occasionally, however, you may not want a thin, solid pen. In that case, you can create new Pen objects.

The Pen class provides a couple of useful constructors that you can use to create non-standard pens. You can pass the constructor a Color to create a pen of that color. You can also pass the constructor the width that you would like the pen to have.

The following code creates a new Pen that is orange and 10 pixels wide and uses it to draw a line from the point (30, 30) to the point (200, 30):

```csharp
using (Pen the_pen = new Pen(Color.Orange, 10))
{
    e.Graphics.DrawLine(the_pen, 30, 30, 200, 30);
}
```
Notice that this code makes its `Pen` in a `using` clause so that C# automatically calls the pen’s `Dispose` method when the block ends. While you should call `Dispose` for any pen that you create, do not call `Dispose` for a stock pen or your program will crash.

Other versions of the `Pen` class’s constructor take a `Brush` instead of a `Color` as a parameter. C# then fills the area covered by the pen with the brush.

The following code creates a `LinearGradientBrush` that shades from red to blue (the following section says more about this). It then uses the `Brush` to make a new 10-pixel-wide `Pen` and uses the `Pen` to draw an ellipse. The result is an ellipse with a thick border that shades from red in the upper left to blue in the lower right.

```csharp
using (Brush br = new LinearGradientBrush(
    new Point(20, 100), new Point(170, 200),
    Color.Red, Color.Blue))
{
    using (Pen the_pen = new Pen(br, 10))
    {
        e.Graphics.DrawEllipse(the_pen, 20, 100, 150, 100);
    }
}
```

The `Pen` class has a few useful properties for controlling its appearance. As you can probably guess, the `Color` property determines the pen’s color. Normally you select the color by picking a stock pen or by passing the color into a new pen’s constructor, but sometimes it’s useful to change a pen’s color after you have created it. For example, if you build a thick pen with a custom dash style, it is easier to change the color and reuse the pen than it is to create several new pens.

The `DashStyle` property determines the pen’s dash style. Standard `DashStyle` values include `Solid`, `Dash`, `DashDot`, `DashDotDot`, and `Dot`. The following code draws a polygon with a thick, purple, dashed border.

```csharp
Point[] pts = {
    new Point(250, 50),
    new Point(220, 100),
    new Point(270, 150),
    new Point(200, 170),
    new Point(180, 80),
    new Point(210, 30)
};
using (Pen the_pen = new Pen(Color.Purple, 5))
{
    the_pen.DashStyle = DashStyle.Dash;
    e.Graphics.DrawPolygon(the_pen, pts);
}
```
The code first creates an array of `Point` objects to define the polygon. It creates a thick, purple pen, sets its ` DashStyle ` to Dash, and draws the polygon.

> The DashStyle values (and many other more advanced drawing features) are defined in the System.Drawing.Drawing2D namespace. If you need to use these values a lot, you may want to import that namespace into your code.

The `UsePens` example program demonstrates simple stock pens, thick pens, pens defined by a brush, and dashed pens.

The `Pen` class’s `DashCap` property determines how the ends of a dash are drawn. This property can take the values ` Triangle, Flat, and Round.`

The `DashOffset` property determines how far into the first dash the line begins. For example, if the `Pen`’s `DashOffset` property is 1, then the line starts 1 unit into the first dash.

Note that dash measurements are relative to the pen’s thickness. For example, a dot is 1 unit long. If a pen is 10 pixels wide, that means each dot is 10 pixels long.

The `DashCaps` example program, which is shown in Figure 1, draws three line segments with `DashStyle` values `Dash, DashDot, and DashDotDot`. It sets the lines’ `DashCap` values to `Triangle, Flat, and Round`.

After it draws the three lines, the example program skips some space, sets the `Pen` object’s `DashOffset` property to 1, and then draws the lines again.

![Figure 1: The DashCaps example program showing the three DashStyle values Dash, DashDot, and DashDotDot.](image)

Just as the `DashCap` property determines how the ends of dashes are drawn, the `StartCap` and `EndCap` properties determine how a line’s end points are drawn. This property can take the values `ArrowAnchor, DiamondAnchor, Flat, Round, RoundAnchor, Square, SquareAnchor, and Triangle. The LineCaps example program, which is shown in Figure 2, demonstrates these values.
Section 2, “Using Advanced Pens and Brushes,” describes some even more advanced Pen features.

**Creating Brushes**

Just as the Pens class defines stock pens, the Brushes class defines stock brushes with various colors.

*Just as you should not call a stock pen’s Dispose method, you should not call a stock brush’s Dispose method.*

The StockBrushes example program uses the following code to fill an ellipse with yellow and then outline it in orange:

```csharp
// Fill an ellipse.
e.Graphics.FillEllipse(Brushes.Yellow, 20, 20, 200, 150);

// Outline the ellipse.
using (Pen the_pen = new Pen(Color.Orange, 5))
{
    e.Graphics.DrawEllipse(the_pen, 20, 20, 200, 150);
}
```

The stock brushes are of the SolidBrush class, a fairly simple class that provides only one interesting property: Color. You can use the class’s constructor to make solid brushes with non-standard colors, but otherwise it’s a pretty boring class. C# provides several other brush classes that are a bit more interesting.

The HatchBrush class provides an assortment of hatched brushes. These fill an area with a background color covered by some sort of pattern drawn in a foreground color. Figure 3 shows samples of the available hatch styles drawn with blue foreground color on a yellow background color.
The HatchStyles example program demonstrates hatch brush patterns.

The HatchedRectangle example program uses the following code to draw a rectangle filled with diagonal bricks:

```csharp
using (HatchBrush br = new HatchBrush(HatchStyle.DiagonalBrick, Color.Red, Color.Pink))
{
    e.Graphics.FillRectangle(br, 10, 10, 200, 150);
    e.Graphics.DrawRectangle(Pens.Red, 10, 10, 200, 150);
}
```

Because hatched brushes are not stock objects, you should call their `Dispose` methods either explicitly or with a `using` block as in the previous code.

The `TextureBrush` class fills an area with an image that you provide. The TexturedRectangle example program uses the following code to fill its form’s background. It makes a `TextureBrush`, passing its constructor the bitmap resource `My.Resources.smile`. It then uses that brush to fill the form’s client area.

```csharp
// Fill an area with a texture.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    using (TextureBrush br = new TextureBrush(Properties.Resources.smile))
    {
        e.Graphics.FillRectangle(br, this.ClientRectangle);
    }
}
```
Figure 4 shows the TexturedRectangle example program in action.

![TexturedRectangle](image)

**Figure 4:** The TexturedRectangle example program uses a TextureBrush to fill its form with an image.

No matter where the area is that a TextureBrush fills, it starts tiling the image relative to the upper-left corner of the Graphics object on which it is drawing. If the area you are drawing starts in the upper-left corner of the Graphics object, the image lines up nicely as it does in Figure 4. If the area’s corner is not an integer multiple of the size of the image away from the Graphics object’s corner, the image doesn’t line up.

The TextureTests example program demonstrates this issue. The program fills the rectangle on the left with the same brush used in Figure 4. If you compare the two pictures carefully, you can see that the images in the two pictures line up.

![TextureTests](image)

**Figure 5:** The TextureTests example program fills an area with an image at its default origin and another area with a translated origin.
Because every filled area starts tiling at the same origin, if you fill overlapping areas, their images line up smoothly. If you are filling a single area that is not lined up properly, as on the left in Figure 5, the result can look strange.

One way to fix this problem is to translate the brush’s origin. The following code shows how the program TextureTests fills its rectangles. It draws its first rectangle with a normal TextureBrush. It then uses the brush’s TranslateTransform method to move its origin to the position where it will fill its second rectangle. Now when it fills the rectangle on the right, the image begins in the rectangle’s upper-left corner.

```csharp
// Fill some rectangles with different origins.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    int wid = Properties.Resources.smile.Width;
    int hgt = Properties.Resources.smile.Height;

    using (TextureBrush br = new TextureBrush(Properties.Resources.smile))
    {
        e.Graphics.FillRectangle(br, 30, 30, wid, hgt);
        e.Graphics.DrawRectangle(Pens.Red, 30, 30, wid, hgt);

        br.TranslateTransform(150, 30);
        e.Graphics.FillRectangle(br, 150, 30, wid, hgt);
        e.Graphics.DrawRectangle(Pens.Red, 150, 30, wid, hgt);
    }
}
```

**Section 1 Wrap-up**

Section 1 is a bit longer than the others because it covers concepts that you need to understand for any graphics programming. Every graphics program uses pens and brushes to draw on a Graphics object.

This section explains how to obtain Graphics objects. The two most common methods are to use the e.Graphics parameter in a Paint or PrintPage event handler and to make a Graphics object associated with a bitmap. The second technique lets you build a persistent image to display and is the closest thing C# has to an auto-re-draw capability.

This section also explains how to use stock pens and brushes; how to build pens that are thick, dashed, and with different end caps; and how to make hatched and textured brushes. These kinds of pens and brushes can handle most of your graphics needs.

The next section describes more advanced pens and brushes that provide new features such as lines with customized dash patterns, longitudinal stripes, and customized end caps; and brushes that fill with color gradients.
Section 2: Using Advanced Pens and Brushes

Most graphics programs can make do with relatively simple pens and brushes, but occasionally it’s nice to bump it up a notch. This section describes more advanced pen and brush classes and techniques that can add an extra dimension of style and interest to an otherwise ordinary application.

Custom Dash Patterns

The `Pen` class supports five standard dash patterns: `Solid`, `Dash`, `DashDot`, `DashDotDot`, and `Dot`. Those are good enough for simple drawings, but if you need to distinguish among many lines, you can create your own dash patterns.

This technique is also useful when the standard dash patterns don’t look good for a particular line thickness. Normally a unit of line length is the same as its width. If a line is 1 pixel wide, then a dot is also 1 pixel long. If a line is 10 pixels wide, then a dot is 10 pixels long. Sometimes you may not like the results. In particular, since the default dash and dot lengths are fairly small for thin lines, I often prefer bigger dashes.

If you set a `Pen` object’s `DashStyle` property to `Custom`, then you can set its `DashPattern` property to an array of floats that determine the line’s pattern of drawn and skipped pieces. For example, a `DashPattern` of `{4, 1, 2, 1}` means draw 4 units, skip 1 unit, draw 2 units, skip 1 unit, and then repeat.

The CustomDash example program, which is shown in Figure 6, draws several lines with different dash styles. The top two lines show the standard `Dot` and `Dash` styles. The third line uses a custom dash style defined by the array `{10, 2}`.

The units used to measure dashes are the same as the pen’s width. To define a dash pattern in absolute numbers of pixels, divide the number of pixels by the pen’s width. For example, the fourth line in Figure 6 draws a thick line. It uses 10-pixel-wide dashes separated by 2-pixel-wide gaps just as the previous line does. This pen is 5 pixels wide, thus the program uses the array `{10/5, 2/5}` to make this pattern.

The fifth line in Figure 6 uses the same thick pen and the dash pattern `{10, 2}`. The final line uses the array `{5, 2, 5, 2, 2, 2, 2, 2}` to make a thick dash-dash-dot-dot pattern.

