Some of the most interesting and fun parts of a programming language are its graphics elements. In general, graphics fall into two major categories: vector and bitmap. Vector graphics are images generated by graphics methods such as DrawLine and DrawEllipse. The drawing you create is based on mathematical descriptions of the various shapes. Bitmap graphics are images made up of pixels arranged in rows and columns. Each pixel is represented by a Long numeric value, which is the pixel’s color. The difference between vector and bitmap graphics is that vector graphics aren’t tied to a specific monitor resolution; that is, they can be displayed at various resolutions. Bitmap graphics, on the other hand, have a fixed resolution. An image that is 1,024 pixels wide and 768 pixels tall has that specific resolution. If you attempt to use that image to fill a monitor that’s 1,280 pixels wide and 1,024 pixels tall, you’ll have to repeat some pixels. Image-processing software can interpolate between pixels, but when you blow up a bitmap, you see its block-like structure.

Displaying and Sizing Images

The primary control for displaying images is the PictureBox control. To load an image to a PictureBox control, locate the Image property in the Properties window and click the button with the ellipsis next to it. The Select Resource dialog box will appear, in which you can select the image to be displayed (see Figure 4.1). The image, along with every other image or icon you use in the same project, is stored in the Resources folder under the project’s folder. As a result, you don’t have to distribute the image with your application; it will be included in the setup file that the installer will create for your application.

After the image is loaded, you must make sure that it fills the available space. The PictureBox control exposes theSizeMode property, which determines how the image will be sized and aligned on the control. TheSizeMode property can be set to a member of the PictureBoxSizeMode enumeration: AutoSize, CenterImage, Normal, StretchImage, and Zoom. Its default setting is Normal, and in this mode the control displays the image at its normal magnification. If the image is larger than the control, part of the image will be invisible. If the image is smaller than the control, part of the control will be empty. In this case, you can set theSizeMode property to CenterImage to center the image on the control.

The StretchImage setting resizes the image so that it fills the control. If the control’s aspect ratio isn’t the same as the aspect ratio of the image, the image will be distorted in the process. If you want to use the StretchImage setting, you must also resize one of the dimensions of the control, so that the image will be properly resized. You’ll see how to do this shortly. The
AutoSize setting causes the control to be resized according to the image’s dimensions. This is not the most convenient setting because the control might cover other controls on the form.

**Figure 4.1**
Adding image resources to a project through the Select Resource dialog box.

The Zoom setting of the SizeMode property resizes the image without distorting its aspect ratio. In this mode, the control attempts to resize the image as well as it can in the given area while maintaining its aspect ratio. If the image’s aspect ratio is different from the aspect ratio of the control, this setting will fill the control vertically or horizontally and will center the image in the other direction.

Figure 4.2 shows a PictureBox control with an image in four of the five settings (the AutoSize mode, which isn’t shown in the figure, stretches the PictureBox control to the size of the image) Notice that the Zoom mode filled the PictureBox vertically, but left a margin on either side of the image to avoid distortion of the image’s aspect ratio.

**Designing a Scrolling PictureBox**

A problem with the PictureBox control is that it doesn’t provide an AutoScroll property; thus you can’t display a large image at its original resolution and scroll any part of it into view at runtime. A scrollable PictureBox would be highly desirable in many applications (images are so common in many types of applications today), but because the control doesn’t support this functionality, here’s the next best thing you can do:

1. Place a Panel control on the form and set its AutoSize property to True. Set its AutoScroll property to True also, so that the appropriate scroll bars will appear automatically as soon as the control’s contents exceed its dimensions. Finally, set its Dock property to Fill, so that it will cover the entire form.

2. Place a PictureBox control on the Panel control and set its SizeMode property to AutoSize. We want the PictureBox control to be sized according to the image it contains.

3. Finally, assign a large image to the PictureBox control (any of the images in the folder Pictures/Sample Pictures will do). As soon as you assign the image to the control, the necessary scroll bars will be displayed and you can scroll any part of the image into view, even at design time.

Open the Scrolling PictureBox project, shown in the following figure, and experiment with large images. The sample project’s main form contains a menu and a status bar, which remain in place.
as you scroll the PictureBox control with the image in the Panel control. The menu contains commands to zoom in and out of the image as well as commands to rotate the image.

I've also added a few statements to display the coordinates of the upper-left corner of the visible section of the image and the current zoom on the form's status bar. Every time the Panel's contents are scrolled, the Scroll event takes place. I'm using this event handler's arguments to read the horizontal and vertical displacement of the image and print them with the following statements:

Private Sub Panel1_Scroll(ByVal sender As Object, ByVal e As System.Windows.Forms.ScrollEventArgs) Handles Panel1.Scroll
    If e.ScrollOrientation = ScrollOrientation.HorizontalScroll Then
        X = e.NewValue
    Else
        Y = e.NewValue
    End If
    toolStripStatusLabel1.Text = 
        '
        ', X.ToString & ', Y: ' & Y.ToString & ']
    End Sub

**Figure 4.2**
The settings of the SizeMode property
Drawing with GDI+

You have seen the basics of displaying images on your forms; now let’s move on to some real graphics operations, namely how to create your own graphics with the Framework. Windows graphics are based on a graphics engine, known as GDI. GDI, which stands for Graphics Design Interface, is a collection of classes that enable you to create graphics, text, and images. The most recent version on GDI is called GDI+.

One of the basic characteristics of GDI is that it’s stateless. This means that each graphics operation is totally independent of the previous one and can’t affect the following one. To draw a line, you must specify a Pen object and the two endpoints of the line. You must do the same for the next line you’ll draw. You can’t assume that the second line will use the same pen or that it will start at the point where the previous line ended. There isn’t even a default font for text-drawing methods. Every time you draw some text, you must specify the font in which the text will be rendered, as well as the Brush object that will be used to draw the text.

The GDI+ classes reside in the following namespaces, and you must import one or more of them in your projects: System.Drawing, System.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text. This tutorial explores all three aspects of GDI+ — namely vector drawing, imaging, and typography.

Before you start drawing, you must select the surface you want to draw on, the types of shapes you want to draw, and the instrument you’ll use to draw them. The surface on which you can draw is a Graphics object, which is your canvas, and it’s the control’s Graphics property. Most controls expose a Graphics property, but most applications draw on either forms or PictureBox controls. The Graphics property is an object that exposes numerous methods for drawing basic (and not-so-basic) shapes.

The next step is to decide which instrument you’ll use to draw. There are two major drawing instruments: the Pen object and the Brush object. You use pens to draw stroked shapes (lines, rectangles, curves) and brushes to draw filled shapes (any area enclosed by a shape, including text). The main characteristics of the Pen object are its color and its width (the size of the trace left by the pen). The main characteristic of the Brush object is the color or pattern with which it fills the shape. An interesting variation of the Brush object is the gradient brush, which changes color as it moves from one point of the shape you want to fill to another. You can start filling a shape with red in the middle and specify that as you move toward the edges of the shape, the fill color fades to yellow.

After you have specified the drawing surface and the drawing instrument, you draw the actual shape by calling the appropriate method of the Graphics object. To draw lines, call the DrawLine method of the Graphics object; to draw text, call the DrawString method of the same object. There are many drawing methods, as well as other methods that support the main drawing methods, and they’re all discussed later in this tutorial. Here are the statements to draw a line on the form:

```vbnet
Dim redPen As Pen = New Pen(Color.Red, 2)
Dim point1 As Point = New Point(10,10)
Dim point2 As Point = New Point(120,180)
Me.CreateGraphics.DrawLine(redPen, point1, point2)
```

The first statement declares a new Pen object, which is initialized to draw in red with a width of 2 pixels. The following two statements declare and initialize two points, which are the line’s starting and ending points. The coordinates are expressed in pixels, and the origin is at the form’s top-left corner. The last statement draws the line by calling the DrawLine method. The expression Me.CreateGraphics retrieves the Graphics object of the form, which exposes
all the drawing methods, including the DrawLine method. You can also create a new Graphics object and associate it with the form:

```vba
' set up a pen and the two endpoints as before
Dim G As Graphics
G = Me.CreateGraphics
G.DrawLine(redPen, point1, point2)
```

The DrawLine method accepts as an argument the pen it will use to draw and the line's starting and ending points. I have used two Point objects to make the code easier to read. The DrawLine method, like all other drawing methods, is heavily overloaded. You can also omit the declarations of the various objects and initialize them in the same statement that draws the line:

```vba
Me.CreateGraphics.DrawLine(New Pen(Color.Red, 2), New Point(10, 10), New Point(120, 180))
```

All coordinates are expressed by default in pixels. It's possible to specify coordinates in different units and let GDI+ convert them to pixels before drawing. For now, we'll use pixels, which are quite appropriate for simple objects. After you familiarize yourself with the drawing methods, you can specify different coordinate systems. For more information, see the discussion of the PageUnit property of the Graphics object in the following section.

**The Basic Drawing Objects**

This is a good point to introduce some of the objects we'll be using all the time when drawing. No matter what you draw or which drawing instrument you use, one or more of the objects discussed in this section will be required.

**The Graphics Object**

The Graphics object is the drawing surface — your canvas. All the controls you can draw on expose a Graphics property, which is an object, and you can retrieve it with the CreateGraphics method. Conversely, if an object doesn't expose the CreateGraphics method, you can't draw on its surface. It goes without saying that the PictureBox control exposes a Graphics property, but so does the TextBox control, as well as many controls you wouldn't expect. It's not recommended that you draw on a TextBox control, of course, unless you're coding a peculiar application. Bear in mind that anything you draw on the TextBox control will disappear as you start typing. You must first place the text on the control and then draw on its surface — or make the control read-only.

The Graphics object exposes all the methods and properties you will use to create graphics on the control. If you enter the string `Me.CreateGraphics` and a period, you will see a list of the members of the Graphics object in a drop-down list.

Start by declaring a variable of the Graphics type and initialize it to the Graphics object returned by the control's CreateGraphics method:

```vba
Dim G As Graphics
G = PictureBox1.CreateGraphics
```

At this point, you're ready to start drawing on the `PictureBox1` control with the methods presented in the following sections. In essence, the CreateGraphics method returns the drawing surface of the control or form on which you wish to draw.
WHEN DO WE INITIALIZE A GRAPHICS OBJECT?

The Graphics object is initialized to the control’s drawing surface at the moment you create it. If the form is resized at runtime, the Graphics object won’t change, and part of the drawing surface might not be available for drawing. If you create a Graphics object to represent a form in the form’s Load event handler and the form is resized at runtime, the drawing methods you apply to the Graphics object will take effect in part of the form. The most appropriate event for initializing the Graphics object and inserting the painting code is the form’s Paint event. This event is fired when the form must be redrawn — when the form is uncovered or resized. Insert your drawing code there and create a Graphics object in the Paint event handler. Then draw on the Graphics object and release it when you’re done.

The Graphics object exposes the following basic properties, in addition to the drawing methods discussed in the following sections.

**DpiX, DpiY**

These two properties return the horizontal and vertical resolutions of the drawing surface, respectively. Resolution is expressed in pixels per inch (or dots per inch, if the drawing surface is your printer). On an average monitor, these two properties return a resolution of 96 dots per inch (dpi).

**PageUnit**

This property determines the units in which you want to express the coordinates on the Graphics object; its value can be a member of the GraphicsUnit enumeration (Table 4.1). If you set the PageUnit property to World, you must also set the PageScale property to a scaling factor that will be used to convert world units to pixels.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>The unit is 1/75 of an inch.</td>
</tr>
<tr>
<td>Document</td>
<td>The unit is 1/300 of an inch.</td>
</tr>
<tr>
<td>Inch</td>
<td>The unit is 1 inch.</td>
</tr>
<tr>
<td>Millimeter</td>
<td>The unit is 1 millimeter.</td>
</tr>
<tr>
<td>Pixel</td>
<td>The unit is 1 pixel (the default value).</td>
</tr>
<tr>
<td>Point</td>
<td>The unit is a printer’s point (1/72 of an inch).</td>
</tr>
<tr>
<td>World</td>
<td>The developer specifies the unit to be used.</td>
</tr>
</tbody>
</table>
**TextRenderingHint**

This property specifies how the Graphics object will render text; its value is one of the members of the TextRenderingHint enumeration: AntiAlias, AntiAliasGridFit, ClearTypeGridFit, SingleBitPerPixel, SingleBitPerPixelGridFit, and SystemDefault.

**SmoothingMode**

This property is similar to the TextRenderingHint, but it applies to shapes drawn with the Graphics object's drawing methods. Its value is one of the members of the SmoothingMode enumeration: AntiAlias, Default, HighQuality, HighSpeed, Invalid, and None.

Figure 4.3 shows the effect of the TextRenderingHint property on text. The anti-aliased text looks much better on the monitor, because anti-aliased text is smoother. The edges of the characters contain shades between the drawing and background colors. The ClearType setting has no effect on Cathode Ray Tube (CRT) monitors. You can see the difference only when you render text on Liquid Crystal Display (LCD) monitors, such as flat-panel or notebook monitors. Text in ClearType style looks best when rendered black on a white background. You won’t be able to see the differences among the various settings on the printed image, but you can open the TextRenderingHint project, which I used to create the figure, and examine how the TextRenderingHint property affects the rendering of the text. You can also capture the form by pressing Alt + PrtSc, paste it into Paint or your favorite image-processing application, and zoom into the details of the various characters.

*Figure 4.3*
The effect of the TextRenderingHint setting on the rendering of text

Many of the drawing methods of the Graphics object use some helper classes, such as the Point class that’s used to specify coordinates, the Color class that’s used to specify colors, and so on. I’ll go quickly through these classes, and then I’ll discuss the drawing methods in detail.
THE POINT CLASS

The Point class represents a point on the drawing surface and is expressed as a pair of \((x, y)\) coordinates. The \(x\)-coordinate is its horizontal distance from the origin, and the \(y\)-coordinate is its vertical distance from the origin. The origin is the point with coordinates \((0, 0)\), and this is the top-left corner of the drawing surface.

The constructor of the Point class is the following, where \(X\) and \(Y\) are the point’s horizontal and vertical distances from the origin:

\[
\text{Dim P1 As New Point}(X, Y)
\]

You can also set the \(X\) and \(Y\) properties of the \(P1\) variable. As you will see later, coordinates can be specified as single numbers, not integers (if you choose to use a coordinate system other than pixels). In this case, use the PointF class, which is identical to the Point class except that its coordinates are non-integers. \((F\) stands for floating-point, and floating-point numbers are represented by the Single or Double data type.)

THE rectangle CLASS

Another class that is often used in drawing is the Rectangle class. The Rectangle object is used to specify areas on the drawing surface. Its constructor accepts as arguments the coordinates of the rectangle’s top-left corner and its dimensions:

\[
\text{Dim box As Rectangle}
\text{box = New Rectangle}(X, Y, width, height)
\]

The following statement creates a rectangle whose top-left corner is 1 pixel to the right and 1 pixel down from the origin, and its dimensions are 100 by 20 pixels:

\[
\text{box = New Rectangle}(1, 1, 100, 20)
\]

The \(box\) variable represents a rectangle, but it doesn’t generate any output on the monitor. If you want to draw the rectangle, you can pass it as argument to the DrawRectangle or FillRectangle method, depending on whether you want to draw the outline of the rectangle or a filled rectangle.