Figure 6: The CustomDash example program draws lines with custom dash styles.
The program CustomDash uses the following code to draw its lines:

```csharp
// Draw some custom dash patterns.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    using (Pen the_pen = new Pen(Color.Blue))
    {
        int y = 10;

        // Standard dot.
        the_pen.DashStyle = DashStyle.Dot;
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;

        // Standard dash.
        the_pen.DashStyle = DashStyle.Dash;
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;

        // Thin with long dashes.
        the_pen.DashStyle = DashStyle.Custom;
        the_pen.DashPattern = new float[] { 10, 2 };
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;

        // Thick with the same absolute dash lengths.
        the_pen.Width = 5;
        the_pen.DashStyle = DashStyle.Custom;
        the_pen.DashPattern = new float[] { 2f, 0.4f };
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;

        // Thick with the same relative dash lengths.
        the_pen.Width = 5;
        the_pen.DashStyle = DashStyle.Custom;
        the_pen.DashPattern = new float[] { 10f, 2f };
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;

        // Thick Dash Dash Dot Dot.
        the_pen.Width = 5;
        the_pen.DashStyle = DashStyle.Custom;
        the_pen.DashPattern = new float[] { 5f, 2f, 5f, 2f, 2f, 2f, 2f, 2f, 2f, 2f };
        e.Graphics.DrawLine(the_pen, 10, y, 250, y);
        y += 15;
    }
}
```

**Longitudinal Stripes**

Just as you can use an array of floats to define a custom dash pattern, you can use a similar array to define longitudinal stripes. This array should contain pairs of numbers between 0.0 and 1.0 that represent where drawing should start and stop across the width of the pen. To define a striped line, set a Pen object’s CompoundArray property to the array.
For example, the array \( \{0.0, 0.4, 0.6, 1.0\} \) means draw from 0.0 to 0.4 of the line’s width, skip to 0.6, and then draw the rest of the width to 1.0. The result is a line that has a blank stripe that is 20 percent of the line’s width \((0.6 - 0.4 = 0.2, \text{or } 20\%)\) running down the middle.

The array \( \{0.0, 0.1, 0.3, 0.7, 0.9, 1.0\} \) draws a thin section from 0.0 to 0.1, skips from 0.1 to 0.3, draws a thicker section from 0.3 to 0.7, skips another section from 0.7 to 0.9, and draws another thin section from 0.9 to 1.0.

The Stripes example program, shown in Figure 7, demonstrates these two arrays. It draws the red triangle with the first pen and the green ellipse with the second.

![Figure 7: The Stripes example program draws lines with longitudinal stripes.](image)

The program Stripes uses the following code to draw its shapes. Notice that it sets the Graphics object’s SmoothingMode property to make the lines smooth.

```csharp
// Draw some shapes with striped lines.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
    using (Pen the_pen = new Pen(Color.Red, 20))
    {
        Point[] pts = {
            new Point(20, 20),
            new Point(120, 20),
            new Point(70, 120)
        };
        the_pen.CompoundArray = new float[] {0.0f, 0.4f, 0.6f, 1.0f};
        e.Graphics.DrawPolygon(the_pen, pts);
        the_pen.Color = Color.Green;
        the_pen.CompoundArray = new float[] {0.0f, 0.1f, 0.3f, 0.7f, 0.9f, 1.0f};
        e.Graphics.DrawEllipse(the_pen, 150, 20, 100, 150);
    }
}
```
If any of the values in the CompoundArray property is smaller than 0.0 or greater than 1.0, or the values are not arranged in increasing order, or if the array doesn’t contain an even number of values, the program crashes.

**Custom End Caps**

You can set a `Pen` object’s `StartCap` and `EndCap` properties to determine how a line’s ends are drawn. Figure 2 shows the possible line cap values. For some applications, the values shown in Figure 2 may not be good enough. In some cases, you may want to use specialized line caps to better suit your application. (I once worked on a professional football coaching tool where players’ routes were drawn with various kinds of lines. The end caps showed the type of block linesmen were supposed to perform: angled left, angled right, square, pushing left, etc.)

You can make a custom start cap or end cap by setting a `Pen` object’s `CustomStartCap` and `CustomEndCap` properties to a new `CustomLineCap` object. One of the `CustomLineCap` constructors takes as parameters `GraphicsPath` objects that it should use to fill and draw the custom cap. Finally, you can make a `GraphicsPath` object and use its methods to add lines, curves, and other shapes to the path.

The `CustomEndCaps` example program uses the following code to display a line with custom caps:

```csharp
// Draw lines with custom end caps.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    using (Pen the_pen = new Pen(Color.Green, 5))
    {
        // Define the start cap.
        Point[] start_pts = {
            new Point(0, 0),
            new Point(-3, 3),
            new Point(3, 3),
            new Point(0, 0)
        };
        GraphicsPath start_path = new GraphicsPath();
        start_path.AddLines(start_pts);
        CustomLineCap start_cap = new CustomLineCap(null, start_path);
        the_pen.CustomStartCap = start_cap;

        // Define the end cap.
        Point[] end_pts = {
            new Point(0, 0),
            new Point(-3, 0),
            new Point(-3, 3),
            new Point(3, 3)
        };
        GraphicsPath end_path = new GraphicsPath();
        end_path.AddLines(end_pts);
        CustomLineCap end_cap = new CustomLineCap(null, end_path);
        the_pen.CustomEndCap = end_cap;

        // Draw a sample.
    }
}
```
The code creates a thick, green pen. It then makes an array that defines the custom start cap. If you stand on the line looking off the starting end, X coordinates increase to the left, and Y coordinates increase to the front (off of the line).

Next the program creates a GraphicsPath object and uses its AddLines method to add the lines defined by the array to the path. It then creates a CustomLineCap object, passing its constructor the GraphicsPath. Finally, the code sets the Pen object’s CustomStartCap property to the new CustomLineCap.

The code performs similar steps to create a custom end cap. The process is similar to the one used to create a custom start cap. Even the coordinate system is the same: If you stand on the end of the line looking away from the line, X increases to the left, and Y increases to the front.

Figure 8 shows the result.

**Figure 8:** The CustomEndCaps example program draws a line with custom start and end caps.

**Linear Gradient Brushes**

Like other brush classes, the LinearGradientBrush class fills an area. This kind of brush fills the area with a color gradient that blends smoothly from one color to another or that blends between several colors.

You can create a simple LinearGradientBrush by passing its constructor two colors and two points that determine where the colors should be. The brush smoothly shades from one color to the other between those points.
A second version of the class’s constructor takes as parameters two colors, a rectangle, and a flag indicating how the colors should be used to fill the rectangle (horizontally, vertically, or diagonally).

The LinearGradientBrushes example program uses the following code to create a horizontal gradient brush for a rectangle. It then fills that rectangle with the brush.

```csharp
// Draw with a horizontal gradient.
Rectangle rect1 = new Rectangle(20, 20, 250, 100);
using (LinearGradientBrush br = new LinearGradientBrush(
    rect1, Color.Red, Color.Blue, LinearGradientMode.Horizontal))
{
    e.Graphics.FillRectangle(br, rect1);
    e.Graphics.DrawRectangle(Pens.Black, rect1);
}
```

The LinearGradientBrush class provides some properties that you can use to make more complicated shading patterns. The InterpolationColors property is a ColorBlend object that defines a series of colors and their positions within the blend. The brush then smoothly shades between all of the colors.

The program LinearGradientBrushes uses the following code to create a horizontal gradient brush and creates a ColorBlend object. It sets the object’s Colors property to an array of colors that the brush should use. It sets the object’s Positions property to an array of floats giving the positions of the colors between 0.0 (the blend’s start) and 1.0 (the blend’s end). This example uses the colors of the rainbow spaced equally throughout the blend area.

```csharp
// Draw with a horizontal gradient.
Rectangle rect2 = new Rectangle(20, 140, 250, 100);
using (LinearGradientBrush br = new LinearGradientBrush(
    rect2, Color.Red, Color.Blue, LinearGradientMode.Horizontal))
{
    // Define a rainbow blend.
    ColorBlend color_blend = new ColorBlend();
    color_blend.Colors = new Color[]
    {
        Color.Red, Color.Orange, Color.Yellow,
        Color.Lime, Color.Blue, Color.Indigo, Color.DarkViolet
    };
    color_blend.Positions = new float[]
    {0 / 6f, 1 / 6f, 2 / 6f, 3 / 6f, 4 / 6f, 5 / 6f, 6 / 6f};
    br.InterpolationColors = color_blend;
    // Fill a rectangle with the blended brush.
    e.Graphics.FillRectangle(br, rect2);
    e.Graphics.DrawRectangle(Pens.Black, rect2);
}
```

Figure 9 shows the program LinearGradientBrushes in action. The upper rectangle shows a simple blend fading from red to blue. The bottom rectangle shows the more complicated rainbow blend.
Figure 9: The LinearGradientBrushes example program draws two filled rectangles.

Note that the number of colors must match the number of positions and that the positions must start with 0.0 and end with 1.0. Strangely, the positions do not need to be in increasing order, and mixing them up can produce some interesting results.

Path Gradient Brushes

The colors in a LinearGradientBrush vary smoothly along a linear path. In a PathGradientBrush, the colors vary smoothly between a “center point” (which need not be in the actual center) and the points along a path. The brush’s CenterPoint and CenterColor properties determine the center’s location and color. The SurroundColors property is an array giving the colors that should be used for the points along the path.

The GradientStar example program uses the following code to fill a star:

```csharp
// Make a path gradient brush.
using (PathGradientBrush br = new PathGradientBrush(pts))
{
    br.CenterPoint = new Point(cx, cy);
    br.CenterColor = Color.White;
    br.SurroundColors = new Color[
    ];

    // Fill the star.
    e.Graphics.FillPolygon(br, pts);
}

// Outline the star.
    e.Graphics.DrawPolygon(Pens.Black, pts);
```
The code creates a `PathGradientBrush` defined by a `PointF` array named `pts` (the code that builds the start’s points isn’t shown here). It sets the brush’s center to the point \((cx, cy)\) at the center of the star and makes the center point white.

The program then defines the `SurroundColors` array. The array contains 10 colors alternating red and blue. Because this is a five-pointed star, the colors are distributed so that the star’s tips are red and the areas between them are blue. (If you don’t include enough colors for all of the path’s points, the brush repeats the final color as many times as necessary.)

Finally, the code fills the star with the brush and then outlines the star in black. Figure 10 shows the result.

![GradientStar](image)

**Figure 10:** The GradientStar example program draws a star filled with a `PathGradientBrush`.

The `LinearGradientBrush` and `PathGradientBrush` classes provide other properties to let you fine-tune how the colors shade from one to another. For example, you can make the blend start slowly and then shade quickly near the end of the gradient. These techniques don’t provide a huge additional benefit for most applications, so I won’t cover them here. See the online Help for details (http://msdn2.microsoft.com/library/system.drawing.drawing2d.lineargradientbrush.aspx and http://msdn2.microsoft.com/library/system.drawing.drawing2d.pathgradientbrush.aspx).

### Section 2 Wrap-up

Many graphics programs rely on stock pens and brushes, but with only a little extra work, you can build something really special. Custom dash styles, stripes, and line caps let you draw lines that are truly unique. Linear and path gradient fills can also add an extra bit of interest to an otherwise run-of-the-mill application. These features may seldom be absolutely essential, but adding them only takes a few minutes and can really impress your users.

The `Graphics` object’s draw methods (those with names beginning with “Draw”—`DrawRectangle`, `DrawLine`, `DrawPolygon`, etc.) use pens to determine how objects are drawn, while its fill methods use...
Section 3: Drawing Text

It’s easy to display text in a Label control. In fact, if the Label control does everything you need, you should use it and skip this section.

However, if you need more control over text, C# provides lots of tools to help. For example, if you want to draw text in multiple colors and fonts or text centered and wrapped within a drawing area with text that doesn’t fit ended with an ellipsis, C# can do the job. This is both good and bad: It’s good that C# provides lots of tools, but learning to get the most out of those tools can be a struggle.

Drawing Simple Text

The Graphics object’s DrawString method displays text. The DrawString method takes as its first three parameters the string to draw, the font to draw it in, and a brush to use to fill the text. Different overloaded versions of the method take other parameters that specify the string’s location and formatting in various ways. You can specify the text’s position by X and Y coordinates, by a PointF structure, or by a formatting rectangle.

The simplest way to display text is by using X and Y coordinates. The SimpleText example program uses the following code to display three lines of text:

```csharp
// Draw some text.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    int x = 10;
    int y = 10;

    using (Font the_font = new Font("Times New Roman", 20,
                                           FontStyle.Bold, GraphicsUnit.Point))
    {
        e.Graphics.DrawString("SimpleText", the_font, Brushes.Blue, x, y);
        SizeF text_size = e.Graphics.MeasureString("SimpleText", the_font);
        e.Graphics.DrawRectangle(Pens.Red, x, y,
                                 text_size.Width, text_size.Height);
        y += (int)text_size.Height;
    }

    using (Font the_font = new Font("Comic Sans MS", 20,
                                           FontStyle.Regular, GraphicsUnit.Point))
    {
        e.Graphics.DrawString("SimpleText", the_font, Brushes.Blue, x, y);
        SizeF text_size = e.Graphics.MeasureString("SimpleText", the_font);
        e.Graphics.DrawRectangle(Pens.Red, x, y,
                                 text_size.Width, text_size.Height);
    }
}
```
The code creates a font and uses the `DrawString` method to draw some sample text in that font. It then calls the `Graphics` object’s `MeasureString` method to see how big the text is in that font and draws a rectangle around the text. It then increases variable `y` by the height of the text to start a new line and repeats these steps to draw two other samples with different fonts.

Figure 11 shows the program SimpleText displaying its sample strings. Notice that the three rectangles are slightly different sizes even though all three were drawn with 20-point fonts. Since different fonts and styles (such as italic and bold) have slight differences in spacing and character size, the resulting strings have different heights and widths.

Using Layout Rectangles

Instead of using X and Y coordinates to specify the text’s position, you can pass `DrawString` a rectangle in which it should draw the text. The `DrawString` method automatically wraps the text as needed to fit within the area.
The WrappedText example program uses the following code to draw a long string so that it fills the program’s form. It first makes a rectangle that fills the form inside a 10-pixel-wide margin. It then calls the DrawString method, passing this rectangle as the place to draw the text. It finishes by drawing this formatting rectangle.

```csharp
// Draw some text wrapped to fill the form.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.Clear(this.BackColor);
    e.Graphics.TextRenderingHint = TextRenderingHint.AntiAliasGridFit;

    // Make a margin rectangle.
    Rectangle rect = new Rectangle(10, 10,
        this.ClientSize.Width - 20, this.ClientSize.Height - 20);

    // Draw the text inside the rectangle.
    using (Font the_font = new Font("Times New Roman", 20,
        FontStyle.Regular, GraphicsUnit.Point))
    {
        e.Graphics.DrawString(SAMPLE_TEXT, the_font,
            Brushes.Black, rect);
        e.Graphics.DrawRectangle(Pens.Blue, rect);
    }
}
```

Figure 12 shows the result. Notice that the text is wrapped at word breaks to fit within the rectangle. Also notice that the last line is chopped off vertically so only the tops of its letters are visible.

Figure 12: The WrappedText example program displaying text in a formatting rectangle.

Layout rectangles make it much easier to display long pieces of text wrapped across multiple lines.

In addition to a formatting rectangle, the DrawString method can take as a parameter a StringFormat object that specifies the text’s formatting characteristics. Using this object’s properties, you can align the text vertically or horizontally, determine how the last line is displayed (in Figure 12 it is partly visible), and determine how text that is not shown is represented (e.g., with an ellipsis).
The most useful `StringFormat` properties are `Alignment`, `FormatFlags`, `HotkeyPrefix`, `LineAlignment`, and `Trimming`. The following sections describe these properties in detail.

### Alignment and LineAlignment

The `Alignment` property determines how text is aligned horizontally within the formatting rectangle. This property can take the values `Near` (left aligned), `Center`, and `Far` (right aligned).

The `LineAlignment` property determines how the text is aligned vertically within the formatting rectangle. This property can take the values `Near` (top aligned), `Center`, and `Far` (bottom aligned).

### FormatFlags

The `FormatFlags` property specifies general formatting characteristics. Some of the more useful `FormatFlags` values include:

- **FitBlackBox** — This allows characters to slightly overhang the formatting rectangle if necessary.
- **LineLimit** — This indicates that a line which cannot fit completely within the vertical space of the formatting rectangle should be dropped entirely. For example, in Figure 12, the last line does not fit. If `LineLimit` were set, that line would be completely omitted.
- **NoClip** — This indicates that text drawn outside of the formatting rectangle should be visible. If this were set in Figure 12, the last line would appear entirely even though part of it would lie below the formatting rectangle.
- **NoWrap** — This prevents `DrawString` from wrapping lines to fit in the formatting rectangle. If a line doesn’t fit, it is truncated according to the other `StringFormat` properties. The text will break to a new line only if it contains line feeds.

### HotkeyPrefix

The `HotkeyPrefix` property determines whether the drawn text displays hotkeys if the text contains ampersand (&) characters. For example, if the text is “&Click Me,” then the ampersand means that the following character C is the hotkey and should normally be underlined, as in “ClicK Me.”

The `HotkeyPrefix` property can take these values:

- **Hide** — In this case, any ampersand characters in the text are hidden — for example, “Click Me.”
- **None** — In this case, any ampersand characters are displayed as ampersands in the text — for example, “&Click Me.”
- **Show** — In this case, the character following an ampersand is underlined — for example, “ClicK Me.”
Trimming

The Trim property indicates how text should be truncated if it won’t fit completely within the formatting rectangle. This property can take the following values:

- **Character** — Text is trimmed to the nearest character.
- **EllipsisCharacter** — Text is trimmed to the nearest character and is followed by an ellipsis.
- **EllipsisPath** — This value assumes that the text is a slash-delimited string similar to a directory path. In that case, the middle of the text is replaced with an ellipsis. This lets the user see the beginning and end of the path.
- **EllipsisWord** — Text is trimmed to the nearest word and is followed by an ellipsis.
- **None** — Specifies no trimming.
- **Word** — Text is trimmed at the nearest word.

The FormattedText example program uses the following code to display text in a formatting rectangle. It is similar to the code used by the program WrappedText except it uses a different font and a StringFormat object to specify centered horizontal alignment, line limit (partial lines are not shown vertically), hotkey prefix (show underlined characters), and ellipsis word trimming (trim to the nearest word and display an ellipsis).

```csharp
// Draw formatted text.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.Clear(this.BackColor);
    e.Graphics.TextRenderingHint = TextRenderingHint.AntiAliasGridFit;

    // Make a margin rectangle.
    Rectangle rect = new Rectangle(10, 10, this.ClientSize.Width - 20, this.ClientSize.Height - 20);

    // Draw the text inside the rectangle.
    using (Font the_font = new Font("Times New Roman", 20, FontStyle.Regular | FontStyle.Italic, GraphicsUnit.Point))
    {
        using (StringFormat sf = new StringFormat())
        {
            sf.Alignment = StringAlignment.Center;
            sf.FormatFlags = StringFormatFlags.LineLimit;
            sf.Trimming = StringTrimming.EllipsisWord;
            sf.HotkeyPrefix = HotkeyPrefix.Show;

            e.Graphics.DrawString(SAMPLE_TEXT, the_font, Brushes.Black, rect, sf);
            e.Graphics.DrawRectangle(Pens.Blue, rect);
        }
    }
}
```
Figure 13 shows the program FormattedText displaying its text.

Figure 13: The FormattedText example program displaying text centered horizontally, hotkey characters, partial lines omitted, and trailing ellipsis.

Section 3 Wrap-up

The Graphics object’s DrawString method lets you draw text. Its MeasureString method tells you how big a piece of text will be drawn so you can arrange text properly.

To get the greatest benefit out of DrawString, however, you need to use a formatting rectangle and a StringFormat object. If you provide those two parameters, DrawString can wrap text across multiple lines, align text horizontally and vertically, truncate text as needed, and even display ellipses to indicate omitted text. While there are occasions when you have more work to do (e.g., if you want to display some words within the text in different colors or fonts), DrawString is a powerful tool for formatting text.

For information on an alternative to using DrawString to arrange and display text and other objects, see Section 8, “Building FlowDocuments.”

The previous sections explained how to fill and draw various shapes such as lines, rectangles, ellipses, and text. The following section explains how to work with a different kind of graphics: images. It explains how to load, modify, and save raster image data.

Section 4: Manipulating Images

The previous sections in this Wrox Blos have explained how to draw lines, shapes, and text at a moderately high level. You don’t need to understand what colors to give which pixels to draw a line; you simply call a Graphics object’s DrawLine method, and C# does the rest.
Usually it’s easiest to work at this high level, but sometimes you may want to work directly with the pixels in an image. This is particularly useful when you want to manipulate an existing image such as a bitmap, JPG, or GIF file, or when you want to draw something and then save it in a file of one of those types.

The following sections explain how to load, manipulate, and save these kinds of images.

## Creating and Loading Bitmaps

Creating a new bitmap from scratch is simple. Simply create a new `Bitmap` object, passing its constructor the dimensions that the image should have. The `MakeBitmap` example program uses the following code to create and display a bitmap. It creates a new `Bitmap` object and makes a `Graphics` object associated with it. It uses that object to draw on the `Bitmap` and displays the result in the `PictureBox` named `picResult`.

```csharp
// Create and display a bitmap.
private void Form1_Load(object sender, EventArgs e)
{
    // Make the Bitmap.
    Bitmap bm = new Bitmap(200, 150);

    // Make an associated Graphics object.
    using (Graphics gr = Graphics.FromImage(bm))
    {
        // Draw.
        gr.SmoothingMode = SmoothingMode.AntiAlias;
        gr.Clear(Color.Yellow);
        using (Pen thick_pen = new Pen(Color.Red, 5))
        {
            gr.DrawEllipse(thick_pen, 3, 3, 195, 145);
            thick_pen.Color = Color.Green;
            gr.DrawLine(thick_pen, 3, 3, 197, 147);
            thick_pen.Color = Color.Blue;
            gr.DrawLine(thick_pen, 3, 147, 197, 3);
        }
    }

    // Display the result in a PictureBox.
    picResult.Image = bm;
}
```

The only “gotcha” in this code is that it must not call the `Bitmap` object’s `Dispose` method. The `PictureBox` needs the `Bitmap` to re-draw itself. If you dispose of the `Bitmap`, then the program crashes when the `PictureBox` tries to re-draw.
Figure 14 shows the program MakeBitmap displaying its Bitmap object.

![Figure 14: The MakeBitmap example program creates a Bitmap and draws on it.](image)

Loading a Bitmap from an image file is just as easy. Simply create a new Bitmap object, passing its constructor the name of the file you want to open. For example, the following code creates a Bitmap object and loads the file C:\Temp\test.bmp into it. It then displays the Bitmap by setting a PictureBox control’s Image property to it. The code does all this work inside a Try Catch block so it won’t crash if it cannot read the file.

```csharp
try
{
    // Load and display the image.
    Bitmap bm = new Bitmap("C:\Temp\test.bmp");

    // Display the image in a PictureBox.
    picTest.Image = bm;
    catch (Exception ex)
    {
        MessageBox.Show("Error opening file\n" + ex.Message);
    }
}
```

This code leads to one small problem. While the Bitmap object is alive, it seems to hold some sort of handle to the original image file. That means you cannot modify or delete that file while the program is running.