Another form of the Rectangle constructor uses a Point and a Size object to specify the location and dimensions of the rectangle:

\[
\text{box = New Rectangle}(\text{point}, \text{size})
\]

The \(\text{point}\) argument is a Point object that represents the coordinates of the rectangle’s upper-left corner. To create the same Rectangle object as in the preceding example with this form of the constructor, use the following statement:

\[
\text{Dim P As New Point}(1, 1)
\text{Dim S As New Size}(100, 20)
\text{box = New Rectangle}(P, S)
\]

Both sets of statements create a rectangle that extends from point \((1, 1)\) to the point \((1 + 100, 1 + 20)\) or \((101, 21)\), in the same manner as the ones shown in Figure 4.4. Alternatively, you can declare a Rectangle object and then set its \(X, Y, \text{Width}, \) and \(\text{Height}\) properties.
Figure 4.4
Specifying rectangles with the coordinates of their top-left corner and their dimensions

**THE SIZE CLASS**

The Size class represents the dimensions of a rectangle; it's similar to a Rectangle object, but it doesn't have an origin, just dimensions. To create a new Size object, use the following constructor:

```csharp
Dim S1 As New Size(100, 400)
```

If you want to specify coordinates as fractional numbers, use the SizeF class, which is identical to the Size class except that its dimensions are nonintegers.

**THE COLOR CLASS**

The Color class represents colors, and there are many ways to specify a color. We'll discuss the Color class in more detail later in this tutorial. In the meantime, you can specify colors by name. Declare a variable of the Color type and initialize it to one of the named colors exposed as properties of the Color class:

```csharp
Dim myColor As Color
myColor = Color.Azure
```

The 128 color names of the Color class will appear in the IntelliSense box as soon as you enter the period following the keyword `Color`. You can also use the `FromARGB` method, which creates a new color from its basic color components (the Red, Green, and Blue components).

**THE FONT CLASS**

The Font class represents fonts, which are used when rendering strings via the `DrawString` method. To specify a font, you must create a new Font object; set its family name, size, and style; and then pass it as argument to the `DrawString` method. Alternatively, you can prompt the user for a font via the Font common dialog box and use the object returned by the dialog box's `Font` property as an argument with the `DrawString` method. To create a new Font object, use a statement like the following:

```csharp
Dim drawFont As New Font("Verdana", 12, FontStyle.Bold)
```

The Font constructor has 13 forms in all. Two of the simpler forms of the constructor, which allow you to specify the size and the style of the font, are shown in the following code lines,
where size is an integer and style is a member of the FontStyle enumeration (Bold, Italic, Regular, Strikeout, and Underline):

```vbnet
dim drawfont as new font(name, size)
dim drawfont as new font(name, size, style)
```

To specify multiple styles, combine them with the OR operator:

```vbnet
fontstyle.bold or fontstyle.italic
```

You can also initialize a `Font` variable to an existing font. The following statement creates a `Font` object and initializes it to the current font of the form:

```vbnet
dim textfont as new font
textfont = me.font
```

**THE PEN CLASS**

The `Pen` class represents virtual pens, which you use to draw on the `Graphics` object’s surface. To construct a new `Pen` object, you must specify the pen’s color and width in pixels. The following statements declare three `Pen` objects with the same color and different widths:

```vbnet
dim thinpen, mediumpen, thickpen as pen
thinpen = new pen(color.black, 1)
mediumpen = new pen(color.black, 3)
thickpen = new pen(color.black, 5)
```

If you omit the second argument, a pen with a width of a single pixel will be created by default. Another form of the `Pen` object’s constructor allows you to specify a brush instead of a color, as follows, where `brush` is a `Brush` object (discussed later in this tutorial):

```vbnet
dim patternpen as pen
patternpen = new pen(brush, width)
```

The quickest method of creating a new `Pen` object is to use the built-in `Pens` collection, which creates a `Pen` with a width of 1 pixel and the color you specify. The following statement can appear anywhere a `Pen` object is required and will draw shapes in blue color:

```vbnet
pens.blue
```

The `Pen` object exposes these properties:

**Alignment**

Determines the alignment of the `Pen`, and its value is one of the members of the `PenAlignment` enumeration: Center or Inset. When set to `Center`, the width of the pen is centered on the outline (half the width is inside the shape, and half is outside). When set to `Inset`, the entire width of the pen is inside the shape. The default value of this property is `PenAlignment.Center`.

**LineJoin**

Determines how two consecutive line segments will be joined. Its value is one of the members of the `LineJoin` enumeration: Bevel, Miter, MiterClipped, and Round.
StartCap, EndCap
Determines the caps at the two ends of a line segment, respectively. Their value is one of the members of the LineCap enumeration: Round, Square, Flat, Diamond, and so on.

DashCap
Determines the caps to be used at the beginning and end of a dashed line. Its value is one of the members of the DashCap enumeration: Flat, Round, and Triangle.

DashStyle
Determines the style of the dashed lines drawn with the specific Pen. Its value is one of the members of the DashStyle enumeration (Solid, Dash, DashDot, DashDotDot, Dot, and Custom).

PenType
Determines the style of the Pen; its value is one of the members of the PenType enumeration: HatchFilled, LinearGradient, PathGradient, SolidColor, and TextureFill.

The Brush Class
The Brush class represents the instrument for filling shapes; you can create brushes that fill with a solid color, a pattern, or a bitmap. In reality, there’s no Brush object. The Brush class is actually an abstract class that is inherited by all the classes that implement a brush, but you can’t declare a variable of the Brush type in your code. The brush objects are shown in Table 4.2.

<table>
<thead>
<tr>
<th>BRUSH</th>
<th>FILL EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidBrush</td>
<td>Fills shapes with a solid color</td>
</tr>
<tr>
<td>HatchBrush</td>
<td>Fills shapes with a hatched pattern</td>
</tr>
<tr>
<td>LinearGradientBrush</td>
<td>Fills shapes with a linear gradient</td>
</tr>
<tr>
<td>PathGradientBrush</td>
<td>Fills shapes with a gradient that has one starting color and many ending colors</td>
</tr>
<tr>
<td>TextureBrush</td>
<td>Fills shapes with a bitmap</td>
</tr>
</tbody>
</table>

Solid Brushes
To fill a shape with a solid color, you must create a SolidBrush object with the following constructor, where brushColor is a color value, specified with the help of the Color object:

```vbnet
Dim sBrush As SolidBrush
sBrush = New SolidBrush(brushColor)
```

Every filled object you draw with the `sBrush` object will be filled with the color of the brush.
**Hatched Brushes**

To fill a shape with a hatch pattern, you must create a HatchBrush object with the following constructor:

```vbnet
Dim hBrush As HatchBrush
hBrush = New HatchBrush(hatchStyle, hatchColor, backColor)
```

The first argument is the style of the hatch, and it can have one of the values shown in Table 4.3 and in the following illustration. The HatchStyle enumeration has 54 members, so Table 4.3 shows only a few common patterns. You can fill shapes with plaid, spheres, waves, and a lot more patterns that aren’t listed here, but you will see their names in the IntelliSense box. The other two arguments are the colors to be used in the hatch. The hatch is a pattern of lines drawn on a background, and the two color arguments are the color of the hatch lines and the color of the background on which the hatch is drawn.

**Table 4.3:** The HatchStyle Enumeration

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackwardDiagonal</td>
<td>Diagonal lines from top-right to bottom-left</td>
</tr>
<tr>
<td>Cross</td>
<td>Vertical and horizontal crossing lines</td>
</tr>
<tr>
<td>DiagonalCross</td>
<td>Diagonally crossing lines</td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>Diagonal lines from top-left to bottom-right</td>
</tr>
<tr>
<td>Horizontal</td>
<td>Horizontal lines</td>
</tr>
<tr>
<td>Vertical</td>
<td>Vertical lines</td>
</tr>
</tbody>
</table>

**Gradient Brushes**

A gradient brush fills a shape with a specified gradient. The LinearGradientBrush fills a shape with a linear gradient, and the PathGradientBrush fills a shape with a gradient that has one starting color and one or more ending colors. Gradient brushes are discussed in detail later in this tutorial.

**Textured Brushes**

In addition to solid and hatched shapes, you can fill a shape with a texture by using a TextureBrush object. The texture is a bitmap that is tiled as needed to fill the shape. Textured brushes are used to create rather fancy graphics, and we won’t explore them in this book.

**The Path Class**

The Path class represents shapes made up of various drawing entities, such as lines, rectangles, and curves. You can combine as many of these drawing entities as you’d like and build a new entity, which is called a path. Paths are usually closed and filled with a color, a gradient, or a bitmap. You can create a path in several ways. The simplest method is to create a new Path object and then use one of the following methods to append the appropriate shape to the path:
AddArc   AddEllipse   AddPolygon
AddBezier AddLine     AddRectangle
AddCurve  AddPie      AddString

These methods add to the path the same shapes you can draw on the Graphics object with the methods discussed in the following section. There’s even an AddPath method, which adds an existing path to the current one. The syntax of the various methods that add shapes to a path is identical to the corresponding methods that draw. We simply omit the first argument (the Pen object) because all the shapes that make up a path will be rendered with the same pen. The following method draws an ellipse:

```
Me.CreateGraphics.DrawEllipse(mypen, 10, 30, 40, 50)
```

To add the same ellipse to a Path object, use the following statement:

```
Dim myPath As New Path
myPath.AddEllipse(10, 30, 40, 50)
```

To display the path, call the DrawPath method, passing a Pen and Path object as arguments:

```
Me.CreateGraphics.DrawPath(myPen, myPath)
```

Why combine shapes into paths instead of drawing individual shapes? After the shape has been defined, you can draw multiple instances of it, draw the same path with a different pen, or fill the path’s interior with a gradient. Paths are also used to create the ultimate type of gradient, the PathGradient.

**Drawing Shapes**

Now that we’ve covered the auxiliary drawing objects, we can look at the drawing methods of the Graphics class. Before getting into the details of the drawing methods, however, let’s write a simple application that draws a couple of simple shapes on a form. First, we must create a Graphics object with the following statements:

```
Dim G As Graphics
G = Me.CreateGraphics
```

Everything you’ll draw on the surface represented by the G object will appear on the form. Then, we must create a Pen object to draw with. The following statement creates a Pen object that’s 1 pixel wide and draws in blue:

```
Dim P As New Pen(Color.Blue)
```

We created the two basic objects for drawing: the drawing surface and the drawing instrument. Now we can draw shapes by calling the Graphics object’s drawing methods. The following statement will print a rectangle with its top-left corner near the top-left corner of the form (at a point that’s 10 pixels to the right and 10 pixels down from the form’s corner) and is 200 pixels wide and 150 pixels tall. These are the values you must pass to the DrawRectangle method as arguments, along with the Pen object that will be used to render the rectangle:

```
G.DrawRectangle(P, 10, 10, 200, 150)
```
Let’s add the two diagonals of the rectangle with the following statements:

```vbnet
G.DrawLine(P, 10, 10, 210, 160)
G.DrawLine(P, 210, 10, 10, 160)
```

We wrote all the statements to create a shape on the form, but where do we insert them? Let’s try a button. Start a new project, place a button on it, and then insert the statements of Listing 4.1 in the button’s Click event handler.

**Listing 4.1:** Drawing Simple Shapes

```vbnet
Private Sub Button1_Click(...) Handles Button1.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    Dim P As New Pen(Color.Blue)
    G.DrawLine(P, 10, 10, 210, 160)
    G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

Run the application and click the Draw On Graphics button. You will see the shape shown in Figure 4.5. This figure was created by the SimpleShapes sample application.

**Figure 4.5**
The output of Listing 4.1

**Persistent Drawing**

If you switch to the Visual Studio IDE or any other window, and then return to the form of the SimpleShapes application, you’ll see that the drawing has disappeared! The same will happen if you minimize the window and then restore it to its normal size. Everything you draw on the Graphics object is temporary. It doesn’t become part of the Graphics object and is visible only
while the control, or the form, need not be redrawn. As soon as the form is redrawn, the shapes disappear.

So, how do we make the output of the various drawing methods permanent on the form? Microsoft suggests placing all the graphics statements in the Paint event handler, which is triggered automatically when the form is redrawn. The Paint event handler passes the e argument, which (among other properties) exposes the form's Graphics object. You can create a Graphics object in the Paint event handler and then draw on this object.

Listing 4.2 is the Paint event handler that creates the shape shown in Figure 4.5 and refreshes the form every time it's totally or partially covered by another form. Delete the code in the button's Click event handler and insert the statements of Listing 4.2 into the Paint event's handler, as shown here. (Notice that the Graphics object is a property of the PaintEventArgs argument of the event handler.)

Listing 4.2: Drawing Simple Shapes in the Paint Event

```vbnet
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As System.Windows.Forms.PaintEventArgs) Handles Me.Paint
    Dim G As Graphics
    G = e.Graphics
    Dim P As New Pen(Color.Blue)
    G.DrawRectangle(P, 10, 10, 200, 150)
    G.DrawLine(P, 10, 10, 210, 160)
    G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

If you run the application now, it works like a charm. The shapes appear to be permanent, even though they’re redrawn every time you switch to the form. This technique is fine for a few graphics elements you want to place on the form to enhance its appearance. But many applications draw something on the form in response to user actions, such as the click of a button or a menu command. Using the Form's Paint event in a similar application is out of the question. The drawing isn’t always the same, and you must figure out from within your code which shapes you have to redraw at any given time. The solution is to make the drawing permanent on the Graphics object, so it won’t have to be redrawn every time the form is hidden or resized.

### FORCING REFRESHERS

A caveat of drawing from within the Paint event is that it isn’t fired when the form is resized by default. To force a refresh when the form is resized, you must insert the following statement in the form's Load event handler:

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

It is possible to make the graphics permanent by drawing not on the Graphics object, but directly on the control’s (or the form’s) bitmap. The Bitmap object contains the pixels that make
up the image and is very similar to the Image object. As you will see later in this tutorial, you can create a Bitmap object and assign it to an Image object. To create this “permanent” drawing surface, you must first create a Bitmap object that has the same dimensions as the form (or PictureBox control) on which you want to draw:

```vbnet
Dim bmp As Bitmap
bmp = New Bitmap(Me.Width, Me.Height)
```

The `bmp` variable represents an empty bitmap. Set the control’s Image property to this bitmap by using the following statement:

```vbnet
Me.BackgroundImage = bmp
```

Immediately after that, you must set the bitmap to the control’s background color via the Clear method:

```vbnet
G.Clear(Me.BackColor)
```

If you’re using the PictureBox control to draw on, replace the `BackgroundImage` property with the `Image` property. After the execution of this statement, anything we draw on the `bmp` bitmap is shown on the surface of the PictureBox control and is permanent. All we need is a Graphics object that represents the bitmap, so that we can draw on the control. The following statement creates a Graphics object based on the `bmp` variable:

```vbnet
Dim G As Graphics
G = Graphics.FromImage(bmp)
```

Now, we’re in business. We can call the `G` object’s drawing methods to draw and create permanent graphics on the form. You can put all the statements presented so far in a function that returns a Graphics object (Listing 4.3) and use it in your applications.