To work around this problem, the program can load the file, make a copy of the Bitmap, and then dispose of the original Bitmap. Now the image file is free, and the program still has a copy of the image. Unfortunately, since a clone of the original Bitmap seems to inherit the same file handle, you need to make a completely new Bitmap and copy the image into it.

The following code shows one solution. It loads the image file into a Bitmap as before. It makes a new Bitmap of the same size, creates a Graphics object associated with it, uses that object to draw on the new Bitmap, and disposes of the original Bitmap.

```csharp
```
try
{
// Load and display the image.
Bitmap bm = new Bitmap("C:\Temp\test.bmp");

Bitmap original_bm = new Bitmap(file_name);
Bitmap bm = new Bitmap(original_bm.Width, original_bm.Height);
using (Graphics gr = Graphics.FromImage(bm))
{
    gr.DrawImageUnscaled(original_bm, 0, 0);
}
original_bm.Dispose();
original_bm = null;

// Display the image in a PictureBox.
picTest.Image = bm;
}
catch (Exception ex)
{
    MessageBox.Show("Error opening file\n" + ex.Message);
}

The LoadImageFile example program demonstrates these two methods for loading an image file. It uses compilation constants to let you test the locking or non-locking version of the code.

**Manipulating Bitmaps**

After you have created or loaded a Bitmap, there are several ways you can modify it. Some of the previous examples showed one method that creates a Graphics object associated with the Bitmap and then uses that object to draw. The program MakeBitmap, which is shown in Figure 14, demonstrates this technique.

A Graphics object is good if you want to draw high-level items such as lines, polygons, and ellipses on a Bitmap, but it’s sometimes useful to manipulate an image’s pixels directly. The Bitmap object provides two methods, GetPixel and SetPixel, that make this possible.

The GetPixel method takes as parameters the X and Y coordinates of the pixel that you want to use and returns a Color representing that pixel’s color. You can then use Color properties to study the color. For example, you can use the A, R, G, B to learn about the Color’s alpha (opacity), red, green, and blue color components.

The SetPixel method takes as parameters a pixel’s X and Y coordinates and the Color that you want to make the pixel.

The InvertPixels example program uses the following code to demonstrate both methods of drawing on a Bitmap: by using a Graphics object and by using GetPixel and SetPixel.

```csharp
// Invert the image.
private void mnuDataInvert_Click(object sender, EventArgs e)
{
    this.Cursor = Cursors.WaitCursor;
    Application.DoEvents();

    Bitmap bm = (Bitmap)(picOriginal.Image.Clone());
```
The code first makes a clone of the original image stored in the picOriginal control’s Image property. It calculates the size and midpoints of the image and then enters a loop that makes variables x and y cover every pixel in the image’s upper-left quadrant.

For each pixel, the code gets the pixel’s color. It creates a new color that is the inverse of the original color by subtracting the red, green, and blue components from 255 (the colors lie in the range 0 to 255, so subtracting from 255 gives a value in the range 255 to 0). It then uses SetPixel to give the pixel in the new Bitmap the new color.
The program repeats these steps to invert the pixels in the image’s lower-right quadrant.

Next the code creates a `Graphics` object associated with the `Bitmap` and uses it to draw rectangles around the upper-left and lower-right quadrants.

Finally, the code displays the result in the `PictureBox` named `picInverted`.

Figure 15 shows the result.

![Figure 15: The InvertPixels example program inverts some of the pixels in an image.](image)

The InvertPixels example program contains code to load the original image similar to the code described in the previous section. It also contains code to save the partly inverted result. That code is described in the next section.

### Saving Image Files

After you’ve generated a beautiful image, you’ll want to be able to save it so that you can e-mail it to friends, post it all over the Web, print it on tee-shirts, and so forth. Saving a `Bitmap` into a file in C# is easy. Simply call the `Bitmap` object’s `Save` method, passing it the name of the file you want to create and a parameter indicating the format you want to use (BMP, JPEG, GIF, BMP, etc.).

It is the second parameter that determines the file’s format, not the file’s name. For example, you can tell the `Save` method to write the `Bitmap` into a file named `Test.gif` with the JPEG format. If you do that, your system may have trouble displaying the image. If you double-click on the file, the system will try to open it with the program assigned to view GIF files. If that program doesn’t realize that the file actually has the JPEG format, it will fail. To prevent this kind of trouble, make the file extension match its format.
The following code shows how the program InvertImage lets you save the Bitmap it builds:

```csharp
// Save the result bitmap.
private void mnuFileSaveAs_Click(object sender, EventArgs e)
{
    if (sfdImage.ShowDialog() == DialogResult.OK)
    {
        try
        {
            Bitmap bm = (Bitmap)picInverted.Image;
            string file_name = sfdImage.FileName;
            string ext = file_name.Substring(file_name.LastIndexOf(".")).ToLower();
            switch (ext)
            {
                case ".bmp":
                    bm.Save(file_name, ImageFormat.Bmp);
                    break;
                case ".gif":
                    bm.Save(file_name, ImageFormat.Gif);
                    break;
                case ".jpg":
                case ".jpeg":
                    bm.Save(file_name, ImageFormat.Jpeg);
                    break;
                case ".png":
                    bm Save(file_name, ImageFormat.Png);
                    break;
                case ".tif":
                case ".tiff":
                    bm.Save(file_name, ImageFormat.Tiff);
                    break;
                default:
                    MessageBox.Show("Unknown file extension " + ext);
                    return;
            }
            MessageBox.Show("Ok", "Done", MessageBoxButtons.OK,
                        MessageBoxIcon.Information);
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error opening file " + ofdImage.FileName + 
                        "\n" + ex.Message);
        }
    }
}
```

The code displays a Save File dialog. If the user enters a file name and clicks Save, the code gets the file name and determines its extension. It then uses a switch statement to save the file in an appropriate format. This example handles the extensions bmp, gif, jpg, jpeg, png, tif, and tiff. If the file has some other extension, the program complains and doesn’t save the file.
Section 4 Wrap-up

The Bitmap class’s constructors allow you to easily create a Bitmap or load one from a file.

After you have a Bitmap, you can associate a Graphics object with it to draw lines and shapes on it. You can use the Bitmap object’s GetPixel and SetPixel methods to view and change individual pixels.

After you have finished editing a Bitmap, you can use its Save method to save the resulting image into a file in a wide variety of formats.

All of the methods described so far in this Wrox Blox measure positions and sizes in pixels. While it is natural to think of the computer’s screen in terms of pixels, it doesn’t always make sense to think of the data in that way. The next section explains how you can transform the data so that you can work with coordinates that make sense to you while still producing useful results on the screen.

Section 5: Using Transformations

All of the discussion and examples so far have been drawn in C#’s native drawing units: pixels. By default, all of the Graphics object’s filling and drawing methods work in pixels.

However, pixels may not always be the most convenient unit for you. For example, suppose you want to draw a simple bar chart showing sales figures between $1M and $30M for the years 1990 through 2000. Since you can’t use pixel coordinates as large as (1995, 1,000,000), you’ll need to perform some mathematical calculations to map these big values into something that will fit on your screen.

While you can perform those calculations yourself if you like, the Graphics object provides transformation methods that can do this for you. The main transformation methods are TranslateTransform, ScaleTransform, and RotateTransform, which translate, scale, and rotate subsequent graphics, respectively.

Basic Transformations

The TranslateTransform method takes as parameters the distances by which graphics should be offset in the X and Y directions; the ScaleTransform method takes parameters that indicate the amount by which graphics should be scaled in the X and Y directions; and RotateTransform indicates the angle in degrees by which graphics should be rotated around the origin.

For example, the following code uses TranslateTransform to translate subsequent graphics 100 pixels in the X direction and 50 pixels in the Y direction. It then draws a rectangle with $-50 \leq X \leq 50, -50 \leq Y \leq 50$. After translation, the result appears on the form in the area $50 \leq X \leq 150, 0 \leq Y \leq 100$

```csharp
    e.Graphics.TranslateTransform(100, 50)
    e.Graphics.DrawRectangle(Pens.Red, -50, -50, 100, 100)
```

Individually these three methods are fairly simple, but you can combine them to implement very complex transformations. For example, you could scale, rotate, and then translate a drawing.
All three of these methods take an optional final parameter that indicates whether the transformation should be applied before or after any previous transformations. The difference is important because, for example, a rotation followed by a translation generally does not give the same result as a translation followed by a rotation.

By default, transformations are applied before any previous transformations. I find this counterintuitive. Why would I want to have code I add later apply before earlier code? To avoid problems, I always append new transformations after the previous ones.

In addition to these three basic transformation methods, the Graphics class provides several other related methods. One of the most important of these is ResetTransform, which resets the object’s transformation to its default non-transformed value. After you finish using a transformation, it’s usually a good idea to reset the transformation so that it doesn’t affect later drawing code, possibly confusing you.

The TransformedSmiley example program uses the following code to draw a smiley face centered at the origin on a Graphics object. Notice that this code uses a pen with width 0. A Pen with width 0 is always drawn as thinly as possible (1 pixel wide), no matter how it is transformed. If you draw with a stock pen such as Pens.Blue, then the result is modified by any scaling transformation and, in this example at least, produces some weird results.

```csharp
private void DrawSmiley(Graphics gr)
{
    using (Pen thin_pen = new Pen(Color.Blue, 0))
    {
        gr.FillEllipse(Brushes.Yellow, -1, -1, 2, 2);
        gr.DrawEllipse(thin_pen, -1, -1, 2, 2);
        gr.DrawArc(thin_pen, -0.75f, -0.75f, 1.5f, 1.5f,
                   0, 180);
        gr.FillEllipse(Brushes.Red, -0.2f, -0.2f, 0.4f, 0.6f);
        gr.DrawEllipse(Brushes.White, -0.5f, -0.6f, 0.3f, 0.5f);
        gr.DrawEllipse(thin_pen, -0.5f, -0.6f, 0.3f, 0.5f);
        gr.FillEllipse(Brushes.Black, -0.4f, -0.5f, 0.2f, 0.3F);
        gr.FillEllipse(Brushes.White, 0.2f, -0.6f, 0.3f, 0.5f);
        gr.DrawEllipse(thin_pen, 0.2f, -0.6f, 0.3f, 0.5f);
        gr.FillEllipse(Brushes.Black, 0.3f, -0.5f, 0.2f, 0.3F);
    }
}
```

The program uses the following code to draw three copies of the smiley face:

```csharp
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;

    // Draw 50x50 pixels.
    e.Graphics.ScaleTransform(50, 50);
    e.Graphics.TranslateTransform(60, 60, MatrixOrder.Append);
    DrawSmiley(e.Graphics);
    e.Graphics.ResetTransform();

    // Draw 50x100 pixels.
    e.Graphics.ScaleTransform(50, 100);
}
To draw the first face, the code scales by a factor of 50 in the X and Y directions. This enlarges the original smiley to fill the area $-50 \leq X \leq 50$, $-50 \leq Y \leq 50$. It then translates the result by 60 pixels in the X and Y directions, so the scaled face lies entirely on the form in the area $10 \leq X \leq 110, 10 \leq Y \leq 110$.

To draw the second face, the code scales by a factor of 50 in the X direction and 100 in the Y direction to produce a tall, thin face. It then translates to move the result onto the visible part of the form.

To draw the final face, the code rotates the original smiley by 45 degrees. It then scales the result by 50 in the X direction and -30 in the Y direction. Scaling by -30 in the Y direction makes the result 30 times bigger and inverts the Y coordinate so that the result is flipped vertically. Finally, the code translates the result onto the visible part of the form.

Notice that the code calls `ResetTransform` after drawing each smiley face to reset the transformation. If it did not do this, all of the transformations would add up and produce some very odd results (the second smiley face’s nose is so big it covers the entire form).

Figure 16 shows the program TransformedSmiley displaying its three transformed smiley faces.

```csharp
e.Graphics.TranslateTransform(170, 110, MatrixOrder.Append);
DrawSmiley(e.Graphics);
e.Graphics.ResetTransform();

// Draw 50x30 pixels rotated and flipped vertically.
e.Graphics.RotateTransform(45, MatrixOrder.Append);
e.Graphics.ScaleTransform(50, -30, MatrixOrder.Append);
e.Graphics.TranslateTransform(60, 170, MatrixOrder.Append);
DrawSmiley(e.Graphics);
e.Graphics.ResetTransform();
```
World Coordinate Mapping

One particularly common kind of transform maps a rectangle in some sort of world coordinate system to a rectangle on the screen in device coordinates. For example, suppose you want to draw a bar chart showing sales figures between $1M and $30M for the years 1990 through 2000. Drawing will be easier if you can map the world coordinates 1990 ≤ year ≤ 2000, $1M ≤ sales ≤ $30M to the device coordinates that cover the form 0 ≤ X ≤ form width, 0 ≤ Y ≤ form height.

The BarChart example program uses the MapRectangles subroutine shown in the following code to build such a transformation for a Graphics object.

```csharp
// Transform the Graphics object so
// world coordinates wxmin <= X <= wxmax, wymin <= Y <= wymax are mapped to
// device coordinates dxmin <= X <= dxmax, dymin <= Y <= dymax.
private void MapRectangles(Graphics gr,
    float wxmin, float wxmax, float wymin, float wymax,
    float dxmin, float dxmax, float dymin, float dymax)
{
    // Make a world coordinate rectangle.
    RectangleF wrectf = new RectangleF(wxmin, wymin, wxmax - wxmin, wymax - wymin);

    // Make PointF objects representing the upper left, upper right,
    // and lower right corners of device coordinates.
    PointF[] dpts = {
        new PointF(dxmin, dymin),
        new PointF(dxmax, dymin),
        new PointF(dxmin, dymax)
    };

    // Map the rectangle to the points.
    gr.Transform = new Matrix(wrectf, dpts);
}
```

It's not too hard to build a transformation to map one rectangle to another (translate to the original rectangle to the origin, scale, translate to the final destination), but the Matrix class provides a constructor that does this for you. This constructor takes as parameters a source rectangle in world coordinates and an array of three PointF structures defining the upper-left, upper-right, and lower-left corners of the device rectangle. The code prepares the RectangleF and PointF structures, creates a Matrix, and sets the Graphics object's Transform property to the result.

The following code shows how the program BarChart uses the subroutine MapRectangles:

```csharp
// Draw a bar chart filling the form.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
    const int MIN_YEAR = 1990;
    const int MAX_YEAR = 2001;
    const int MIN_SALES = 0;
    const int MAX_SALES = 30000000;
    // Map the coordinates 1990 <= X <= 2001, 0 <= Y <= 30M
```
The code first declares some constants to define the world coordinate bounds. It then calls `MapRectangles` to define a transformation to map the world coordinates to the form’s client area minus a 10-pixel margin. Notice that the minimum Y value in device coordinates is greater than the maximum Y value. Inside the subroutine `MapRectangles`, this gives the device coordinate rectangle a negative height, and that flips the drawing vertically. The result is that world coordinates are now drawn so that they increase upward as you would normally expect for a bar chart.
After building the transformation, the program defines some data values. It then loops through the values drawing rectangles for each.

Next the code adds a new translation transformation of 0.5 years in the X direction. It prepends the translation to the existing transformation, and thus shifts the data in world coordinates before the other transformations are applied. Finally, the code draws lines connecting the data points.

Figure 17 shows the program BarChart displaying its data.

![Figure 17: The BarChart example program uses transformations to draw a bar chart in world coordinates.](image)

Transformations apply to all graphical operations, not just drawing lines and rectangles. The TransformedText example program uses the following code to draw a transformed bitmap and transformed text:

```csharp
// Draw a transformed bitmap and transformed text.
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TextRenderingHint = TextRenderingHint.AntiAliasGridFit;
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
    e.Graphics.InterpolationMode = InterpolationMode.HighQualityBilinear;
    e.Graphics.TranslateTransform(-Properties.Resources.Smiley.Width / 2,
                                -Properties.Resources.Smiley.Height / 2);
    e.Graphics.ScaleTransform(1.5f, 1f, MatrixOrder.Append);
    e.Graphics.RotateTransform(-30, MatrixOrder.Append);
    e.Graphics.TranslateTransform(220, 150, MatrixOrder.Append);
    e.Graphics.DrawImageUnscaled(Properties.Resources.Smiley, 0, 0);
    e.Graphics.ResetTransform();
    using (Font the_font = new Font("Times New Roman", 16))
```
```csharp
{ 
    e.Graphics.ScaleTransform(1, 4);
    e.Graphics.TranslateTransform(10, 10, MatrixOrder.Append);
    e.Graphics.DrawString("Tall And Thin", the_font, Brushes.Blue, 0, 0);
    e.Graphics.ResetTransform();

    e.Graphics.RotateTransform(45);
    e.Graphics.TranslateTransform(30, 90, MatrixOrder.Append);
    e.Graphics.DrawString("Rotated 45", the_font, Brushes.Blue, 0, 0);
    e.Graphics.ResetTransform();

    e.Graphics.RotateTransform(-90);
    e.Graphics.TranslateTransform(100, 200, MatrixOrder.Append);
    e.Graphics.DrawString("Rotated -90", the_font, Brushes.Blue, 0, 0);
    e.Graphics.ResetTransform();

    e.Graphics.ScaleTransform(4, 1);
    e.Graphics.TranslateTransform(140, 10, MatrixOrder.Append);
    using (StringFormat sf = new StringFormat())
    {
        sf.Alignment = StringAlignment.Center;
        sf.LineAlignment = StringAlignment.Near;
        Rectangle rect = new Rectangle(0, 0, 60, 80);
        e.Graphics.DrawString("Short And Wide", the_font,
                              Brushes.Blue, rect, sf);
        e.Graphics.DrawRectangle(Pens.Red, rect);
    }
    e.Graphics.ResetTransform();
}
```

Figure 18 shows the result.

Figure 18: The TransformedText example program draws a transformed bitmap and transformed text.
The NameGrid example program, which is shown in Figure 19, performs a more useful task, using transformations to draw a grid with angled names across the top.

![Figure 19: The NameGrid example program uses transformations to draw rotated text above a grid.](image)

Download the sample project to see the code.

**Section 5 Wrap-up**

Transformations may very well be the most powerful and underused feature in .NET graphics programming. They simplify programming tasks by allowing you to work in world coordinates that are convenient for you. They also let you reuse code by calling the same drawing routines with different transformations. Finally, they let you produce special effects that are not otherwise possible, such as rotated images and text.

The sections up to this point have shown how to produce graphics on a form. The next section explains how to produce the same results on a print preview or a printout.

**Section 6: Printing**

The way in which you draw graphics for printing and print previewing is exactly the same as the way you draw on a bitmap, form, picture box, or other screen object. In every case, you obtain a `Graphics` object and use its methods to create a picture.
The only difference is in how you obtain the Graphics object. When you are drawing on the screen, you use CreateGraphics or the e.Graphics parameter provided by a Paint event handler. Similarly, when you generate a printout or print preview, you use an e.Graphics parameter provided by a PrintDocument object’s PrintPage event handler.

**Using PrintPage**

The PrintDocument class represents a printout. When you print or create a print preview, you use a PrintDocument object created either at design time or at run time in the code. That object raises a PrintPage event when it needs you to generate output. Like the Paint event handler, PrintPage provides an e.Graphics parameter that you can use to generate graphics.

In addition to its Graphics property, the e parameter provides the following useful properties, shown in Table 2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancel</td>
<td>If you set this to True, the printout is canceled. While this seems to work for printing, it doesn’t cancel print previews.</td>
</tr>
<tr>
<td>MarginBounds</td>
<td>Returns a Rectangle representing the page’s margins. Normally you should print within this area.</td>
</tr>
<tr>
<td>HasMorePages</td>
<td>Set this value to False when there are no more pages to print.</td>
</tr>
<tr>
<td>PageBounds</td>
<td>Returns a Rectangle representing the page’s physical bounds.</td>
</tr>
<tr>
<td>PageSettings</td>
<td>Returns a PageSetting object that describes the page’s settings.</td>
</tr>
</tbody>
</table>

The form used by the PrintWithPreview example program contains a PrintDocument named pdShapes created at design time. It also contains a PrintPreviewDialog named ppdShapes with the Document property set to pdShapes. When the program displays the dialog, it automatically uses the PrintDocument to generate the graphics that it displays in the preview.

The example program uses the following code to display the print preview dialog:

```csharp
// Display the print preview.
private void btnPreview_Click(object sender, EventArgs e)
{
    m_NextPage = 0;
    ppdShapes.ShowDialog();
}
```
The following code shows how the program sends a printout directly to the printer without displaying a preview:

```csharp
// Print without a preview.
private void btnPrint_Click(object sender, EventArgs e)
{
    m_NextPage = 0;
    pdShapes.Print();
}
```

The most interesting part of this program is the `PrintDocument` object’s `PrintPage` event handler, which is shown in the following code:

```csharp
// Generate a page of the printout.
private int m_NextPage;

private void pdShapes_PrintPage(object sender,
{
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;

    int xmin = e.MarginBounds.Left;
    int ymin = e.MarginBounds.Top;
    int xmax = e.MarginBounds.Right;
    int ymax = e.MarginBounds.Bottom;
    int xmid = (int)(xmin + (xmax - xmin) / 2);
    int ymid = (int)(ymin + (ymax - ymin) / 2);

    // Draw a different shape for each page
    switch (m_NextPage)
    {
        case 0:
            // Draw a triangle.
            Point[] pts = {
                new Point(xmid, ymin),
                new Point(xmax, ymax),
                new Point(xmin, ymax)
            };
            using (Pen thick_pen = new Pen(Color.Blue, 10))
            {
                thick_pen.DashStyle = DashStyle.Dot;
                e.Graphics.DrawPolygon(thick_pen, pts);
            }
            break;
        case 1:
            // Draw an ellipse.
            using (Pen thick_pen = new Pen(Color.Red, 10))
            {
                e.Graphics.DrawEllipse(thick_pen, e.MarginBounds);
            }
    }
}
```
// Draw a rectangle.
using (Pen thick_pen = new Pen(Color.Green, 10))
{
    thick_pen.DashStyle = DashStyle.Dash;
    e.Graphics.DrawRectangle(thick_pen, e.MarginBounds);
}
break;

case 2:
    // Draw an X.
    using (Pen thick_pen = new Pen(Color.Black, 10))
    {
        thick_pen.DashStyle = DashStyle.Custom;
        thick_pen.DashPattern = new float[] {10, 10};
        e.Graphics.DrawLine(thick_pen, xmin, ymin, xmax, ymax);
        e.Graphics.DrawLine(thick_pen, xmin, ymax, xmax, ymin);
    }
break;

// Draw the page number.
using (Font the_font = new Font("Times New Roman", 250,
    FontStyle.Bold, GraphicsUnit.Point))
{
    using (StringFormat sf = new StringFormat())
    {
        sf.Alignment = StringAlignment.Center;
        sf.LineAlignment = StringAlignment.Center;
        e.Graphics.DrawString(m_NextPage.ToString(),
            the_font, Brushes.Black, xmid, ymid, sf);
    }
}
m_NextPage++;
e.HasMorePages = (m_NextPage <= 3);

Both of the button event handlers set m_NextPage = 0 before generating a printout or preview. The PrintPage event handler uses m_NextPage to determine what it should draw on a page. When it has finished drawing the page, the code increments m_NextPage and sets e.HasMorePages to false if m_NextPage is greater than 3.

Figure 20 shows the program PrintWithPreview displaying a print preview. The buttons on the preview dialog let the user view one, two, three, four, or six printout pages at a time. They also let the user zoom in on the pages or send the printout to the printer.
Using Other Event Handlers

While PrintPage is the most important event provided by PrintDocument, the class does provide a few other events that are occasionally useful. The BeginPrint event occurs before the printout is generated. You could use this event handler to open files, fetch data from a database, and perform other setup tasks.

The PrintBooklet example program uses the following code to remember whether the program is generating a printout or a print preview. Later the PrintPage event handler checks this value and draws only in black if the program is printing a printout. This example also sets m_NextPage = 0 in this event handler rather than making the Print and Print Preview buttons do it.