**LISTING 4.3: Retrieving a Graphics Object from a Form’s Bitmap**

```vbnet
Function GetGraphicsObject(ByVal PBox As PictureBox) As Graphics
    Dim bmp As Bitmap
    bmp = New Bitmap(Me.Width, Me.Height)
    Dim G As Graphics
    Me.BackgroundImage = bmp
    G = Graphics.FromImage(bmp)
    Return G
End Function
```

To create permanent drawings on the surface of the form, you must call the `GetGraphicsObject()` function to obtain a Graphics object from the form’s bitmap. Listing 4.4 is the revised `GetGraphicsObject()` function for the PictureBox control.
LISTING 4.4: Retrieving a Graphics Object from a PictureBox Control’s Bitmap

```vbnet
Function GetGraphicsObject() As Graphics
    Dim bmp As Bitmap
    bmp = New Bitmap(PBox.Width, PBox.Height)
    PBox.Image = bmp
    Dim G As Graphics
    G = Graphics.FromImage(bmp)
    Return G
End Function
```

Now that you know how to draw on the Graphics object and you’re familiar with the basic drawing objects, we can discuss the drawing methods in detail. In the following sections, I use the CreateGraphics method to retrieve the drawing surface of a PictureBox or form to keep the examples short. You can modify any of the projects to draw on the Graphics object derived from a bitmap. All you have to do is replace the statements that create the G variable with a call to the CreateGraphics() function.

Drawing Methods

The Framework provides several drawing methods, one for each basic shape. You can create much more elaborate shapes by combining the methods described in the following sections.

All drawing methods have a few things in common. The first argument is always a Pen object, which will be used to render the shape on the Graphics object. The following arguments are the parameters of a shape: They determine the location and dimensions of the shape. The DrawLine method, for example, needs to know the endpoints of the line to draw, whereas the DrawRectangle method needs to know the origin and dimensions of the rectangle to draw. The parameters needed to render the shape are passed as arguments to each drawing method, following the Pen object.

The drawing methods can also be categorized in two major groups: the methods that draw stroked shapes (outlines) and the methods that draw filled shapes. The methods in the first group start with the Draw prefix (DrawRectangle, DrawEllipse, and so on). The methods of the second group start with the Fill prefix (FillRectangle, FillEllipse, and so on). Of course, some DrawXXX methods don’t have an equivalent FillXXX method. For example, you can’t fill a line or an open curve, so there are no FillLine or FillCurve methods.

Another difference between the drawing and filling methods is that the filling methods use a Brush object to fill the shape — you can’t fill a shape with a pen. So, the first argument of the methods that draw filled shapes is a Brush object, not a Pen object. The remaining arguments are the same because you must still specify the shape to be filled. In the following sections, I present in detail the shape-drawing methods but not the shape-filling methods. If you can use a drawing method, you can just as easily use its filling counterpart.

Table 4.4 shows the names of the drawing methods. The first column contains the methods for drawing stroked shapes, and the second column contains the corresponding methods for drawing filled shapes (if there’s a matching method).
<table>
<thead>
<tr>
<th>DRAWING METHOD</th>
<th>FILLING METHOD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawArc</td>
<td></td>
<td>Draws an arc</td>
</tr>
<tr>
<td>DrawBezier</td>
<td></td>
<td>Draws very smooth curves with fixed endpoints, whose exact shape is determined by two control points</td>
</tr>
<tr>
<td>DrawBeziers</td>
<td></td>
<td>Draws multiple Bezier curves in a single call</td>
</tr>
<tr>
<td>DrawClosedCurve</td>
<td>FillClosedCurve</td>
<td>Draws a closed curve</td>
</tr>
<tr>
<td>DrawCurve</td>
<td></td>
<td>Draws curves that pass through certain points</td>
</tr>
<tr>
<td>DrawEllipse</td>
<td>FillEllipse</td>
<td>Draws an ellipse</td>
</tr>
<tr>
<td>DrawIcon</td>
<td></td>
<td>Renders an icon on the Graphics object</td>
</tr>
<tr>
<td>DrawImage</td>
<td></td>
<td>Renders an image on the Graphics object</td>
</tr>
<tr>
<td>DrawLine</td>
<td></td>
<td>Draws a line segment</td>
</tr>
<tr>
<td>DrawLines</td>
<td></td>
<td>Draws multiple line segments in a single call</td>
</tr>
<tr>
<td>DrawPath</td>
<td>FillPath</td>
<td>Draws a GraphicsPath object</td>
</tr>
<tr>
<td>DrawPie</td>
<td>FillPie</td>
<td>Draws a pie section</td>
</tr>
<tr>
<td>DrawPolygon</td>
<td>FillPolygon</td>
<td>Draws a polygon (a series of line segments between points)</td>
</tr>
<tr>
<td>DrawRectangle</td>
<td>FillRectangle</td>
<td>Draws a rectangle</td>
</tr>
<tr>
<td>DrawRectangles</td>
<td>FillRectangles</td>
<td>Draws multiple rectangles in a single call</td>
</tr>
<tr>
<td>DrawString</td>
<td></td>
<td>Draws a string in the specified font on the drawing surface</td>
</tr>
<tr>
<td></td>
<td>FillRegion</td>
<td>Fills a Region object</td>
</tr>
</tbody>
</table>

Some of the drawing methods allow you to draw multiple shapes of the same type, and they’re properly named DrawLines, DrawRectangles, and DrawBeziers. We simply supply more shapes as arguments, and they’re drawn one after the other with a single call to the corresponding method. The multiple shapes are stored in arrays of the same type as the individual shapes. The DrawRectangle method, for example, accepts as an argument the Rectangle object to be drawn. The DrawRectangles method accepts as an argument an array of Rectangle objects and draws all of them at once.

**DrawLine**

The DrawLine method draws a straight-line segment between two points with a pen supplied as an argument. The simplest forms of the DrawLine method are the following, where point1 and point2 are either Point or PointF objects, depending on the coordinate system in use:
Graphics.DrawLine(pen, X1, Y1, X2, Y2)
Graphics.DrawLine(pen, point1, point2)

**DRAWRECTANGLE**
The DrawRectangle method draws a stroked rectangle and has two forms:

Graphics.DrawRectangle(pen, rectangle)
Graphics.DrawRectangle(pen, X1, Y1, width, height)

The rectangle argument is a Rectangle object that specifies the shape to be drawn. In the second form of the method, the arguments X1 and Y1 are the coordinates of the rectangle’s top-left corner, and the other two arguments are the dimensions of the rectangle. All these arguments can be integers or singles, depending on the coordinate system in use. However, they must be all of the same type.

The following statements draw two rectangles, one inside the other. The outer rectangle is drawn with a red pen with the default width, whereas the inner rectangle is drawn with a 3-pixel-wide green pen and is centered within the outer rectangle:

G.DrawRectangle(Pens.Red, 100, 100, 200, 100)
G.DrawRectangle(New Pen(Color.Green, 3),
               125, 125, 150, 50)

**DRAWELLIPSE**
An ellipse is an oval or circular shape, determined by the rectangle that encloses it. The two dimensions of this rectangle are the ellipse’s major and minor diameters. Instead of giving you a mathematically correct definition of an ellipse, I prepared a few ellipses with different ratios of their two diameters (these ellipses are shown in Figure 4.6). The figure was prepared with the GDIPlus sample application, which demonstrates a few more graphics operations. The ellipse is oblong along the direction of the major diameter and squashed along the direction of the minor diameter. If the two diameters are exactly equal, the ellipse becomes a circle. Indeed, the circle is just a special case of the ellipse, and there’s no DrawCircle method.

![Figure 4.6](image)
Two ellipses with their enclosing rectangles
To draw an ellipse, call the `Graphics.DrawEllipse` method, which has two basic forms:

```vbnet
Graphics.DrawEllipse(pen, rectangle)
Graphics.DrawEllipse(pen, X1, Y1, width, height)
```

The arguments are the same as with the `DrawRectangle` method because an ellipse is basically a circle deformed to fit in a rectangle. The two ellipses and their enclosing rectangles shown in Figure 4.6 were generated with the statements of Listing 4.5.

**LISTING 4.5: Drawing Ellipses and Their Enclosing Rectangles**

```vbnet
Private Sub btnEllipses_Click(...) Handles btnEllipses.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
    Dim R1, R2 As Rectangle
    R1 = New Rectangle(10, 10, 160, 320)
    R2 = New Rectangle(200, 85, 320, 160)
    G.DrawEllipse(New Pen(Color.Black, 3), R1)
    G.DrawRectangle(Pens.Black, R1)
    G.DrawEllipse(New Pen(Color.Black, 3), R2)
    G.DrawRectangle(Pens.Red, R2)
End Sub
```

The ellipses were drawn with a 3-pixel-wide pen. As you can see in the figure, the width of the ellipse is split to the inside and outside of the enclosing rectangle, which is drawn with a 1-pixel-wide pen.

**DRAWPIE**

A pie is a shape similar to a slice of pie (an arc along with the two line segments that connect its endpoints to the center of the circle or the ellipse, to which the arc belongs). The `Graphics.DrawPie` method accepts as arguments the pen with which it will draw the shape, the circle to which the pie belongs, the arc's starting angle, and its sweep angle. The circle (or the ellipse) of the pie is defined with a rectangle. The starting and sweeping angles are measured clockwise. The `Graphics.DrawPie` method has two forms:

```vbnet
Graphics.DrawPie(pen, rectangle, start, sweep)
Graphics.DrawPie(pen, X, Y, width, height, start, sweep)
```

The two forms of the method differ in how the rectangle is defined (a Rectangle object versus its coordinates and dimensions). The `start` argument is the pie's starting angle, and `sweep` is the angle of the pie. The ending angle is `start + sweep`. Angles are measured in degrees (there are 360 degrees in a circle) and increase in a clockwise direction. The 0 angle corresponds to the horizontal axis.

The statements of Listing 4.6 create a pie chart by drawing individual pie slices. Each pie starts where the previous one ends, and the sweeping angles of all pies add up to 360 degrees, which corresponds to a full rotation (a full circle). Unlike the other samples of this section, I've
used the FillPie method, because we hardly ever draw the outlines of the pies; we fill each one with a different color instead. Figure 4.7 shows the output produced by Listing 4.6.

**LISTING 4.6:** Drawing a Simple Pie Chart with the FillPie Methods

```vbnet
Private Sub btnnPie_Click(...) Handles btnnPie.Click
    Dim G As System.Drawing.Graphics
    G = Me.CreateGraphics
    Dim brush As System.Drawing.SolidBrush
    Dim rect As Rectangle
    brush = New System.Drawing.SolidBrush(Color.Green)
    Dim Angles() As Single = {0, 43, 79, 124, 169, 252, 331, 360}
    G.Clear(Color.Ivory)
    rect = New Rectangle(100, 10, 300, 300)
    Dim angle As Integer
    For angle = 1 To Angles.GetUpperBound(0)
        brush.Color = Colors(angle - 1)
        G.FillPie(brush, rect, Angles(angle - 1), Angles(angle) - Angles(angle - 1))
    Next
    G.DrawEllipse(Pens.Black, rect)
End Sub
```

**Figure 4.7**
A simple pie chart generated with the FillPie method

The code sets up two arrays: one with angles and another with colors. The Angles array holds the starting angle of each pie. The sweep angle of each pie is the difference between its own starting angle and the starting angle of the following pie. The sweep angle of the first pie...
is \( \text{Angles}(1) - \text{Angles}(0) \), which is 43 degrees. The loop goes through each pie and draws it with a color it picks from the \textit{Colors} array, based on the angles stored in the \textit{Angles} array. In your application, you must calculate the total of a quantity (such as all customers, or all units of a product sold in a territory) and then use the individual percentages to set the starting and ending angles of each pie. If there are 800 customers and 20 of them belong to a specific area, this area’s sweep angle should be 1/40 of the circle, which is 9 degrees.

Notice that the \texttt{FillPie} method doesn’t connect the pie’s endpoints to the center of the ellipse. The second button on the PieChart project’s form draws the same pie chart, but it also connects each slice’s endpoints to the center of the circle. The code behind this button is identical to the code shown in Listing 4.6 — with the exception that after calling the \texttt{FillPie} method, it calls the \texttt{DrawPie} method to draw the outline of the pie.

\section*{DrawPolygon}

The \texttt{DrawPolygon} method draws an arbitrary polygon. It accepts two arguments: the Pen that it will use to render the polygon and an array of points that define the polygon. The polygon has as many sides (or vertices) as there are points in the array, and it’s always closed, even if the first and last points are not identical. In fact, you do not need to repeat the starting point at the end because the polygon will be automatically closed. The syntax of the \texttt{DrawPolygon} method is the following:

\begin{verbatim}
Graphics.DrawPolygon(pen, points())
\end{verbatim}

where \texttt{points} is an array of points, which can be declared with a statement like the following:

\begin{verbatim}
Dim points() As Point = {New Point(x1, y1), New Point(x2, y2), …}
\end{verbatim}

\section*{DrawCurve}

Curves are smooth lines drawn as \textit{cardinal splines}. A real spline is a flexible object (made of soft wood) that designers used to flex on the drawing surface with spikes. The spline goes through all the fixed points and assumes the smoothest possible shape, given the restrictions imposed by the spikes. If the spline isn’t flexible enough, it breaks. In modern computer graphics, there are mathematical formulas that describe the path of the spline through the fixed points and take into consideration the tension (the degree of flexibility) of the spline. A more flexible spline yields a curve that bends easily. Less-flexible splines do not bend easily around their fixed points. Computer-generated splines do not break, but they can take unexpected shapes.

To draw a curve with the \texttt{DrawCurve} method, you specify the locations of the spikes (the points that the spline must go through) and the spline’s tension. If the tension is 0, the spline is totally flexible, like a rubber band: All the segments between points are straight lines. The higher the tension, the smoother the curve will be. Figure 4.8 shows four curves passing through the same points, but each curve is drawn with a different tension value. The curves shown in the figure were drawn with the GDIPPlus project (using the Ordinal Curves button).

The simplest form of the \texttt{DrawCurve} method has the following syntax, where \texttt{points} is an array of points:

\begin{verbatim}
Graphics.DrawCurve(pen, points, tension)
\end{verbatim}

The first and last elements of the array are the curve’s endpoints, and the curve will go through the remaining points as well.
These curves go through the same points, but they have different tensions.

The curves shown in Figure 4.8 were produced by the code shown in Listing 4.7. Notice that a tension of 0.5 is practically the same as 0 (the spline bends around the fixed points like a rubber band). If you drew the same curve with a tension of 5, you’d get an odd curve indeed because although a physical spline would break, the mathematical spline takes an unusual shape to accommodate the fixed points.