```csharp
private void pdShapes_BeginPrint(object sender, System.Drawing.Printing.PrintEventArgs e)
{
    // Remember whether we're printing (as opposed to previewing).
}
```
The `EndPrint` event occurs after the printout is finished. You can use this event handler to close files, shut down databases, and otherwise clean up after the printing process.

The `QueryPageSettings` event occurs just before each `PrintPage` event. You can use this event to prepare for printing the next page. The program `PrintBooklet` assumes that its pages will be printed double-sided. It adds some extra space called a *gutter* to the margins on alternating sides of the pages to allow room for binding. (In this example, the gutter is 2 inches to make it easy to see. That's bigger than you would probably need in a real booklet.) The code adds the gutter on the left of even-numbered pages and on the right of odd-numbered pages.

```csharp
m_IsPrinting = (e.PrintAction == PrintAction.PrintToPrinter);

// Start with the first page.
m_NextPage = 0;
}
```

The code that draws the pages in the `PrintPage` event handler doesn't need to know about the gutters. The `QueryPageSettings` event handler adjusts the margins, and `PrintPage` simply draws inside those margins.

Figure 21 shows the program `PrintBooklet` displaying a preview of its four pages. This program draws each page's margins in red so that you can easily see the gutters.
Printing Transformations

Just as you can use transformations while drawing on the screen, you can use similar transformations while drawing a printout. The process is exactly the same as the one described in Section 5, “Using Transformations”; thus, it isn’t described in much detail here. However, there are three particularly common uses of transformations in printouts: centering a drawing, scaling a drawing to fit the page, and stretching a drawing to fill the page.

The `MapRectangles` subroutine described in the subsection, “World Coordinate Mapping,” in Section 5, “Using Transformations”; maps, a world coordinate rectangle to a device coordinate rectangle. You can use that routine to stretch a picture in world coordinates to fill the printout’s margins. The program `PrintBarChart` uses the following code to stretch the world coordinate area $\min_x \leq x \leq \max_x$, $\min_y \leq y \leq \max_y$ to fill the page’s margins:

```csharp
MapRectangles(e.Graphics,
    MIN_X, MAX_X, MIN_Y, MAX_Y,
    e.MarginBounds.Left, e.MarginBounds.Right,
    e.MarginBounds.Top, e.MarginBounds.Bottom);
```

The program uses the `CenterRectangles` subroutine shown in the following code to center the world coordinate rectangle within the device coordinate rectangle. It calculates the width and height of the world coordinates. It then adds and subtracts half those values from the device rectangle’s center to

Figure 21: The PrintBooklet example program adds gutters to alternating sides of its pages.
determine the location of the device rectangle that will center the world coordinates. After it has calculated
the necessary device coordinate bounds, the routine calls MapRectangles to build the final transformation.

```csharp
private void CenterRectangles(Graphics gr,
float wxmin, float wxmax, float wymin, float wymax,
float dxmin, float dxmax, float dymin, float dymax)
{
    // Figure out where we need to map in device coordinates.
    float wwid = wxmax - wxmin;
    float whgt = wymax - wymin;
    float dxmid = (dxmax + dxmin) / 2;
    float dymid = (dymax + dymin) / 2;

    dxmin = dxmid - wwid / 2;
    dxmax = dxmid + wwid / 2;
    dymin = dymid - whgt / 2;
    dymax = dymid + whgt / 2;

    // Map the rectangles.
    MapRectangles(gr, wxmin, wxmax, wymin, wymax, dxmin, dxmax, dymin, dymax);
}
```

Note that the subroutine CenterRectangles relies on the fact that the minimum bounds are smaller
than the maximum bounds. For example, wxmin < wxmax and dxmin < dxmax. The drawing code was
adjusted to draw the bar chart without a flipping transformation.

The program PrintBarChart uses the FitRectangles subroutine shown in the following code to make a
world coordinate rectangle as large as possible within a device rectangle without distorting it:

```csharp
private void FitRectangles(Graphics gr,
float wxmin, float wxmax, float wymin, float wymax,
float dxmin, float dxmax, float dymin, float dymax)
{
    // Compare the world and device aspect ratios.
    float dxmid = (dxmax + dxmin) / 2;
    float dymid = (dymax + dymin) / 2;
    float wwid = wxmax - wxmin;
    float whgt = wymax - wymin;
    float dwid = dxmax - dxmin;
    float dhgt = dymax - dymin;
    float new_dwid, new_dhgt;

    if (wwid / whgt > dwid / dhgt)
    {
        // World rectangle is shorter/wider than device rectangle.
        // Make the result as wide as possible.
        new_dhgt = dwid * whgt / wwid;
        new_dwid = dwid;
    }
    else
    {
        // World rectangle is taller/thinner than device rectangle.
        // Make the result as tall as possible.
        new_dwid = dhgt * wwid / whgt;
        new_dhgt = dhgt;
    }

    dxmin = dxmid - new_dwid / 2;
    dxmax = dxmid + new_dwid / 2;
    dymin = dymid - new_dhgt / 2;
    dymax = dymid + new_dhgt / 2;
}
```
This code compares the world and device rectangles’ aspect ratios (the width-to-height ratios) to see whether the world rectangle is tall and thin or short and wide compared to the device rectangle. That comparison determines whether the world rectangle will fill the device rectangle vertically or horizontally. The code sets the limiting dimension equal to the corresponding device rectangle dimension and calculates the other dimension.

For example, suppose the world rectangle is 100 units wide and 200 units tall and the device rectangle is a square 300 units on a side. The world rectangle is tall and thin compared to the device rectangle, so it is limited in height. The resized world rectangle should have height 300 so that it fills the device rectangle vertically. The new width is 100/200 × 300 = 150 units. The resized world rectangle fills the device rectangle vertically, so it can’t be made any larger and still fit. The new world rectangle’s aspect ratio is 150/300 = 1/2, just as it was before; thus, this doesn’t distort the rectangle.

After it calculates the world rectangle’s new size, the code determines where to position the rectangle to center it and then calls `MapRectangles` to build the final transformation.

Figure 22 shows the program `PrintBarChart` displaying a print preview of the same bar chart arranged in three different ways. The first page uses the subroutine `CenterRectangles` to center the chart. The second page uses the subroutine `FitRectangles` to enlarge the chart to fill as much of the page as possible without distorting it. The last page uses `MapRectangles` to make the chart fill the page even though that distorts it. The program draws a red rectangle around the pages’ margins, so you can see how the chart fits.

![Print preview](image)
Section 6 Wrap-up

The key to printing and print previews is the PrintDocument object. Your code should catch that
object's PrintPage event to generate the printout's pages. It can also catch the BeginPrint, EndPrint,
and QueryPageSetting events to provide support for the main printing routines.

Subroutines such as FillRectangles, CenterRectangles, and FitRectangles make mapping world
coordinates easier for any graphics program, but they are particularly useful for printing where
programs often need to center or resize graphics. (You may want to copy those routines from the
PrintBarChart example program so that you can use them in your projects.)

This section gives you the tools you need to generate printouts, but there's still a lot of work to do. The
techniques in this section, combined with the text-drawing techniques described in Section 3, “Drawing
Text,” let you produce pages that are all text or that contain images, but making text flow around
pictures and other objects can be difficult. (After reading about WPF Graphics in Section 7, Section 8 on
FlowDocument objects gives you an easier approach to mixing graphics and text.)

The sections up to this point have explained how to use code to produce graphics at run time. The next
section explains how to use WPF graphics to produce graphics at run time.

Section 7: Using WPF Graphics

WPF (Windows Presentation Foundation) is a fairly recent addition to Visual Studio. It includes a whole
new set of forms, controls, components, and other gimmickry for building form-based applications. The
Visual Studio development environment includes a large set of tools for building and editing WPF forms
and the controls they contain. Unfortunately, those tools are somewhat limited, and there are lots of
things you cannot do easily with the WPF form designer. For example, the designer lets you set a
Rectangle object's Fill property to a solid color, but it won't let you set it to a more complicated brush.

WPF is extremely complicated, thus there's hardly room here to even scratch the surface. Since the focus
of this Wrox Blox is graphics, this section concentrates on WPF elements that you can use to produce
graphics. The following sections describe WPF controls that you can use to display decorations and
draw shapes. A separate section discusses brushes that you can use to fill WPF graphics. For a more
complete discussion of WPF, see the online Help or search the Web.

Decorative Controls

WPF includes an assortment of the usual kinds of controls: text boxes, labels, buttons, scroll bars, and so
forth. You can use these controls more or less as you would in a Windows Forms application.

Three controls that play a mainly decorative role are Border, GroupBox, and Label. I consider these
controls decorative because they usually just sit there and don't interact with the user. If you can use them
to provide the graphics that you need, go ahead and avoid the extra work needed to use more complicated
controls or code. Since these controls are straightforward, I won't say any more about them here.
Shape Controls

Windows Forms controls don’t include any shape controls; thus, if you want to draw an ellipse or rectangle on a form, you must do so in code. In contrast, WPF provides several shape controls that let you draw graphics at design time. Unfortunately, since the WPF design time editors are not very powerful, there are many graphics tasks that you cannot accomplish using the designer alone. In those cases, you must write XAML (eXtensible Application Markup Language, pronounced “zammel”) code to fill in the missing pieces.

The ShapeControls example program, which is shown in Figure 23, demonstrates the most useful WPF shape controls. The following sections describe these controls and show the XAML code that this program uses to draw them.

![Shape Controls](image)

Figure 23: The ShapeControls example program displaying several WPF shape controls.

**Line**

The Line control displays a line segment. You can set its XAML properties to specify the line’s position, color, thickness, start and end caps, dash caps, and so forth.

The Line control does not use the same kind of DashStyle property that the Windows Forms Pen class uses. Instead, you can set its StrokeDashArray property to a series of numbers that define the dash pattern much as a Pen object’s DashPattern array does.

The following code shows how the program ShapeControls draws its dashed line:

```xml
<Line Height="37" Margin="110,0,11,9" Name="Line1" Stroke="BlueViolet"
    StrokeThickness="10" VerticalAlignment="Bottom" X2="200" Y2="20"
    StrokeDashArray="3,3" StrokeDashCap="Triangle" StrokeEndLineCap="Round"
    StrokeStartLineCap="Round" />
```

52
Rectangle
The Rectangle control draws a rectangle. Its properties let you determine the rectangle’s line color, size, and fill color. The following code shows how the program ShapeControls draws its rectangle. The RadiusX and RadiusY properties determine the curvature at the rectangle’s corners. Leaving these at their default values of 0 gives a normal, square-cornered rectangle.