**Listing 4.7: Curves with Common Fixed Points and Different Tensions**

```vbnet
Private Sub btnCurves_Click(...) Handles btnCurves.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.Clear(PictureBox1.BackColor)
    G.FillRectangle(Brushes.Silver, ClientRectangle)
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.HighQuality
    Dim points() As Point = { 
        New Point(20, 50), New Point(220, 190), 
        New Point(330, 80), New Point(450, 280)}
    G.DrawCurve(Pens.Blue, points, 0.1)
    G.DrawCurve(Pens.Red, points, 0.5)
    G.DrawCurve(Pens.Green, points, 1)
    G.DrawCurve(Pens.Black, points, 2)
End Sub
```

**DrawBezier**

The `DrawBezier` method draws Bezier curves, which are smoother than cardinal splines. A Bezier curve is defined by two endpoints and two control points. The control points act as magnets. The curve is the trace of a point that starts at one of the endpoints and moves toward the second one. As it moves, the point is attracted by the two control points. Initially, the first control point’s influence is predominant. Gradually, the curve comes into the second control point’s field and it ends at the second endpoint.
The `DrawBezier` method accepts a pen and four points as arguments:

```vbnet
g.Graphics.DrawBezier(pen, X1, Y1, X2, Y2, X3, Y3, X4, Y4)
g.Graphics.DrawBezier(pen, point1, point2, point3, point4)
```

Figure 4.9 shows four Bezier curves, which differ in the y-coordinate of the third control point. All control points are marked with little squares: one each for the three points that are common to all curves, and four in a vertical column for the point that differs in each curve.

**Figure 4.9** Bezier curves and their control points

The code of Listing 4.8 draws the four Bezier curves (I'm not showing the statements that draw the small rectangles; they simply call the `FillRectangle` method). The endpoints and one control point (P1, P2, and P4) remain the same, whereas the other control point (P3) is set to four different values. Notice how far the control point must go to have a significant effect on the curve's shape.

**Listing 4.8:** Drawing Bezier Curves and Their Control Points

```vbnet
Private Sub btnBezier_Click(...) Handles btnBezier.Click
    Dim G As Graphics
    G = Me.PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
    G.FillRectangle(Brushes.Silver, ClientRectangle)
    Dim P1 As New Point(120, 150)
    Dim P2 As New Point(220, 90)
    Dim P3 As New Point(330, 30)
    Dim P4 As New Point(410, 110)
    Dim sqrSize As New Size(6, 6)
    G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
    P3 = New Point(330, 130)
    G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
```
To draw the curve, all you need is to specify the four control points and pass them along with a Pen object to the `DrawBezier` method.

**DRAWPATH**

This method accepts a Pen object and a Path object as arguments and renders the specified path on the screen:

```vbnet
Graphics.DrawPath(pen, path)
```

To construct the Path object, use the `AddXXX` methods (AddLine, AddRectangle, and so on).

**DRAWSTRING, MEASURESTRING**

The DrawString method renders a string in a single line or multiple lines. As a reminder, the `TextRenderingHint` property of the Graphics object allows you to specify the quality of the rendered text. The simplest form of the DrawString method is the following:

```vbnet
Graphics.DrawString(string, font, brush, X, Y)
```

The first argument is the string to be rendered in the font specified by the second argument. The text will be rendered with the Brush object specified by the `brush` argument. `X` and `Y`, finally, are the coordinates of the top-left corner of a rectangle that completely encloses the string.

While working with strings, in most cases you need to know the actual dimensions of the string when rendered with the DrawString method in the specified font. The MeasureString method allows you to retrieve the metrics of a string before actually drawing it. This method returns a SizeF structure with the width and height of the string when rendered on the same Graphics object with the specified font. We'll use this method extensively in the tutorial “Printing with Visual Basic 2010,” to position text precisely on the printed page. You can also pass a Rectangle object as an argument to the MeasureString method to find out how many lines it will take to render the string on the rectangle.

The simplest form of the MeasureString method is the following, where `string` is the string to be rendered and `font` is the font in which the string will be rendered:

```vbnet
Dim textSize As SizeF
textSize = Me.Graphics.MeasureString(string, font)
```

To center a string on the form, use the x-coordinate returned by the MeasureString method, as in the following code segment:

```vbnet
Dim textSize As SizeF
Dim X As Integer, Y As Integer = 0
textSize = Me.Graphics.MeasureString(string, font)
X = (Me.Width - textSize.Width) / 2
G.DrawString("Centered string", font, brush, X, Y)
```
We subtract the rendered string’s length from the form’s width, and we split the difference in half at the two sides of the string.

Figure 4.10 shows a string printed at the center of the form and the two lines passing through the same point. Listing 4.9 shows the statements that produced the string. This listing is part of the TextEffects sample project.

**LISTING 4.9: Printing a String Centered on the Form**

```vbnet
Private Sub Center(...) Handles bttnCentered.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
    G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
    FontDialog1.Font = Me.Font
    FontDialog1.ShowDialog()
    Dim txtFont As Font
    txtFont = FontDialog1.Font
    G.DrawLine(New Pen(Color.Green), CInt(Me.Width / 2), CInt(0), CInt(Me.Width / 2), CInt(Me.Height))
    G.DrawLine(New Pen(Color.Green), 0, CInt(Me.Height / 2), CInt(Me.Width), CInt(Me.Height / 2))
    Dim txtLen, txtHeight As Integer
    Dim str As String = "Visual Basic 2008"
    Dim txtSize As SizeF
    txtSize = G.MeasureString(str, txtFont)
    Dim txtX, txtY As Integer
    txtX = (Me.Width - txtSize.Width) / 2
    txtY = (Me.Height - txtSize.Height) / 2
    G.DrawString(str, txtFont, New SolidBrush(Color.Red), txtX, txtY)
    Me.Invalidate()
End Sub
```

**FIGURE 4.10**
Centering a string on a form
The coordinates passed to the `DrawString` method (variables `txtX` and `txtY`) are the coordinates of the top-left corner of the rectangle that encloses the first character of the string.

Another form of the `DrawString` method accepts a rectangle as an argument and draws the string in this rectangle, breaking the text into multiple lines if needed. The syntax of this form of the method is as follows:

```csharp
Graphics.DrawString(string, font, brush, rectanglef)
Graphics.DrawString(string, font, brush, rectanglef, stringFormat)
```

If you want to render text in a box, you will most likely use the equivalent form of the `MeasureString` method to retrieve the metrics of the text in the rectangle. This form of the `MeasureString` method returns the number of lines it will take to render the string in the supplied rectangle, and it has the following syntax, where `string` is the text to be rendered, and `font` is the font in which the string will be rendered:

```csharp
e.Graphics.MeasureString(string, font, fitSize,
                          stringFormat, lines, cols)
```

The `fitSize` argument is a `SizeF` object that represents the width and height of a rectangle, where the string must fit. The `lines` and `cols` variables are passed by reference, and they are set by the `MeasureString` method to the number of lines and number of characters that will fit in the specified rectangle. The exact location of the rectangle doesn’t make any difference — only its dimensions matter, and that’s why the third argument is a `SizeF` object, not a `Rectangle` object.

Figure 4.11 shows a string printed in two different rectangles by the TextEffects sample project; the figure was created with the Draw Boxed Text button. The code that produced the figure is shown in Listing 4.10.

**LISTING 4.10:** Printing Text in a Rectangle

```csharp
Private Sub BoxedText(...) Handles bttnBoxed.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
    FontDialog1.Font = Me.Font
    FontDialog1.ShowDialog()
    Dim txtFont As Font
    txtFont = FontDialog1.Font
    Dim txt As String = "This text was rendered in a rectangle " & _
                       "with the DrawString method of the Form's " & _
                       "Graphics object. "
    txt = txt & txt & txt & txt & txt
    G.DrawString(txt, txtFont, Brushes.Black, _
                 New RectangleF(100, 80, 180, 250))
    G.DrawRectangle(Pens.Red, 100, 80, 180, 250)
    G.DrawString(txt, txtFont, Brushes.Black, _
                 New RectangleF(350, 100, 400, 150))
    G.DrawRectangle(Pens.Red, 350, 100, 400, 150)
    Me.Invalidate()
End Sub
```
The StringFormat Object

Some of the overloaded forms of the DrawString method accept an argument of the StringFormat type. This argument determines characteristics of the text and exposes a few properties of its own, which include the following:

Alignment
Determines the alignment of the text; its value is a member of the StringAlignment enumeration: Center (text is aligned in the center of the layout rectangle), Far (text is aligned far from the origin of the layout rectangle), and Near (text is aligned near the origin of the layout rectangle).

Trimming
Determines how text will be trimmed if it doesn’t fit in the layout rectangle. Its value is one of the members of the StringTrimming enumeration: Character (text is trimmed to the nearest character), EllipsisCharacter (text is trimmed to the nearest character and an ellipsis is inserted at the end to indicate that some of the text is missing), EllipsisPath (text at the middle of the string is removed and replaced by an ellipsis), EllipsisWord (text is trimmed to the nearest word and an ellipsis is inserted at the end), None (no trimming), and Word (text is trimmed to the nearest word).

FormatFlags
Specifies layout information for the string. Its value can be one of the members of the StringFormatFlags enumeration. The two members of this enumeration that you might need often are DirectionRightToLeft (prints to the left of the specified point) and DirectionVertical.

To use the stringFormat argument of the DrawString method, instantiate a variable of this type, set the desired properties, and then pass it as an argument to the DrawString method, as shown here:

```vbnet
Dim G As Graphics = Me.CreateGraphics
Dim SF As New StringFormat()
SF.FormatFlags = StringFormatFlags.DirectionVertical
G.DrawString("Visual Basic", Me.Font, Brushes.Red, 80, 80, SF)
```
The call to the DrawString method will print the string from top to bottom. It will also rotate the characters. The DirectionRightToLeft setting will cause the DrawString method to print the string to the left of the specified point, but it will not mirror the characters.

You can find additional examples of the MeasureString method in the tutorial “Printing with VB2010,” in which we’ll use this method to fit strings on the width of the page. The third button on the form of the TextEffects project draws text with a three-dimensional look by overlaying a semitransparent string over an opaque string. You might also wonder why none of the DrawString methods’ forms accept as an argument an angle of rotation for the text. You can draw text or any shape at any orientation as long as you set up the proper rotation transformation. This topic is discussed in the section “Applying Transformations” of the tutorial.

**DRAWIMAGE**

The DrawImage method, which renders an image on the Graphics object, is a heavily overloaded and quite flexible method. The following form of the method draws the image at the specified location. Both the image and the location of its top-left corner are passed to the method as arguments (as Image and Point arguments, respectively):

```csharp
Graphics.DrawImage(img, point)
```

Another form of the method draws the specified image within a rectangle. If the rectangle doesn’t match the original dimensions of the image, the image will be stretched to fit in the rectangle. The rectangle should have the same aspect ratio as the Image object, to avoid distorting the image in the process.

```csharp
Graphics.DrawImage(img, rectangle)
```

Another form of the method allows you to change not only the magnification of the image, but also its shape. This method accepts as an argument not a rectangle, but an array of three points that specifies a parallelogram. The image will be sheared to fit in the parallelogram, where `points` is an array of points that define a parallelogram:

```csharp
Graphics.DrawImage(img, points())
```

The array holds three points, which are the top-left, top-right, and bottom-left corners of the parallelogram. The fourth point is determined uniquely by the other three, and you need not supply it. The ImageCube sample project, shown later in this tutorial, uses this overloaded form of the DrawImage method to draw a cube with a different image on each face.

Another interesting form of the method allows you to set the attributes of the image:

```csharp
Graphics.DrawImage(image, points(), srcRect, units, attributes)
```

The first two arguments are the same as in the previous forms of the method. The `srcRect` argument is a rectangle that specifies the portion of image to draw, and `units` is a constant of the GraphicsUnit enumeration. It determines how the units of the rectangle are measured (pixels, inches, and so on). The last argument is an ImageAttributes object that contains information about the attributes of the image you want to change (such as the gamma value, and a transparent color value or color key). The properties of the ImageAttributes class are discussed shortly.

The DrawImage method is quite flexible, and you can use it for many special effects, including wipes. A *wipe* is the gradual appearance of an image on a form or PictureBox control. You
can use this method to draw stripes of the original image, or start with a small rectangle in the
center that grows gradually until it covers the entire image.
You can also correct the color of the image by specifying the attributes argument. To
specify the attributes argument, create an ImageAttributes object with a statement like the
following:

```vbnet
dim attr as new system.drawing.imaging.imageattributes
```

Then call one or more of the ImageAttributes class's methods:

**SetWrapMode**
Specifies the wrap mode that is used to decide how to tile a texture across a shape. This
attribute is used with textured brushes (a topic that isn’t discussed in this book).

**SetGamma**
This method sets the gamma value for the image’s colors and accepts a Single value, which
is the gamma value to be applied. A gamma value of 1 doesn’t affect the colors of the image.
A smaller value darkens the colors, whereas a larger value makes the image colors brighter.
Notice that the gamma correction isn’t the same as manipulating the brightness of the colors.
The gamma correction takes into consideration the entire range of values in the image; it
doesn’t apply equally to all the colors. In effect, it takes into consideration both the brightness
and the contrast and corrects them in tandem with a fairly complicated algorithm. The syntax
of the SetGamma method is as follows:

```vbnet
imageattributes.setgamma(gamma)
```

The following statements render the image stored in the `img` Image object on the `G` Graphics
object, and they gamma-correct the image in the process by a factor of 1.25:

```vbscript
dim attrs as new system.drawing.imaging.imageattributes()
attrs.setgamma(1.25)
dim dest as new rectangle(0, 0, picturebox1.width, picturebox1.height)
g.drawimage(img, dest, 0, 0, img.width, img.height, _
graphicsunit.pixel, attrs)
```

**Gradients**
In this section, you’ll look at the tools for creating gradients. The techniques for gradients can
get quite complicated, but I will limit the discussion to the types of gradients you’ll need for
business or simple graphics applications.

**Linear Gradients**
Let’s start with linear gradients. Like all other gradients, they’re part of the System.Drawing
class and are implemented as brushes. To draw a linear gradient, you must create an instance
of the LinearGradientBrush class with a statement like the following:

```vbscript
dim lgbrush as lineargradientbrush
lgbrush = new lineargradientbrush(rect, startcolor, endcolor, gradientmode)
```
To understand how to use the arguments, you must understand how the linear gradient works. This method creates a gradient that fills a rectangle, specified by the `rect` object passed as the first argument. This rectangle isn’t filled with any gradient; it simply tells the method how long (or how tall) the gradient should be. The gradient starts with the `startColor` at the left side of the rectangle and ends with the `endColor` at the opposite side. The gradient changes color slowly as it moves from one end to the other. The last argument, `gradientMode`, specifies the direction of the gradient and can have one of the values shown in Table 4.5.

**Table 4.5: The LinearGradientMode Enumeration**

<table>
<thead>
<tr>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackwardDiagonal</td>
<td>The gradient fills the rectangle diagonally from the top-right corner (<code>startColor</code>) to the bottom-left corner (<code>endColor</code>).</td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>The gradient fills the rectangle diagonally from the top-left corner (<code>startColor</code>) to the bottom-right corner (<code>endColor</code>).</td>
</tr>
<tr>
<td>Horizontal</td>
<td>The gradient fills the rectangle from left (<code>startColor</code>) to right (<code>endColor</code>).</td>
</tr>
<tr>
<td>Vertical</td>
<td>The gradient fills the rectangle from top (<code>startColor</code>) to bottom (<code>endColor</code>).</td>
</tr>
</tbody>
</table>

Notice that in the descriptions of the various modes in the table, I state that the gradient fills the rectangle, not the shape. The gradient is calculated according to the dimensions of the rectangle specified with the first argument. If the actual shape is smaller than this rectangle, only a section of the gradient will be used to fill the shape. If the shape is larger than this rectangle, the gradient will repeat as many times as necessary to fill the shape. We usually fill a shape that’s as wide (or as tall) as the rectangle used to specify the gradient.