```xml
<Rectangle Height="73" Margin="13,10,0,0" Name="Rectangle1" Stroke="Red" VerticalAlignment="Top" HorizontalAlignment="Left" Width="86" StrokeThickness="4" Fill="LightBlue" RadiusX="10" RadiusY="50" />
```

Ellipse
The Ellipse control draws an ellipse. Its properties are similar to those of the other shape controls and let you determine the ellipse’s size, position, line color, and fill color. The following code shows how the program ShapeControls draws its ellipse.

```xml
<Ellipse HorizontalAlignment="Left" Margin="13,97,0,0" Name="Ellipsel" Stroke="Cyan" StrokeThickness="10" Width="86" Height="147" VerticalAlignment="Top" Fill="Lime" />
```

Image
The Image control displays an image. Its Stretch property determines how the image is sized to fit the control. The Stretch property can take the values shown in Table 3.

<table>
<thead>
<tr>
<th>Stretch Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Displays the image at its original size.</td>
</tr>
<tr>
<td>Fill</td>
<td>Stretches the image to fill the control.</td>
</tr>
<tr>
<td>Uniform</td>
<td>Stretches the image uniformly so that it is as large as possible while still fitting within the control without distortion. The image is centered in the control.</td>
</tr>
<tr>
<td>UniformToFill</td>
<td>Stretches the image uniformly so that it completely fills the control. Some of the image may not fit within the control and will be clipped.</td>
</tr>
</tbody>
</table>

The Image control’s Source property determines what image is displayed. This can be a file path, URL, or a path to a resource.

A file path might be absolute, as in `C:\Images\logo.bmp`, or relative to the current directory, as in `logo.bmp` or `Images\logo.bmp`.

A URL is the address of an image file available on the Web as in `www.vb-helper.com/vb_prog_ref.jpg`. 
To make the control load an embedded resource, add an image file to the application’s resources and set the image’s Build Action property to Resource. Then if the file’s name is smile.bmp, you can specify the Image control’s Source property as in Resources/smile.bmp:

The following code shows how the program ShapeControls displays an image stored in the resource smile.bmp:

```xml
<Image Height="73" HorizontalAlignment="Left" Margin="111,12,0,0" Name="Image1" Stretch="None" VerticalAlignment="Top" Width="71" Source="Resources/smile.bmp"/>
```

**Polygon**

The Polygon control draws a polygon much as the Graphics object can with its DrawPolygon and FillPolygon methods. Its Points property contains a list of the point coordinates in pixels that the polygon should connect. The coordinates can be separated by spaces, commas, or a mix of the two. To make reading the point data easier, I separate points with spaces and the X and Y coordinates in a point with a comma.

The following code shows how the program ShapeControls draws its polygon. The Points property’s value is split across two lines in this code, but in the program it’s all on one line.

```xml
<Polygon Name="Polygon1" Stroke="Red" StrokeThickness="5"
    Points="50.00,0.00 64.69,29.77 97.55,34.55 73.78,57.73 79.39,90.45 50.00,75.00
    20.61,90.45 26.22,57.73 2.45,34.55 35.31,29.77" Margin="110,97,123,56"
    Fill="Blue"/>
```

**Polyline**

The Polyline control connects a series of points with lines. It is similar to the Polygon control except it does not connect the last point to the first point. Because it is not a closed shape, the Polyline also does not fill its shape.

The following code shows how the program ShapeControls draws its polyline. The Points property’s value is split across two lines in this code, but in the program it’s all on one line.

```xml
<Polyline Margin="0,97,33,75" Name="Polyline1" Stroke="Orange" StrokeThickness="5"
    HorizontalAlignment="Right" Width="80" Points="34,43 34,34 26,34 24,47 30,53
    47,47 45,32 29,19 14,36 14,54 30,70 54,61 62,38 47,14"/>
```

**Path**

The Path control draws a series of connected lines and curves. You can either define a Path with the path mini-language or by building a series of objects nested within the Path object’s XAML code.

Table 4 describes the path mini-language’s most useful commands.
Table 4: The Path Control’s Mini-Language Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>M x,y</td>
<td>Moves to the point (x, y).</td>
</tr>
<tr>
<td>L x,y</td>
<td>Draws a line to the point (x, y).</td>
</tr>
<tr>
<td>H x</td>
<td>Draws a horizontal line to the X coordinate x.</td>
</tr>
<tr>
<td>V y</td>
<td>Draws a vertical line to the Y coordinate y.</td>
</tr>
<tr>
<td>C x1,y1 x2,y2</td>
<td>Draws a cubic Bezier to the end point (x3, y3) using the control points</td>
</tr>
<tr>
<td></td>
<td>(x1, y1) and (x2, y2).</td>
</tr>
<tr>
<td>Q x1,y1 x2,y2</td>
<td>Draws a quadratic Bezier curve to the end point (x2, y2) using the control point (x1, y1).</td>
</tr>
<tr>
<td>S x1,y1 x2,y2</td>
<td>Draws a smooth cubic Bezier curve to the end point (x2, y2). The first control point is calculated as a reflection of the second control point in the previous command. The second control point is (x1, y1).</td>
</tr>
<tr>
<td>T x1,y1 x2,y2</td>
<td>Draws a smooth quadratic Bezier curve to the end point (x2, y2). The first control point is calculated as a reflection of the second control point in the previous command. The second control point is (x1, y1).</td>
</tr>
<tr>
<td>A xr,yr ang</td>
<td>Draws an elliptical arc from the current point to (x, y). The parameters xr and yr give the ellipse’s X and Y radii, ang gives the ellipse’s angle of rotation in degrees, is_big should be 0 (False) to indicate that the sweep angle should be less than 180 degrees or 1 to indicate a big angle, and dir should be 1 for a positive sweep direction or 0 otherwise.</td>
</tr>
<tr>
<td>Z</td>
<td>Closes the figure by connecting the current point to the starting point.</td>
</tr>
</tbody>
</table>

Each of the command characters (M, L, H, etc.) can be in uppercase to indicate absolute coordinates or lowercase to indicate coordinates relative to the current position.

The following code shows how the program ShapeControls draws the red path to the right of the image in Figure 23:

```xml
<Path HorizontalAlignment="Right" Margin="0,12,81,0" Name="Path2" Stroke="Red" StrokeThickness="3" Width="63" Data="M 0,0 C 30,90 40,-40 50,70" Height="73" VerticalAlignment="Top" />
```

55
The second method of building a Path control is to place other objects inside its XAML tags. The following code shows how the program ShapeControls draws the green path in the upper-right corner of Figure 23:

```xml
<Path Height="63" Margin="201,12,0,0" Name="Path1" Stroke="Green"
StrokeThickness="3" VerticalAlignment="Top" HorizontalAlignment="Left" Width="62">
  <Path.Data>
    <PathGeometry>
      <PathFigure>
        <BezierSegment Point1="0,0" Point2="50,0" Point3="50,20" />
        <BezierSegment Point1="50,40" Point2="30,10" Point3="0,50" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

In this code, the Path element contains a Path.Data element to hold its data. That element can hold a single instance of one of several geometry elements including EllipseGeometry, RectangleGeometry, and LineGeometry. Alternatively, it can hold a GeometryGroup, which can contain one or more other geometries.

In this example, the Path.Data element holds a PathGeometry element containing a PathFigure that holds the BezierSegments that actually produce results.

Whether you use the path mini-language or an object model to define a Path is up to your preference. In many cases, the path mini-language is easier to write and understand.

**Brushes**

The ideas behind filled areas in WPF are similar to those behind the brushes you use to fill areas in Windows Forms applications. Many of the WPF controls allow you to specify a Fill property that determines the control’s fill color. For example, the following code draws an ellipse and fills it with the color lime:

```xml
<Ellipse HorizontalAlignment="Left" Margin="13,97,0,0" Name="Ellipse1"
Stroke="Cyan" StrokeThickness="10" Width="86" Height="147"
VerticalAlignment="Top" Fill="Lime"/>
```

WPF provides more complicated brushes with classes such as LinearGradientBrush, RadialBrush, DrawingBrush, and ImageBrush. Because these classes are moderately complex, they cannot be represented in XAML by simple text values; thus, you cannot use them directly in a control’s XAML tag. Instead, you must nest them within the control’s element.

The Brushes example program, which is shown in Figure 24, demonstrates these types of brushes. The rectangles from left to right in the top row of Figure 24 are filled with a LinearGradientBrush, a RadialBrush, and a DrawingBrush. The bottom rectangle is filled with an ImageBrush.
The following sections describe each of these kinds of brushes in greater detail.

**LinearGradientBrush**

The `LinearGradientBrush` fills an area with colors shading smoothly from one to another across the area. Its `StartPoint` and `EndPoint` properties determine where the color gradient starts and ends. These properties are measured as fractions of the size of the control being filled with (0, 0) in the control’s upper-left corner and (1, 1) in the lower right.

The `LinearGradientBrush` XAML element should contain `GradientStop` objects that define the colors the brush should use and their positions within the gradient. This object’s `Color` property specifies a color, and its `Offset` property determines the position within the gradient where that color should appear.

The program `Brushes` uses the following code to draw the rectangle in the upper-left corner of Figure 24. The `StartPoint` and `EndPoint` properties make the gradient start in the control’s upper-left corner and end in the lower right. The `GradientStop` objects make the gradient start with red, shade to green halfway through, and then shade to blue at the end.

```xml
<Rectangle Height="82" HorizontalAlignment="Left" Margin="0,1,0,0" Name="Rectangle1" Strokes="Black" VerticalAlignment="Top" Width="99">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Red" Offset="0.0" />
      <GradientStop Color="Green" Offset="0.5" />
      <GradientStop Color="Blue" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```
**RadialGradientBrush**

The `RadialGradientBrush` shades colors radially from a defined center point outward. Its `Center` property determines where the center point lies. The point’s coordinates are measured as fractions of the size of the control being filled with (0, 0) in the control’s upper-left corner and (1, 1) in the lower right.

The `RadialGradientBrush` XAML element should contain `GradientStop` objects that define the colors the brush should use and their positions within the gradient.

The program `Brushes` uses the following code to draw the rectangle in the upper center. The `Center` property makes the gradient start in the control’s center, (0.5, 0.5) in the control’s internal coordinate system. The `GradientStop` objects make the gradient start with blue, shade to white halfway through, and then shade to lime 90 percent of the way through. Beyond that point, the area is filled with lime.

```xml
<Rectangle Height="82" HorizontalAlignment="Left" Margin="111,1,0,0" Name="Rectangle4" Stroke="Black" VerticalAlignment="Top" Width="99">
    <Rectangle.Fill>
        <RadialGradientBrush Center="0.5,0.5">
            <GradientStop Color="Blue" Offset="0" />
            <GradientStop Color="White" Offset="0.5" />
            <GradientStop Color="Lime" Offset="0.9" />
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

**DrawingBrush**

The `DrawingBrush` can contain:

- `GeometryDrawing` to draw a shape.
- `ImageDrawing` to draw an image.
- `GlyphRunDrawing` to draw text.
- `VideoDrawing` to play an audio or video file.
- `DrawingGroup` to draw a group of drawings.

The `DrawingBrush` defines a `viewport` that determines the basic unit of drawing, and it then tiles the area with that viewport. By default, the viewport fills the control containing the brush so that you’ll see only one copy of the drawing. Normally you would change this behavior by setting the viewport size to something smaller. You can set the `ViewportUnits` property to `Absolute` to indicate that the viewport’s size is measured in pixels.