Let’s say you want to use the same gradient that extends 300 pixels horizontally to fill two rectangles: one that’s 200 pixels wide and another that’s 600 pixels wide. The first rectangle, which is 200 pixels wide, will be filled with two thirds of the gradient; the second rectangle, which is 600 pixels wide, will be filled with a gradient that’s repeated twice. The code in Listing 4.11 corresponds to the Linear Gradient button of the Gradients project.

**Listing 4.11: Filling Rectangles with a Linear Gradient**

```vbnet
Private Sub LinearGradient_Click(...) Handles btnLinearGradient.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    Dim R As New RectangleF(20, 20, 300, 100)
    Dim startColor As Color = Color.BlueViolet
    Dim EndColor As Color = Color.LightYellow
    Dim LGBrush As New System.Drawing.Drawing2D.LinearGradientBrush(R, startColor, EndColor, LinearGradientMode.Horizontal)
    G.FillRectangle(LGBrush, New Rectangle(20, 20, 200, 100))
    G.FillRectangle(LGBrush, New Rectangle(20, 150, 600, 100))
End Sub
```
For a horizontal gradient, only the width of the rectangle is used; the height is irrelevant. For a vertical gradient, only the height of the rectangle matters. When you draw a diagonal gradient, both dimensions are taken into consideration.

You can create gradients at various directions by setting the gradientMode argument of the LinearGradientBrush object’s constructor. The Diagonal Linear Gradient button on the Gradients project does exactly that.

The button Gradient Text on the form of the Gradients project renders some text filled with a linear gradient. As you recall from our discussion of the DrawString method, strings are rendered with a Brush object, not a Pen object. If you specify a LinearGradientBrush object, the text will be rendered with a linear gradient. The text shown in Figure 4.12 was produced by the Gradient Text button, whose code is shown in Listing 4.12.

Listing 4.12: Rendering Strings with a Linear Gradient

Private Sub btnGradientText_Click(...) Handles btnGradientText.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    G.Clear(Me.BackColor)
    G.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias
    Dim largeFont As New Font("
        'Comic Sans MS', 48, FontStyle.Bold, GraphicsUnit.Point)
    Dim gradientStart As New PointF(0, 0)
    Dim txt As String = "Gradient Text"
    Dim txtSize As New SizeF()
    txtSize = G.MeasureString(txt, largeFont)
    Dim gradientEnd As New PointF()
    gradientEnd.X = txtSize.Width
    gradientEnd.Y = txtSize.Height
    Dim grBrush As New LinearGradientBrush(gradientStart, gradientEnd, _
        Color.Yellow, Color.Blue)
    G.DrawString(txt, largeFont, grBrush, 20, 20)
End Sub

Figure 4.12
Drawing a string filled with a gradient

The code of Listing 4.12 is a little longer than it could be (or than you might expect). Because linear gradients have a fixed size and don’t expand or shrink to fill the shape, you must call the
MeasureString method to calculate the width of the string and then create a linear gradient with the exact same width. This way, the gradient's extent matches that of the string.

**Path Gradients**

This is the ultimate gradient tool. Using a PathGradientBrush, you can create a gradient that starts at a single point and fades into multiple different colors in different directions. You can fill a rectangle starting from a point in the interior of the rectangle, which is colored, say, black. Each corner of the rectangle might have a different ending color. The PathGradientBrush will change color in the interior of the shape and will generate a gradient that's smooth in all directions. Figure 4.13 shows a rectangle filled with a path gradient, although the gray shades on the printed page won't show the full impact of the gradient. Open the Gradients project to see the same figure in color (use the Path Gradient button).

![Figure 4.13](image)

**A path gradient starting at the middle of the rectangle**

To fill a shape with a path gradient, you must first create a Path object. The PathGradientBrush will be created for the specific path and can be used to fill this path — but not any other shape. Actually, you can fill any other shape with the PathGradientBrush created for a specific path, but the gradient won't fit the new shape. To create a PathGradientBrush, use the following syntax, where `path` is a properly initialized Path object:

```csharp
Dim pgBrush As New PathGradientBrush(path)
```

The `pgBrush` object provides properties that determine the exact coloring of the gradient. First, you must specify the color of the gradient at the center of the shape by using the `CenterColor` property. The `SurroundColors` property is an array with as many elements as there are vertices (corners) in the Path object. Each element of the `SurroundColors` array must be set to a color value, and the resulting gradient will have the color of the equivalent element of the `SurroundColors` array.

The following declaration creates an array of three different colors and assigns the colors to the `SurroundColors` property of a PathGradientBrush object:

```csharp
Dim Colors() As Color = {Color.Yellow, Color.Green, Color.Blue}
pgBrush.SurroundColors = Colors
```

After setting the PathGradientBrush, you can fill the corresponding Path object by calling the `FillPath` method. The Path Gradient button on the Gradient application's main form creates
a rectangle filled with a gradient that’s red in the middle of the rectangle and has a different color at each corner. Listing 4.13 shows the code behind the Path Gradient button.

**Listing 4.13:** Filling a Rectangle with a Path Gradient

```vbnet
Private Sub btnPathGradient_Click(...) Handles btnPathGradient.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    Dim path As New System.Drawing.Drawing2D.GraphicsPath()
    path.AddLine(New Point(10, 10), New Point(400, 10))
    path.AddLine(New Point(400, 10), New Point(400, 250))
    path.AddLine(New Point(400, 250), New Point(10, 250))
    Dim pathBrush As New System.Drawing.Drawing2D.PathGradientBrush(path)
    pathBrush.CenterColor = Color.Red
    Dim surroundColors() As Color = {Color.Yellow, Color.Green, Color.Blue, Color.Cyan}
    pathBrush.SurroundColors = surroundColors
    G.FillPath(pathBrush, path)
End Sub
```

The gradient’s center point is, by default, the center of the shape. You can also specify the center of the gradient (the point that will be colored according to the `CenterColor` property). You can place the center point of the gradient anywhere by setting its `CenterPoint` property to a `Point` or `PointF` value.

The Gradients application has a few more buttons that create interesting gradients, which you can examine on your own. The Rectangle Gradient button fills a rectangle with a gradient that has a single ending color all around. All the elements of the `SurroundColors` property are set to the same color. The Animated Gradient animates the same gradient by changing the coordinates of the `PathGradientBrush` object’s `CenterPoint` property slowly over time.

**Clipping**

Anyone who has used drawing or image-processing applications already knows that many of the application’s tools use masks. A mask is any shape that limits the area in which you can draw. If you want to place a star or heart on an image and print something in it, you create the shape in which you want to limit your drawing tools and then you convert this shape into a mask. When you draw with the mask, you can start and end your strokes anywhere on the image. Your actions will have no effect outside of the mask, however.

The mask of the various image-processing applications is a clipping region, which can be anything, as long as it’s a closed shape. While the clipping region is activated, drawing takes place in the area of the clipping region. To specify a clipping region, you must call the `SetClip` method of the `Graphics` object. The `SetClip` method accepts the clipping area as an argument, and the clipping area can be the `Graphics` object itself (no clipping), a `Rectangle`, a `Path`, or a `Region`. A region is a structure made up of simple shapes, just like a path. There are many methods for creating a `Region` object — you can combine and intersect shapes, or exclude shapes from a region — but we aren’t going to discuss the `Region` object in this tutorial.
because it's not among the common objects we use to generate the type of graphics discussed in the context of this book.

The SetClip method has the following forms:

```
Graphics.SetClip(Graphics)
Graphics.SetClip(Rectangle)
Graphics.SetClip(GraphicsPath)
Graphics.SetClip(Region)
```

All methods accept a second optional argument, which determines how the new clipping area will be combined with the existing one. The `combineMode` argument's value is one of the members of the `CombineMode` enumeration: `Complement`, `Exclude`, `Intersect`, `Replace`, `Union`, and `XOR`.

After a clipping area has been set for the Graphics object, drawing is limited to that area. You can specify any coordinates, but only the part of the drawing that falls inside the clipping area is visible. The Clipping project demonstrates how to clip text and images within an elliptical area (see Figure 4.14). The Boxed Text button draws a string in a rectangle. The Clipped Text button draws the same text but first applies a clipping area, which is an ellipse. The Clipped Image button uses the same ellipse to clip an image. Because there's no form of the SetClip method that accepts an ellipse as an argument, we must construct a Path object, add the ellipse to the path, and then create a clipping area based on the path.

**Figure 4.14**
Clipping text (left) and images (right) in an ellipse

---

The following statements create the clipping area for the text, which is an ellipse. The path is created by calling the `AddEllipse` method of the `GraphicsPath` object. This path is then passed as an argument to the `Graphics` object's `SetClip` method:

```
Dim P As New System.Drawing.Drawing2D.GraphicsPath()
Dim clipRect As New RectangleF(30, 30, 250, 150)
P.AddEllipse(clipRect)
Dim G As Graphics
G = PictureBox1.CreateGraphics
G.SetClip(P)
```

Listing 4.14 shows the code behind the Boxed Text and Clipped Text buttons. The Boxed Text button prints some text in a rectangular area that is centered over the clipping area. The Clipped Text button shows how the text is printed within the rectangle. Both the rectangle and the ellipse are based on the same Rectangle object.
LISTING 4.14: The Boxed Text and Clipped Text Buttons

Private Sub bttnBoxedText_Click(...) Handles bttnBoxedText.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    Dim Rect As New Rectangle( _
        Convert.ToInt32((PictureBox1.Width - 250) / 2), _
        Convert.ToInt32((PictureBox1.Height - 150) / 2), 250, 150)
        New Font("Verdana", 12, FontStyle.Regular), _
        Brushes.DarkGreen, Rect, format) G.DrawRectangle(Pens.Yellow, Rect) PictureBox1.Invalidate()
End Sub

Private Sub bttnClippedText_Click(...) Handles bttnClippedText.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    Dim P As New System.Drawing.Drawing2D.GraphicsPath()
    Dim clipRect As New RectangleF( _
        Convert.ToSingle((PictureBox1.Width - 250) / 2), _
        Convert.ToSingle((PictureBox1.Height - 150) / 2), 250, 150)
    P.AddEllipse(clipRect)
    G.ResetTransform() G.DrawEllipse(Pens.Red, clipRect) G.SetClip(P)
        New Font("Verdana", 12, FontStyle.Regular), _
        Brushes.DarkBlue, clipRect, format)
    PictureBox1.Invalidate()
End Sub

The difference between the two subroutines is that the second sets an ellipse as the clipping area; anything we draw on it is automatically clipped.

The Clipped Image button sets up a similar clipping area and then draws an image centered behind the clipping ellipse. As you can see in Figure 4.14, only the segment of the image that’s inside the clipping area is visible. The code behind the Clipped Image button is shown in Listing 4.15.
LISTING 4.15:  The Clipped Image Button

Private Sub btnClippedImage_Click(...) Handles btnClippedImage.Click
    Dim G As Graphics
    G = CreateGraphicsObject
    G.ResetClip()
    Dim P As New System.Drawing.Drawing2D.GraphicsPath()
    Dim clipRect As New RectangleF(10, 10, PictureBox1.Width - 20, PictureBox1.Height - 20)
    P.AddEllipse(clipRect)
    G.SetClip(P)
    G.DrawImage(Image.FromFile(fileName), -150, -150)
    PictureBox1.Invalidate()
End Sub

An easy and interesting technique for creating paths is to use the AddString method of the GraphicsPath object. Then you can draw an image over this path. The net effect is seeing sections of the image through the string’s characters. You can open the StringPath sample project to see how you can clip an image with text; just be sure you select an interesting image to show through the string’s characters.

Applying Transformations

In computer graphics, there are three types of transformations: scaling, translation, and rotation:

- The scaling transformation changes the dimensions of a shape but not its basic form. If you scale an ellipse by 0.5, you’ll get another ellipse that’s half as wide and half as tall as the original one.

- The translation transformation moves a shape by a specified distance. If you translate a rectangle by 30 pixels along the x-axis and 90 pixels along the y-axis, the new origin will be 30 pixels to the right and 90 pixels down from the original rectangle’s top-left corner.

- The rotation transformation rotates a shape by a specified angle, expressed in degrees; 360 degrees correspond to a full rotation, and the shape appears the same. A rotation by 180 degrees is equivalent to flipping the shape vertically and horizontally.

Transformations are stored in a 5 × 5 matrix, but you need not set it up yourself. The Graphics object provides the ScaleTransform, TranslateTransform, and RotateTransform methods, and you can specify the transformation to be applied to the shape by calling one or more of these methods and passing the appropriate argument(s). The ScaleTransform method accepts as arguments scaling factors for the horizontal and vertical directions:

Graphics.ScaleTransformation(Sx, Sy)

If an argument is smaller than one, the shape will be reduced in the corresponding direction; if it’s larger than one, the shape will be enlarged in the corresponding direction. We usually scale both directions by the same factor to retain the shape’s aspect ratio. If you scale a circle
by different factors in the two dimensions, the result will be an ellipse, and not a smaller or larger circle.

The TranslateTransform method accepts two arguments, which are the displacements along the horizontal and vertical directions:

\[
\text{Graphics.TranslateTransform}(T_x, T_y)
\]

The \( T_x \) and \( T_y \) arguments are expressed in the coordinates of the current coordinate system. The shape is moved to the right by \( T_x \) units and down by \( T_y \) units. If one of the arguments is negative, the shape is moved in the opposite direction (to the left or up).

The RotateTransform method accepts a single argument, which is the angle of rotation expressed in degrees:

\[
\text{Graphics.RotateTransform}(\text{rotation})
\]

The rotation takes place about the origin. As you will see, the final position and orientation of a shape is different if two identical rotation and translation transformations are applied in a different order.

Every time you call one of these methods, the elements of the transformation matrix are set accordingly. All transformations are stored in this matrix, and they have a cumulative effect. If you specify two translation transformations, for example, the shape will be translated by the sum of the corresponding arguments in either direction. These two transformations:

\[
\text{Graphics.TranslateTransform}(10, 40) \\
\text{Graphics.TranslateTransform}(20, 20)
\]

are equivalent to the following one:

\[
\text{Graphics.TranslateTransform}(30, 60)
\]

To start a new transformation after drawing some shapes on the Graphics object, call the ResetTransform method, which clears the transformation matrix.

The effect of multiple transformations might be cumulative, but the order in which transformations are performed makes a big difference. You will find some real-world examples of transformations in the tutorial “Printing with Visual Basic 2010,” where I discuss printing with Visual Basic. In specific, you’ll see how to apply transformations to print rotated strings on a page. I’ve also included the Transformations sample project in this tutorial. This project allows you to apply transformations to an entity that consists of a rectangle that contains a string and a small bitmap, as shown in Figure 4.15. Each button on the right performs a different transformation or combination of transformations. The code is quite short, and you can easily insert additional transformations or change their order, and see how the shape is transformed. Keep in mind that some transformations might bring the shape entirely outside the form. In this case, just apply a translation transformation in the opposite direction.