The program `Brushes` uses the following code to draw the rectangle in its upper-right corner:

```xml
<Rectangle Margin="222,1,0,0" Name="Rectangle2" Stroke="Black" HorizontalAlignment="Left" Width="99" Height="82" VerticalAlignment="Top">
    <Rectangle.Fill>
        <DrawingBrush Stretch="None" TileMode="Tile" Viewport="0,0,20,20" ViewportUnits="Absolute">
            <DrawingBrush.Drawing>
            </DrawingBrush.Drawing>
        </DrawingBrush>
    </Rectangle.Fill>
</Rectangle>
```
The code sets the brush’s Viewport and ViewportUnits properties to indicate that the viewport should be 20 pixels wide and 20 pixels tall. It sets the brush’s TileMode property to Tile so that the brush repeats its image across the area it fills.

The brush’s XAML element contains a DrawingGroup, so it can hold more than one other drawing element. The DrawingGroup contains two GeometryDrawing elements that draw a large blue ellipse and a smaller black ellipse.

**ImageBrush**

The ImageBrush fills an area with an image. Like the DrawingBrush, this brush defines a viewport that it uses to fill its area. You can use the brush’s Viewport and ViewportUnits properties to determine the viewport’s size. The TileMode property determines how the brush tiles its area.

The brush’s ImageSource property determines where the brush gets its image, and it can be a URL or the name of an embedded resource.

An ImageSource URL is the address of an image file available on the Web as in **www.vb-helper.com/vb_prog_ref.jpg**.

To make the control load an embedded resource, add an image file to the application’s resources and set the image’s Build Action property to Resource. Then if the file’s name is smile.bmp, you can specify the brush’s Source property as in smile.bmp.

The program Brushes uses the following code to draw the rectangle at the bottom in Figure 24. The ImageBrush has a 73-by-73 pixel viewport and loads its image from the smile.bmp embedded resource.
Section 7 Wrap-up

WPF provides many new controls including several that display graphics. By using those controls, you can construct complex graphics declaratively at design time instead of generating them at run time with code. Some of the WPF designer tools are a bit weak (e.g., they won’t let you define complex brushes), but with some extra effort you can implement additional features in XAML.

The next section describes a particularly powerful tool provided by WPF: the FlowDocument control. This control can display complex graphics including drawing controls such as Rectangle, Ellipse, and Polygon, as well as other controls such as labels, textboxes, buttons, and so forth.

Section 8: Building FlowDocuments

The WPF FlowDocument class represents a document that contains elements that display WPF controls, text, and other graphics. Special objects within the document can arrange elements as lists, tables, paragraphs, and so forth. The FlowDocument object automatically rearranges the elements as needed and as space permits, much as a web page rearranges the text, images, and other items it contains.

This is a very different kind of interface from those produced by simple Windows Forms and WPF pages. The result is appropriate, when displaying as much content as possible is more important than keeping the content in specific relative positions.

To use a FlowDocument in WPF, you put other objects inside its XAML tags. The following list describes the kinds of objects that a FlowDocument can hold directly. You can place other objects inside these top-level objects:

- BlockUIContainer — Contains controls such as TextBoxes and Label.
- List — Contains a list of items.
- Paragraph — Contains elements grouped in a paragraph.
- Section — Contains a higher-level grouping usually containing multiple paragraphs or other objects.
- Table — Contains items displayed in rows and columns.

The following sections give a brief overview of these kinds of objects.
**BlockUIContainer**

The `BlockUIContainer` object contains user interface controls such as `TextBox` and `Label`. Normally you would use this to place these controls inside some larger object such as a `Paragraph`. In other words, a `Paragraph` would contain a `BlockUIContainer` that would contain a `Button`, `Grid`, or `GroupBox`.

While this control can hold only a single Child control, this isn’t a big restriction since that Child can be a container. For example, you could place a `Panel` or `Grid` inside the `BlockUIContainer` and the Child can hold other controls.

**List**

A `List` displays a series of items, optionally with bullets or numbers next to them.

The `MarkerStyle` property determines whether the `List` is numbered or bulleted and can take the values `Box`, `Circle`, `Decimal`, `Disc`, `LowerLatin`, `LowerRoman`, `None`, `Square`, `UpperLatin`, and `UpperRoman`. The `StartIndex` property determines the first number for numbered lists (e.g., so you can continue a previous list).

The `MarkerOffset` property determines how far the markers are from the content area, and `TextAlignment` determines how text is aligned.

**Paragraph**

A `Paragraph` contains elements that should be grouped as a paragraph. Normally a `Paragraph` contains plaintext, but it can also hold formatting elements such as `Bold`, `Italic`, and `Underline`.

A `Paragraph` can also hold more structural elements such as `InlineUIContainer` (similar to `BlockUIContainer` but inside a `Paragraph`), `Figure` (a positioned object that the text flows around), and `Floater` (similar to `Figure` except it is not positioned and the `FlowDocument` will float it to a convenient position if necessary).

**Section**

A `Section` contains a higher-level grouping of objects, usually containing multiple paragraphs or other objects. Its main purpose is to group other block elements. A `Section` can contain `BlockUIContainer`, `List`, `Paragraph`, `Section`, and `Table` elements.

**Table**

A `Table` contains items displayed in rows and columns. At the lower levels, `TableRow` objects hold `TableCell` objects that contain the items to display. Building a table can be a lot of work, but with enough effort, you can build pretty complicated tables with cells that span multiple columns, rows, or cells with different backgrounds and text alignments, and so forth.
The following code builds a simple Table:

```xml
<Table>
  <Table.Columns>
    <TableColumn />
    <TableColumn />
  </Table.Columns>
  <TableRowGroup>
    <TableRow Background="DarkGreen" Foreground="Yellow">
      <TableCell ColumnSpan="2" TextAlignment="Center">
        <Paragraph FontSize="16" FontWeight="700">
          Some Table Components
        </Paragraph>
      </TableCell>
    </TableRow>
    <TableRow Background="Green" Foreground="White">
      <TableCell>
        <Paragraph FontSize="14">Component</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph FontSize="14">Purpose</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow>
      <TableCell>
        <Paragraph>Table.Columns</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Holds TableColumn objects that define the columns.</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow>
      <TableCell>
        <Paragraph>TableColumn</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Defines a column.</Paragraph>
      </TableCell>
    </TableRow>
  </TableRowGroup>
</Table>
```

The Table's first Child is a `Table.Columns` element that contains `TableColumn` elements that define the Table's two columns. The code doesn't define widths for the columns; thus, they share the available area.

The rest of the Table's contents are contained in a `TableRowGroup`. That element contains `TableRow` objects that hold `TableCell` objects.

The first `TableRow` holds one `TableCell` that spans both of the Table's columns and gives the Table an overall title row. `TableCell` properties give the row yellow text with a dark green background. `TableCell` properties make the cell use a large, bold font.
The second TableRow holds two TableCell objects that give column headers. Properties make these cells use white text with a green background and a large font.

The remaining TableRow objects hold detail in ordinary cells with default colors and font sizes.

The FlowDocumentDemo example program, which is shown in Figure 25, contains a Grid that holds a FlowDocumentReader. That object contains a FlowDocument object that demonstrates many basic kinds of Children. For example, it contains (either directly or indirectly) BlockUIContainer, List, Paragraph, Section, and Table objects. It also demonstrates Figure and Floater objects.

![FlowDocumentDemo](image)

**Figure 25:** The FlowDocumentDemo example program displaying a FlowDocument object containing a variety of objects.

The code used by the program FlowDocumentDemo is not too complicated, but it is very long (about six pages); thus, it’s not shown here. Download the example program and see for yourself.

**Section 8 Wrap-up**

A FlowDocument flows the objects it contains to fit the space available; thus, you only have indirect control over where things are positioned. If your goal is to display a lot of information in a format similar to a newspaper, newsletter, or web page, a FlowDocument can give you a lot of flexibility. It also allows you to build complicated documents declaratively at design time.
The **FixedDocument** class is similar to **FlowDocument** except it keeps objects in the exact positions where you place them. If a **FlowDocument** is similar to a web page displaying objects that move as the browser is resized, then a **FixedDocument** is similar to an Adobe Acrobat PDF document displaying objects at specific positions.

**Conclusion**

C# provides a huge number of graphics tools that you can use to solve all sorts of problems. By using **Graphics** objects, pens, and brushes, you can draw and fill all sorts of shapes. You can use stock pens and brushes or create your own with hatch patterns, gradient fills, textures, custom dash patterns and end caps, and longitudinal stripes.

The **Graphics** class’s **DrawString** method lets you draw text formatted and wrapped to fit a particular area, optionally with ellipses. The **Bitmap** class provides methods for getting and setting the values of every pixel in an image.

Transformations add a whole new level of power to any graphics code. By using transformations, you can stretch, rotate, and translate images to reuse code that draws standard pictures. Transformations also allow you to produce results that are otherwise impossible such as stretched text.

C#’s printing capabilities are awkward at first, requiring you to respond to events requesting pages rather than generating pages explicitly, but once you get the hang of it, printing is fairly easy. The fact that you use a **Graphics** object to draw on the screen or printout makes printing a lot easier. The fact that you use a **PrintDocument** both for printing and generating previews makes providing previews practically trivial.

Finally, WPF and the **FlowDocument** control give you a whole new arsenal of graphics tools. They let you generate graphics declaratively at design time instead of writing complicated code that only produces results when the program executes.

If you have questions or comments about the material in this Wrox Blox, please e-mail me at RodStephens@vb-helper.com and let me know. If you get stuck on a graphical technique, let me know, and I may be able to point you in the right direction. Finally, if you produce some really cool and amazing images, send them to me so I can post them and let others see what’s possible with C#.
About Rod Stephens

Rod Stephens is a consultant and author who has written more than 18 books and 250 magazine articles, mostly about Visual Basic. Currently he is a regular contributor of C# and Visual Basic articles at DevX.com (www.devx.com). During his career he has worked on an eclectic assortment of applications spanning such fields as repair dispatch, fuel tax tracking, professional football training, wastewater treatment, geographic mapping, and ticket sales. His VB Helper web site (www.vb-helper.com) receives more than 7 million hits per month and provides three newsletters and thousands of tips, tricks, and examples for Visual Basic programmers.
Providing programmers and developers of all skill levels with a comprehensive tutorial and reference to Visual Basic (VB) 2008, Microsoft MVP Rod Stephens presents a broad, solid understanding of essential topics on the latest version of VB. He explains the forms, controls, and other objects that VB furnishes for building applications in a modern windows environment. Plus, he examines the powerful development environment that makes VB such a productive language, and he delves into the VB language itself to show you how to use it to perform an array of important development tasks.

New examples and extensively revised and retested code that complies with the 2008 release help you obtain a firm understanding of VB 2008. Extensive appendixes prove to be particularly useful to help you translate from the languages you already know into the corresponding VB syntax. Ultimately, you’ll find coverage of the technologies you need in order to build sophisticated applications with VB 2008.

What you will learn from this book:
- Extension methods for adding new features to existing classes
- How to select and use Windows Forms Controls for a specific purpose
- Tips for using subroutines and functions to break a program into manageable pieces
- Techniques for error handling and debugging
- Various important classes and objects to use when building an application
- How to use the graphics device interface routines to draw images in VB
- Ways an application interacts with its environment

Visual Basic 2008 Programmer’s Reference
By ROD STEPHENS
ISBN: 978-0-470-18262-8

Visit us at www.wrox.com