The code behind the Translate Shape, Rotate Shape, and Scale Shape buttons is shown in Listing 4.16. The code in the Click event handlers of the buttons sets the appropriate transformations and then calls the DrawShape() subroutine, passing the current Graphics object as an argument. The DrawShape() subroutine draws the same shape, but its actual output (the position and size of the shape) is affected by the transformation matrix in effect.
LISTING 4.16: The Buttons of the GDIPlusTransformations Project

Private Sub btnTranslate_Click(...) Handles btnTranslate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.TranslateTransform(200, 90)
    DrawShape(G)
End Sub

Private Sub btnRotate_Click(...) Handles btnRotate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.RotateTransform(45)
    DrawShape(G)
End Sub

Private Sub btnTranslateRotate_Click(...) Handles btnTranslateRotate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.TranslateTransform(200, 90)
    G.RotateTransform(45)
    DrawShape(G)
End Sub

VB 2010 at Work: The ImageCube Project

As discussed earlier in this tutorial, the DrawImage method can render images on any parallelogram, not just a rectangle, with the necessary distortion. A way to look at these images is not as distorted images, but as perspective images. Looking at a printout from an unusual angle is
equivalent to rendering an image within a parallelogram. Imagine a cube with a different image glued on each side. To display such a cube on your monitor, you must calculate the coordinates of the cube’s edges and then use these coordinates to define the parallelograms on which each image will be displayed. Figure 4.16 shows a cube with a different image on each side.

**Figure 4.16**
This cube was created with a call to the `DrawImage` method for each visible face of the cube.

If you’re good at math, you can rotate a cube around its vertical and horizontal axes and then map the rotated cube on the drawing surface. You can even apply a perspective transformation, which will make the image look more like the rendering of a three-dimensional cube. This process is more involved than the topics discussed in this book. Instead of doing all the calculations, I came up with a set of coordinates for the parallelogram that represents each vertex (corner) of the cube. For a different orientation, you can draw a perspective view of a cube on paper and measure the coordinates of its vertices. After you define the parallelogram that corresponds to each visible side, you can draw an image on each face by using the `DrawImage` method. The `DrawImage` method will shear the image as necessary to fill the specified area. The result is a 3D-looking cube covered with images. You can open the sample project and examine its code, which contains comments to help you understand how it works.

**Manipulating Bitmaps**
The graphics we have explored so far are called *vector graphics* because they’re based on geometric descriptions and they can be scaled to any extent. Because they’re based on mathematical equations, you can draw any details of the picture without losing any accuracy. You can zoom into a tiny section of an ellipse, for example, and never lose any detail because the ellipse is redrawn every time.

Vector graphics, however, can’t be used to describe the type of images you capture with your digital camera. These images belong to a different category of graphics: *bitmap graphics* or *raster graphics*. A bitmap is a collection of colored pixels arranged in rows and columns. As you will see, a *bitmap* is nothing more than a two-dimensional array of integers that represent
colors, and you can achieve interesting effects with simple arithmetic operations on the pixels of an image.

**Specifying Colors**

You’re already familiar with the Color common dialog box, which lets you specify colors by manipulating their basic components. To specify a Color value through this dialog box, you’ll see three boxes — Red, Green, and Blue (RGB) — whose values change as you move the cross-shaped pointer over the color spectrum. These are the values of the three basic components that computers use to specify colors. Any color that can be represented on a computer monitor is specified by means of these three colors. By mixing percentages of these basic colors, you can design almost any color in the spectrum.

The model of designing colors based on the intensities of their RGB components is called the RGB model, and it’s a fundamental concept in computer graphics. If you aren’t familiar with this model, this section is well worth reading. Nearly every color you can imagine can be constructed by mixing the appropriate percentages of the three basic colors. Each color, therefore, is represented by a triplet of byte values that represent the basic color components of red, green, and blue. The smallest value, 0, indicates the absence of the corresponding color. The largest value, 255, indicates full intensity, or saturation. The triplet (0, 0, 0) is black because all colors are missing, and the triplet (255, 255, 255) is white — it contains all three basic colors in full intensity. Other colors have various combinations: (255, 0, 0) is a pure red tone, (0, 255, 255) is a pure cyan tone (what you get when you mix green and blue), and (0, 128, 128) is a mid-cyan tone (a mix of mid-green and mid-blue tones). The possible combinations of the three basic color components are $256 \times 256 \times 256$, or 16,777,216 colors. Graphics cards that can display all 16 million colors are said to have a color depth of 24 bits (3 bytes). Most graphics cards today support a color depth of 32 bits: 24 bits for color and 8 bits for a transparency layer. (The topic of transparency is discussed shortly.)

Notice that we use the term **basic colors**, not **primary colors**, which are the three colors used in designing colors with paint. The concept is the same: You mix these colors until you get the desired result. The primary colors used in painting, however, are different. They are the colors red, yellow, and blue. Painters can create any shade imaginable by mixing the appropriate percentages of red, yellow, and blue paint. On a computer monitor, you can design any color by mixing the appropriate percentages of red, green, and blue.

There are other color specification models besides the RGB model. Modern color printers use four primary colors: cyan, magenta, yellow, and black (the CMYK model). The color specification model used by computers is called **additive** (you must add all three basic colors to get white on your monitor). The color specification model used by printers, on the other hand, is called **subtractive** (absence of all colors gives white, which is the color of the paper on which an image is printed). For more information on the various color-specification models, see the [additive color](https://en.wikipedia.org/wiki/Additive_color) and [subtractive color](https://en.wikipedia.org/wiki/Subtractive_color) entries in Wikipedia. Here we’ll focus on the RGB color model, which is used to specify colors in all graphics applications.

**The RGB Color Cube**

The process of generating colors with three basic components is based on the RGB color cube, which is shown in Figure 4.17. The three dimensions of the color cube correspond to the three basic colors. The cube’s corners are assigned each of the three primary colors, their complements, and the colors black and white. Complementary colors are easily calculated by subtracting their basic colors from 255. For example, the color (0, 0, 255) is a pure blue tone. Its complementary color is (255 – 0, 255 – 0, 255 – 255) or (255, 255, 0), which is a pure yellow
tone. Blue and yellow thus are mapped to opposite corners of the cube. The same is true for red and cyan, green and magenta, and black and white. If you add a color to its complement, you get white.

**Figure 4.17**
Color specification of the RGB color cube

![Diagram of an RGB color cube with vertices labeled: Green (0, 255, 0), Cyan (0, 255, 255), Blue (0, 0, 255), White (0, 0, 0), Yellow (255, 255, 0), Black (255, 255, 255), Red (255, 0, 0), and Magenta (255, 0, 255).]

Notice that the components of the colors at the corners of the cube have either zero or full intensity. As you move from one corner to another along the same edge of the cube, only one of its components changes value. For example, as you move from the green to the yellow corner, the red component changes from 0 to 255. The other two components remain the same. As you move between these two corners, you get all the available tones from green to yellow (256 in all). Similarly, as you move from the yellow to the red corner, the only component that changes is the green, and you get all the available shades from yellow to red. As you move from one corner to another on the same face of the cube along the diagonal, two components change value. To go from the yellow corner (255, 255, 0) to the cyan corner (0, 255, 255), you must change the red component from 255 to 0 and the blue component from 0 to 255. Finally, as you move along a diagonal of the cube — when you go blue to yellow, for instance — all three components must change value. As you can guess, this is how GDI + calculates the gradients: It draws an (imaginary) line between the two points that represent the starting and ending colors of the gradient in the RGB cube and picks the colors along this line. These colors are then used to generate the gradient.

**Defining Colors**

To manipulate colors, use the Color class of the Framework. This is a shared class, and you need not create new Color objects; just call the appropriate property or method of the Color class. The Color class exposes 128 predefined colors as properties, which you can access by name, and additional members for specifying custom colors. For example, you can define colors by using the FromARGB method of the Color class. This method accepts three arguments, which are the components of the primary colors in the desired color:

```
Color.FromARGB(Red, Green, Blue)
```

The method returns a Color value, which you can assign to a variable of the same type, or use it directly as the value of a Color property. To change the form’s background color to
yellow, you can assign the value returned by the FromARGB method to the BackColor property of a form or control:

```csharp
Form1.BackColor = FromARGB(255, 128, 128)
```

There's another form of the FromARGB method that accepts four arguments. The first argument in the method is the transparency of the color, or the alpha channel (which explains the A in the method's name). This component is similar to the other three color components, in the sense that it can be a value from 0 (totally transparent) to 255 (totally opaque). The other three arguments are the usual red, green, and blue color components.

You can also retrieve the three basic components of a Color value with the R, G, and B methods. (Yes, they’re single-letter method names!) The following statements print the values of the three components of one of the named colors in the Output window:

```csharp
Dim clr As Color = Color.Beige
Debug.WriteLine "Red Component = " & clr.R.ToString
Debug.WriteLine "Green Component = " & clr.G.ToString
Debug.WriteLine "Blue Component = " & clr.B.ToString
```

Image-processing applications read the image’s pixel values, isolate their basic color components, and then process them separately. Then they combine the processed components to produce the new pixel value.

**ALPHA BLENDING**

Besides the red, green, and blue components, a Color value might also contain a transparency component. This value determines whether the color is opaque (255) or transparent (0). In the case of transparent colors, you can specify the degree of transparency. This component is the *alpha component*. The following statement creates a new color value, which is yellow and 25 percent transparent:

```csharp
Dim trYellow As Color
trYellow = Color.FromARGB(192, Color.Yellow)
```

If you want to "wash out" the colors of an image on a form, draw a white rectangle with a transparency of 50 percent or more on top of the image. The size of the rectangle must be the same as the size of the form, so you can use the ClientRectangle form's property to retrieve the area taken by the form. Then create a solid brush with a semitransparent color by using the Color.FromARGB method. The following code segment does exactly that:

```csharp
Dim brush As New SolidBrush(Color.FromARGB(128, Color.White))
Me.CreateGraphics.FillRectangle(brush, Me.ClientRectangle)
```

If you execute these statements repeatedly, the form will eventually become white. Another use of transparent drawing is to place watermarks on images that you’ll publish on the Web. A *watermark* is a string or logo that’s drawn transparently on the image. It doesn’t really disturb the viewers, but it makes the image unusable on another site. It’s a crude but effective way to protect your images on the Web. (If all images have your site's URL or your company name on them, they're useless to anyone else.)
The following statements place a watermark with the string MySite.Com on top of the image of a PictureBox control. The font is fairly large and bold, and the code assumes that the text fits in the width of the image.

```vbnet
Private Sub Button1_Click(...) Handles Button1.Click
    Dim WMFont As New Font("Impact", 36, FontStyle.Bold)
    Dim WMBrush As New SolidBrush(Color.FromArgb(64, 192, 255, 255))
    PictureBox1.CreateGraphics.DrawString("
        MySite.com", WMFont, WMBrush, 240, 0)
    WMBrush.Color = Color.FromArgb(128, 0, 0, 0)
    PictureBox1.CreateGraphics.DrawString("
        MySite.com", WMFont, WMBrush, 10, 320)
End Sub
```

The preceding statements print the logo at two locations on the image of the PictureBox1 control with different colors, as shown in Figure 4.18. Run the ImageWatermarks project and run it to see the watermarked image, or open the Watermark.tif image in the project's folder.

**Figure 4.18**
Watermarking an image with a semitransparent string

You can also experiment with the watermark's size, color, and transparency. You can combine these statements with a simple program that scans all the images in a folder, to write an application that watermarks a large number of files en masse. Notice that if you click the Watermark Image button repeatedly, the watermark will become more and more evident, because the same string will be printed on top of the previous one, in effect reducing the transparency effect.

Another interesting application of transparency is to superimpose a semitransparent drawing over an opaque one. Figure 4.19 shows some text with a 3D look. To achieve this effect, you render a string by using a totally opaque brush. Then you superimpose the same string drawn with a partially transparent brush. The superimposed string is displaced by a few pixels in relation to the first one. The amount of displacement, its direction, and the colors you use determine the type of 3D effect (raised or depressed). The second brush can have any color, as long as the color combination produces a pleasant effect. The strings shown in Figure 4.19 were generated with the TextEffects project (via the Draw Semi-Transparent Text button), which was discussed earlier in this tutorial. If you run the application and look at the rendered strings...
carefully, you’ll see that they’re made up of three colors. The two original colors appear around
the edges. The inner area of each character is what the transparency of the second color allows
us to see.

**Figure 4.19**
Creating a 3D effect by superimposing transparency on an opaque and a semitransparent string

The code behind the Draw Semi-Transparent Text button is quite simple, really. First it
draws the string with the solid blue brush:

```vbnet
brush = New SolidBrush(Color.FromARGB(255, 0, 0, 255))

Then another instance of the same string is drawn, this time with a different brush:

```vbnet
brush.Color = Color.FromARGB(192, 0, 255, 255)
```

This is a semitransparent shade of cyan. The two superimposed strings are displaced a little
with respect to one another. The statements in Listing 4.17 produced the strings of Figure 4.19.

**Listing 4.17:** Simple Text Effects with Transparent Brushes

```vbnet
Dim G As Graphics
Dim brush As SolidBrush

G = GetGraphicsObject()
G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
Dim drawFont As Font
Dim drawString As String = "Visual Basic 2008"
' Draw string
brush = New SolidBrush(Color.FromArgb(255, 0, 0, 255))
drawFont = New Font("Comic Sans MS", 60, Drawing.FontStyle.Bold)
' Draw string
G.DrawString(drawString, drawFont, brush, 10, 30)
brush.Color = Color.FromArgb(192, 0, 255, 255)
```
' Draw same string with a displacement (-3, -3) pixels
G.DrawString(drawString, drawFont, brush, 7, 27)
brush.Color = Color.FromArgb(255, 0, 0, 255)
G.DrawString(drawString, drawFont, brush, 10, 130)
brush.Color = Color.FromArgb(128, 0, 255, 255)
G.DrawString(drawString, drawFont, brush, 7, 127)
' Draw same string with a displacement (3, 3) pixels
brush.Color = Color.FromArgb(255, 128, 64, 255)
G.DrawString(drawString, drawFont, brush, 10, 230)
brush.Color = Color.FromArgb(128, 255, 128, 64)
G.DrawString(drawString, drawFont, brush, 13, 233)
Me.Invalidate()

The Image Object

Images are two-dimensional arrays that hold the color values of the pixels making up the image. This isn’t how images are stored in their respective files: JPG or JPEG (Joint Photographic Experts Group), GIF (Graphics Interchange Format), TIFF (Tagged Image File Format), and so on, but it’s a convenient abstraction for the developer. To access a specific pixel of an image, you need to specify only the horizontal and vertical coordinates of the desired pixel. Let’s turn our attention to images, starting with the discussion of the Image object. Each pixel is a Long value; the first byte is the pixel’s alpha value and the other three bytes are the red, green, and blue components.

The Image property of the PictureBox or Form control is an Image object, and there are several ways to create such an object. You can declare a variable of the Image type and then assign the Image property of the PictureBox control or the Form object to the variable:

```vbnet
Dim img As Image
img = PictureBox1.Image
```

The `img` Image variable holds the bitmap of the `PictureBox1` control. This code segment assumes that an image was assigned to the control at design time. As you will see shortly, you can call the `Save` method of the Image class to save the image to a disk file.

You can also create a new Image object from an image file by using the Image class’s `FromFile` method:

```vbnet
Dim img As Image
img = Image.FromFile("Butterfly.jpg")
```

After the `img` variable has been set up, you can assign it to the Image property of a PictureBox control:

```vbnet
PictureBox1.Image = img
```

Properties

The Image class exposes several members, some of which are discussed next. Let’s start with a few simple properties and then we’ll examine the methods of the Image class.
**HorizontalResolution, VerticalResolution**
These are read-only properties that return the horizontal and vertical resolution of the image, respectively, in pixels per inch.

**Width, Height**
These are read-only properties that return the width and height of the image, respectively, in pixels. If you divide the dimensions of the image (properties Width and Height) by the corresponding resolutions (properties HorizontalResolution and VerticalResolution), you’ll get the actual size of the image in inches — the dimensions of the image when printed, for instance.

**PixelFormat**
This is another read-only property that returns the image’s pixel format, which determines the quality of the image. There are many pixel formats and they’re all members of the PixelFormat enumeration. For now, I assume that you’re using a color display with a depth of 24 bits per pixel. Images with 24-bit color are of the Format24bppRgb type. Rgb stands for red, green, blue, and 24bpp stands for 24 bits per pixel. Each of the basic colors in this format is represented by 1 byte (8 bits).

**Methods**
In addition to the basic properties, the Image class exposes methods for manipulating images, which are discussed next.

**RotateFlip**
This method rotates and/or flips an image, and its syntax is the following, where the type argument determines how the image will be rotated:

```csharp
Image.RotateFlip(type)
```

This argument can have one of the values of the RotateFlipType enumeration, shown in Table 4.6.

To vertically flip the image displayed on a PictureBox control, use the following statement:

```csharp
PictureBox1.Image.RotateFlip(RotateFlipType.RotateNoneFlipY)
```

The Invalidate method redraws the control, and you must call it to display the new (flipped) image on the control. The first statement rotates the image, but it doesn’t refresh the control. If you omit the call to the Refresh method, you won’t see the effect of the RotateFlip method. If you switch to another window and then back to the application’s window, the image will be flipped because Windows refreshes the form automatically when it brings it to the top of the z-order (in front of all other windows).

**GetThumbnailImage**
This method returns the thumbnail of the specified image. The thumbnail is a miniature version of the image, whose exact dimensions you specify as arguments. Thumbnail images are used as
visual enhancements for selecting images in dialog boxes. The thumbnail takes a small fraction of the space taken by the actual image, and we can display many thumbnails on a form to let the user select the desired one(s) instead of selecting filenames. Thumbnails are usually stored in the same folder as the images and are updated as needed. You can also create a hidden sub-folder under each folder of images and populate the subfolder with the thumbnails. Of course, you must also maintain the list of thumbnails as images are added, modified, and removed. Alternatively, you can create the thumbnails as needed and display them to the user, without having to worry about maintaining the thumbnail version of each image. This approach works quite well on fast systems, unless you run into a folder with thousands of large images.

Table 4.6: The RotateFlipType Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate180FlipNone</td>
<td>Rotates image by 180 degrees</td>
</tr>
<tr>
<td>Rotate180FlipX</td>
<td>Rotates image by 180 degrees and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate180FlipXY</td>
<td>Rotates image by 180 degrees and then flips it vertically and horizontally</td>
</tr>
<tr>
<td>Rotate180FlipY</td>
<td>Rotates image by 180 degrees and then flips it vertically</td>
</tr>
<tr>
<td>Rotate270FlipNone</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees)</td>
</tr>
<tr>
<td>Rotate270FlipX</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate270FlipXY</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it vertically and horizontally</td>
</tr>
<tr>
<td>Rotate270FlipY</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it vertically</td>
</tr>
<tr>
<td>Rotate90FlipNone</td>
<td>Rotates image by 90 degrees</td>
</tr>
<tr>
<td>Rotate90FlipX</td>
<td>Rotates image by 90 degrees and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate90FlipXY</td>
<td>Rotates image by 90 degrees and then flips it horizontally and vertically</td>
</tr>
<tr>
<td>Rotate90FlipY</td>
<td>Rotates image by 90 degrees and then flips it vertically</td>
</tr>
<tr>
<td>RotateNoneFlipNone</td>
<td>No rotation and no flipping</td>
</tr>
<tr>
<td>RotateNoneFlipX</td>
<td>Flips image horizontally</td>
</tr>
<tr>
<td>RotateNoneFlipXY</td>
<td>Flips image vertically and horizontally</td>
</tr>
<tr>
<td>RotateNoneFlipY</td>
<td>Flips image vertically</td>
</tr>
</tbody>
</table>
The syntax of the `GetThumbnailImage` method is as follows:

```csharp
Image.GetThumbnailImage(width, height, Abort, Data)
```

The first two arguments are the dimensions of the thumbnail. The other two arguments are callbacks, which are used when the process is aborted. Because thumbnails don't take long to generate, we'll ignore these two arguments for the purposes of this book and we'll set them both to Nothing. These two arguments enable you to request the generation of a large number of thumbnails, without waiting for each thumbnail to be generated. As soon as each thumbnail is generated, a user-supplied procedure (the callback procedure) is called.

The following statements create a thumbnail of the image selected by the user and display it on a PictureBox control. To test these statements, place a PictureBox and a Button on the form. Then add an instance of the Open dialog box to the form and insert the following statements in the Button's `Click` event handler:

```csharp
' Display the FileOpen dialog box
Dim img As Image
img = Image.FromFile(OpenFileDialog1.FileName)
PictureBox1.Image = img.GetThumbnailImage(32, 32, Nothing, Nothing)
```

**SAVE**

If your application processes the displayed image during the course of its execution and you want to save the image, you can use the `Save` method of the Image object. The simplest syntax of the `Save` method accepts a single argument, which is the path of the file in which the image will be saved:

```csharp
Image.Save(path)
```

To save the contents of the `PictureBox1` control to a file, you must use a statement like the following:

```csharp
PictureBox1.Image.Save("c:\tmpImage.bmp")
```

The image will be saved in BMP format, not because of the specified file extension (which could be anything), but because this is the default format of the `Save` method. Another form of the `Save` method allows you to specify the format in which the image will be saved, where the `format` argument's value can be one of the members of the `ImageFormat` enumeration:

```csharp
PictureBox1.Image.Save("c:\tmpImage.bmp", format)
```

The fully qualified name of the enumeration is `System.Drawing.Imaging.ImageFormat`, so you should import the library `System.Drawing.Imaging` into any project that uses the enumerations mentioned here, so that you won't have to fully qualify the name of the enumeration.

The `ImageFormat` enumeration contains members for all common image formats (see Table 4.7). After you import the `System.Drawing.Imaging` namespace to your project, use the following statement to save the image on the `PictureBox1` control in GIF format:

```csharp
PictureBox1.Image.Save("c:\tmpImage.gif", ImageFormat.Gif)
```
### Table 4.7: The ImageFormat Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bmp</td>
<td>Bitmap image</td>
<td>BMP</td>
</tr>
<tr>
<td>Emf</td>
<td>Enhanced Metafile Format</td>
<td>EMF</td>
</tr>
<tr>
<td>Exif</td>
<td>Exchangeable Image Format</td>
<td>EXIF</td>
</tr>
<tr>
<td>Gif</td>
<td>Graphics Interchange Format</td>
<td>GIF</td>
</tr>
<tr>
<td>Icon</td>
<td>Windows icon</td>
<td>ICO</td>
</tr>
<tr>
<td>Jpeg</td>
<td>Joint Photographic Experts Group format</td>
<td>JPEG, JPG</td>
</tr>
<tr>
<td>MemoryBmp</td>
<td>Saves the image to a memory bitmap</td>
<td></td>
</tr>
<tr>
<td>Png</td>
<td>W3C Portable Network Graphics format</td>
<td>PNG</td>
</tr>
<tr>
<td>Tiff</td>
<td>Tagged Image File Format</td>
<td>TIF</td>
</tr>
<tr>
<td>Wmf</td>
<td>Windows metafile</td>
<td>WMF</td>
</tr>
</tbody>
</table>

### VB 2010 at Work: The Thumbnails Project

You can combine the `GetThumbnailImage` method of the Image object with the techniques described in the tutorial “Manipulating Files and Folders” to scan a folder, retrieve all the image files, and create a thumbnail for each. As for displaying them, I suggest that you create as many PictureBox controls as there are images in the folder and then arrange them horizontally and vertically on a form.

Because this isn’t a trivial project, I have included a sample project that demonstrates how to display thumbnails instead of filenames. The project is called Thumbnails, and you will find among this tutorial’s projects. I copied the CustomExplorer project of the tutorial “Making the Most of the ListView and TreeView Controls,” renamed the main form to ThumbnailsForm, and removed the ListView control in which the names of the files in the selected folder were displayed. In its place, the program displays the PictureBox controls with the thumbnails. When the user clicks an image, the program loads the selected image on the PictureBox control of another form and displays it. Figure 4.20 shows the main form of the Thumbnails application. You can double-click a thumbnail to preview the corresponding image on another form.

Then I adjusted the code to accommodate the display of thumbnails instead of filenames. The `ShowFilesInFolder()` subroutine of the original application displayed the names of the files in the current folder on a ListBox control. This subroutine was replaced by the `ShowImagesInFolder()` subroutine, which is shown in Listing 4.18.
Figure 4.20
The Thumbnails application displays the images in a folder as thumbnails.

Listing 4.18: The ShowImagesInFolder Subroutine

```vba
Sub ShowImagesInFolder()
    Dim file As String
    Dim FI As FileInfo
    Dim PBox As PictureBox, img As Image
    Dim thmbLeft As Integer = 10
    Dim thmbTop As Integer = 10
    Dim PictureWidth As Integer = 64
    Panel1.Controls.Clear()
    Me.Invalidate()
    Dim pnlWidth As Integer = Panel1.Width
    For Each file In Directory.GetFiles(Directory.GetCurrentDirectory)
        FI = New FileInfo(file)
        If FI.Extension = ".GIF" Or _
            FI.Extension = ".JPG" Or _
            FI.Extension = ".TIFF" Then
            PBox = New PictureBox()
            PBox.SizeMode = PictureBoxSizeMode.Zoom
            PBox.BorderStyle = BorderStyle.FixedSingle
            img = Image.FromFile(FI.FullName)
            PBox.Image = img.GetThumbnailImage(_,
                PictureWidth, _,
                PictureWidth * (img.Height / img.Width), _
            )
            PBox.Location = New Point(thmbLeft, thmbTop)
            Panel1.Controls.Add(PBox)
            thmbTop += PictureWidth + 5
        End If
    Next
End Sub
```
Nothing, Nothing)
If thmbLeft + PictureWidth > pnlWidth Then
    thmbLeft = 10
    thmbTop += PictureWidth + 10
End If
PBox.Left = thmbLeft
PBox.Top = thmbTop
PBox.Width = PictureWidth
PBox.Height = PictureWidth
PBox.Visible = True
PBox.Tag = FI.FullName
Me.Controls.Item("Panel1"). Controls.Add(PBox)
AddHandler PBox.Click, New System.EventHandler(AddressOf OpenImage)
    thmbLeft += PictureWidth + 10
Application.DoEvents()
End If
FoldersList.Items.Add(FI.Name)
Next
End Sub

The PictureBox controls with the thumbnails are not added directly on the form; instead, they're added to a Panel control with its AutoScroll property set to True, so that users can scroll them independently of the other controls on the form. The subroutine starts by removing all controls from the Panel1 control. Then the code goes through each file in the selected folder and examines its extension. If it's JPG, GIF, or TIFF (you can add more file extensions if you want), it creates a new PictureBox control, sets its size and location, loads the thumbnail of the image, and then adds it to the Controls collection of the Panel1 control. Each image's path is stored in the PictureBox control's Tag property, and it is retrieved later to load the image on the second form, where it can be previewed.

Notice that the code adds a handler for the Click event of each PictureBox control. All the PictureBox controls share a common handler for their Click event, the OpenImage() subroutine. This subroutine reads the selected image's path from the Tag property of the control that fired the Click event and displays the corresponding image on the auxiliary form. The implementation of the OpenImage() subroutine is shown here:

Sub OpenImage(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim imgForm As New previewForm()
    imgForm.PictureBox1.Image = Image.FromFile(sender.Tag)
    imgForm.Show()
End Sub

previewForm is the name of the second form of the application, where the selected image is previewed. If you need more information about this project, please review the material in Chapter 6 of "Mastering Visual Basic 2010," which explains how to create instances of controls...
at runtime. This application is more of an exercise on designing dynamic forms instead of a demo of the GetThumbnailImage method, but it’s an interesting application, and some readers might have a good use for the techniques demonstrated here. Notice that all the thumbnail bitmaps have a width of 64 pixels. To avoid possible distortion, I set the height of the thumbnail to a value that’s close to 64 pixels, but proportionate to the image’s width-to-height ratio.

**Exchanging Images through the Clipboard**

Whether you use bitmap images or create graphics from scratch with the Framework’s drawing methods, sooner or later you’ll want to exchange them with other Windows applications. To do so, you use the Clipboard and its GetImage and SetImage methods. The SetImage method accepts an image object as an argument and places it on the Clipboard. To copy the bitmap displayed on the PictureBox1 control to the Clipboard, use the following statement:

```csharp
Clipboard.SetImage(PictureBox1.Image)
```

The GetImage method returns the image on the Clipboard as an Image object. To read the bitmap stored in the Clipboard and display it on the PictureBox1 control, you must use a statement like the following:

```csharp
PictureBox1.Image = Clipboard.GetImage
```

Another interesting method of the Clipboard object is the GetDataObject method, which allows you to find out whether the Clipboard contains data of a specific type. This method returns an object of the IDataObject type, which in turn exposes the GetData and GetDataPresent methods. The GetData method returns the data on the Clipboard in the format specified by its argument, which is a member of the DataFormats enumeration (Bitmap, WaveAudio, RTF, and so on). The GetDataPresent method also accepts as an argument a member of the DataFormats enumeration and returns a True/False value indicating whether the Clipboard’s contents are of the specific type.

The GetImage method will attempt to read the Clipboard’s data as an Image object. To read the bitmap stored in the Clipboard and display it on the PictureBox1 control, you should make sure that the Clipboard contains an image before calling the GetImage method:

```csharp
If Clipboard.GetDataObject.GetDataPresent(DataFormats.Bitmap) Then
    PictureBox1.Image = Clipboard.GetImage
Else
    MsgBox("The Clipboard doesn't contain a bitmap!")
End If
```

**ACCESSING THE CLIPBOARD WITH THE MY OBJECT**

The Clipboard access techniques discussed in the preceding section are those exposed by the Framework’s Clipboard class. The My object offers a simpler alternative through its My.Computer.Clipboard component. This component exposes the ContainsXXX properties, which return True if the Clipboard contains a specific format (ContainsText for text, ContainsImage for images, and so on). To retrieve the corresponding item from the Clipboard, use the GetXXX method (GetText for text, GetImage for images, and so on). The
following If statement retrieves the image stored in the Clipboard, but only if the Clipboard
contains image data:

```vbnet
Dim img As Image = Nothing
If My.Computer.Clipboard.ContainsImage Then
End If
```

If the Clipboard doesn’t contain an image, the `img` variable will have a value of Nothing.

One of this tutorial’s sample projects is the ImageClipboard project, which demonstrates
how to exchange images between a VB application and any other image-aware application
running under Windows through the Clipboard. You can copy the image displayed in a Pic-
tureBox control on the application’s main form to the Clipboard and then paste it to another
application, or copy an image in any image-processing application and paste it on the same Pic-
tureBox. The application is straightforward, and you can open it with Visual Studio to examine
its code. This example concludes our discussion of the Image object. In the following section,
you’ll explore the Bitmap object, and you’ll learn how to access and manipulate individual pix-
els in an image.

**The Bitmap Object**

The Image class doesn’t provide any methods for manipulating a bitmap; it’s a class for storing
bitmaps only. The Bitmap class, on the other hand, provides methods that allow you to read
and set its pixels. In the last section of this tutorial, you’re going to build an image-processing
application. The Bitmap object’s constructor is heavily overloaded, and you can create empty
bitmaps with specific properties or import images from files and streams. Two of the simplest
forms of the Bitmap object’s constructors are shown here:

```vbnet
Dim bmp As New Bitmap(filename)
Dim bmp As New Bitmap(stream)
```

They both create a new Bitmap object and initialize it to the bitmap of an image. This image
is read from a file (with the first form of the constructor) or from a Stream object (with the sec-
ond form of the constructor). The properties of the Bitmap object (its dimensions and color
depth) are determined by the image assigned to it. You can also create empty bitmaps with
specific properties by using the following form of the constructor:

```vbnet
Dim bmp As New Bitmap(width, height, pixelformat)
```

The first two arguments are the bitmap’s dimensions in pixels, and the last argument is the
same as the `PixelFormat` property of the Image object. An image’s pixel format is fixed; you
can’t change it. When you create an empty Bitmap object, however, you can specify the format
of its pixels. You can also retrieve an image’s bitmap by initializing the Bitmap object with an
Image object, where `img` is a properly initialized Image object:

```vbnet
Dim bmp As New Bitmap(img)
```

The Bitmap object doesn’t expose a Graphics property, and therefore you can’t draw any
shapes directly on a bitmap. The Bitmap object, however, exposes two methods for accessing
its pixels: the GetPixel and SetPixel methods. The syntax of these two methods is as follows, where \( X \) and \( Y \) are the coordinates of the pixel whose value you’re reading, or setting:

\[
color = \text{Bitmap}.\text{GetPixel}(X, Y) \\
\text{Bitmap}.\text{SetPixel}(X, Y, \text{color})
\]

The GetPixel method returns the color of the specified pixel, whereas the SetPixel method sets the pixel’s color to the specified value. (You’ll see how these two methods are used in the following section.) To rotate the bitmap, you can use the RotateFlip method, whose syntax is identical to the syntax of the RotateFlip method of the Image object.

It’s possible to place graphics on a bitmap by using the drawing methods, but you must first create a Graphics object that represents the Bitmap object’s surface. One way to obtain a Graphics object for the bitmap is to call the Graphics class’s FromImage method, passing the Bitmap object as an argument:

\[
\text{Dim G As Graphics} \\
G = \text{Graphics}.\text{FromImage(bmp)}
\]

After you obtain the Graphics object, you can draw on it by using all the drawing methods discussed in the earlier in this tutorial. Eventually, you’ll have to display the bitmap on a control to see what it looks like. The following statements create an Image object based on the bitmap and display it on a PictureBox control:

\[
\text{Dim img As Image} \\
\text{img} = \text{CType(bmp, Image)} \\
\text{PictureBox1}.\text{Image} = \text{img}
\]

When you’re finished editing the bitmap, you can save it as an image file via the Save method of the Bitmap object. The Save method accepts as arguments the path of the file in which the image will be stored and the type of the image:

\[
\text{Bitmap}.\text{Save(path, imageType)}
\]

The second argument is a member of the ImageFormat enumeration. The project BitmapManipulation, whose main form is shown in Figure 4.21, demonstrates how to create a bitmap in memory from within your code, save it to a file, and load the image. I hard-coded the file’s path for the purposes of a simple demo, and it’s always BitmapImage.jpg.

**Figure 4.21**
The BitmapManipulation application demonstrates how to draw on a bitmap from within your code.
The JPG format compresses the image at the expense of its quality. If you carefully examine the image saved and reloaded to the PictureBox control, you’ll notice artifacts. Change the image format to TIFF, and you’ll see that this format doesn’t sacrifice image quality for size compression.

An interesting application of this technique of generating bitmaps in memory is to create graphics on-the-fly. For example, you can create a stack of boxes, books, and so on to indicate the sales volume in a period (the higher the volume, the taller the stack). The exact image depends on some live data and is different every time. Instead of storing dozens of images and selecting the proper one every time, create the image on-the-fly. The same technique can be used to create fancy counter images on web pages.

One last interesting method of the Bitmap object is the MakeTransparent method, which accepts a color as an argument and treats it as transparent. Any areas of the bitmap you want to treat as transparent (for example, as “holes” in the bitmap) should be filled with this color. When this image is placed on a form or another image, the transparent areas allow the underlying colors to show through.

Processing Bitmaps

A bitmap is a two-dimensional array of color values. These values are stored in disk files, and when an image is displayed on a PictureBox or Form control, each of its color values is mapped to a pixel on the PictureBox or form. This is true when the image isn’t resized, of course. When the image is resized (when displayed on a PictureBox control with its SizeMode property set to Stretch or Zoom, for example), the mapping between the monitor pixels and image pixels is no longer one to one. The image is resized via some interpolation technique. Yet the Image object returned by the control’s Image property isn’t affected; the control “sees” the original image.

As you’ll see, image processing is nothing more than simple arithmetic operations on the values of the image’s pixels. The ImageProcessing application we’ll build to demonstrate the various image-processing techniques doesn’t have the features of a professional application, but it demonstrates the principles of these techniques and can be used as a starting point for custom applications.

We’ll build a simple image-processing application that can read all the image types that the Framework can handle (BMP, GIF, TIFF, JPG, and so on), process them, and then display the processed images. There are simpler ways to demonstrate the pixel-manipulation methods, but image processing is an intriguing topic, and I hope you’ll experiment with the techniques presented in this section.

Let’s look at a simple technique: the inversion of an image’s colors. To invert an image, you must change all pixels to their complementary colors — black to white, green to magenta, and so on. (The complementary colors are on opposite corners of the RGB cube.)

To calculate complementary colors, you subtract each of the three color components from 255. For example, a pure green pixel whose value is (0, 255, 0) will be converted to (255 – 0, 255 – 255, 255 – 0) or (255, 0, 255), which is magenta. Similarly, a mid-yellow tone (0, 128, 128) will be converted to (255 – 0, 255 – 128, 255 – 128) or (255, 127, 127), which is a mid-brown tone. If you apply this color transformation to all the pixels of an image, the result will be the negative of the original image (what you’d see if you looked at the negative, back in the days of film cameras).

Other image-processing techniques aren’t as simple, but image processing is generally as straightforward as arithmetic operations on the image’s pixels. After we go through the Image-Processing application, you’ll probably come up with your own techniques and be able to implement them.
The image on the control won’t be refreshed until the outer loop has finished. As a result, users can’t see the progress of the operation; they will see the new image after all its pixels have been processed.

To force the PictureBox control to refresh its image, you must call the Refresh method. This method, however, isn’t instant. If you insert it in the inner loop, it will make the processing time simply unacceptable. You can insert this statement between the two Next statements, so that users will see each new column of pixels as they’re processed. Even so, the Refresh method introduces a substantial delay. On my computer, it took less than 2 seconds to copy the pixels of a 1,024 × 768 image from one PictureBox control to another. When I introduced a call to the Refresh method after processing an entire column of pixels, the time jumped to 8 seconds.

It’s actually much faster to update a ProgressBar control from within your code than to update an image. The obvious solution is to avoid refreshing the PictureBox control too often, but then again you’re giving up the immediate feedback.

**VB 2010 at Work: The ImageProcessing Project**

The application we’ll develop in this section is called ImageProcessing, and its main form is shown in Figure 4.22. It’s not a professional tool, but it can be easily implemented in Visual Basic 2008 and it will give you the opportunity to explore various image-processing techniques on your own.

To process an image with the application, choose File >> Open to load it to the PictureBox control and then select the type of action from the Process menu. You can also zoom in or out by using the commands of the View menu, and you can rotate the image by using the commands of the Rotate menu.

The sample application implements the following image-processing techniques:

**Smooth**

Reduces the amount of detail in the image by smoothing areas with abrupt changes in color and/or intensity. Smoothing blurs the image, and extreme smoothing results in total loss of detail.
Sharpen
Brings out the detail in the image by amplifying the differences between similarly colored pixels.

Emboss
Adds a raised (embossed) look to the image.

**Figure 4.22**
The Image-Processing application demonstrates several image-processing techniques, all implemented with VB 2008.

Diffuse
Gives the image a “painterly” look.

Next, let’s look at how each algorithm works and how it’s implemented in Visual Basic.

**How the Application Works**
Let’s start with a general discussion of the application’s operation before looking at the actual code. After the image is loaded on a PictureBox control, you can access the values of its pixels with the `GetPixel` method of the Bitmap object that holds the image. The `GetPixel` method returns a Color value; you can use the `R`, `G`, and `B` methods of the Color object to extract the basic color components. This is a time-consuming step, and for most algorithms it must be performed more than once for each pixel.

All image-processing algorithms read a few pixel values and process them to calculate the new value of a pixel. This value is then written into the new bitmap via the `SetPixel` method.

To process an image, we set up two nested loops: an outer loop scans the rows of pixels, and an inner loop scans the pixels in each row. In the inner loop’s body, we calculate the current pixel’s new value, taking into consideration the values of the surrounding pixels. Because of this, we can’t save the new pixel values to the original bitmap. When processing the next pixel, some of the surrounding pixels will have their original values, whereas others will have the new values. As a result, we must create a copy of the original bitmap and use this bitmap to retrieve the original values of the pixels. The processed values are displayed on the bitmap of the PictureBox control, so that you can watch the progress of the processing. The following is the outline of all the algorithms that we’ll implement shortly:

```vbnet
bmp = New Bitmap(PictureBox1.Image)
PictureBox1.Image = bmp
```
Dim tempbmp As New Bitmap(PictureBox1.Image)
Dim pixRow, pixCol As Integer
With tempbmp
    For pixRow = DX To .Height - DX - 1
        For pixCol = DY To .Width - DY - 1
            { calculate new pixel value }
            bmp.SetPixel(pixRow, pixCol, new_pixel_value)
        Next
        If i Mod 10 = 0 Then
            PictureBox1.Invalidate()
        End If
    Next
End With

Here's how it works. First, we create a Bitmap object from the image on the PictureBox control. This is the bmp variable, which is then assigned back to the Image property of the control. Everything you draw on the bmp object will appear on the control's surface. We then create another identical Bitmap object, the tempbmp variable. This object holds the original values of all the pixels of the image.

The two nested loops go through every pixel in the image. In the inner loop's body, we calculate the new value of the current pixel and then write this value to the matching location of the bmp object. The new pixel will appear on the control when we refresh it by calling the control's Invalidate method. This method isn't called every time we display a new pixel. It would introduce a significant delay, so we invalidate the control after processing 10 rows of pixels. This is a good balance between performance and a visual feedback of the process's progress.

We could have displayed a dialog box with a progress bar to indicate the progress of the operation. If a simple indication will do, you can simply display the percentage of the completed work on the form's title bar.

**APPLYING EFFECTS**

In the following sections, you'll find a short description of the algorithm that implements each effect. You can open the ImageProcessing project with Visual Studio and examine the code, which contains a lot of comments explaining the various operations.

**Smoothing Images**

One of the simplest and most common operations in all image-processing programs is the smoothing (or blurring) operation. The smoothing operation is equivalent to low-pass filtering: just as you can cut off a stereo's high-frequency sounds with the help of an equalizer, you can cut off the high frequencies of an image. If you're wondering what the high frequencies of an image are, think of them as the areas with abrupt changes in the image's intensity. These are the areas that are mainly affected by the blurring filter.

The smoothed image contains fewer abrupt changes than the original and looks a lot like the original image seen through a semitransparent glass. Figure 4.23 shows a smoothed image obtained with the ImageProcessing application. This image is a good candidate for smoothing, given that it has a lot of detail, especially in the area of the water drops.

To smooth an image, you must reduce the large differences between adjacent pixels. Let's take a block of 9 pixels, centered on the pixel we want to blur. This block contains the pixel to be blurred and its eight immediate neighbors. Let's assume that all the pixels in this block are green except for the middle one, which is red. This pixel is drastically different from its neighbors, and for it to be blurred, it must be pulled toward the average value of the other
pixels. Taking the average of a block of pixels is, therefore, a good choice for a blurring operation. If the current pixel’s value is similar to the values of its neighbors, the average won’t significantly affect its value. If its value is drastically different, the remaining pixels will “pull” the current pixel’s value toward them. In other words, if the middle pixel were green, the average wouldn’t affect it. Because it’s the only red pixel in the block, however, it will come closer to the average value of the remaining pixels. It will assume a green tone.

Figure 4.23
Smoothing an image reduces its detail, but can make the image less “noisy” and “busy.”

The intensity of blurring depends on the size of the block over which the average is calculated. We used a 3 × 3 block in our example, which yields an average blur. To blur the image even more, use a 5 × 5 block. Even larger blocks will blur the image to the point that useful information is lost. The actual code of the Smooth operation scans all the pixels of the image (excluding the edge pixels that don’t have neighbors all around them) and takes the average of their RGB components (one value per component). It then combines the three values by using the method Color.FromArgb to produce the new value of the pixel.

Sharpening Images
Because the basic operation for smoothing an image is addition, the opposite operation will result in sharpening the image. The sharpening effect is more subtle than smoothing, but is also more common and more useful. Nearly every image published, especially in monochrome (“one-color”) publications, must be sharpened to some extent. Sharpening an image consists of highlighting the edges of the objects in it, which are the very same pixels blurred by the previous algorithm. Edges are areas of an image with sharp changes in intensity between adjacent pixels.

In a smooth area of an image, the difference between two adjacent pixels will be zero or a very small number. If the difference is zero, the two pixels are nearly identical, which means that there’s nothing to sharpen. This is called a flat area of the image. If the pixels are on an edge, the difference between two adjacent pixels will be a large value (perhaps negative). This is an area of the image with some degree of detail that can be sharpened. The difference between adjacent pixels isolates the areas with detail and completely flattens the smooth areas. The question now is how to bring out the detail without leveling the rest of the image. How about adding the difference to the original pixel? Where the image
is flat, the difference is negligible, and the processed pixel will be practically the same as the original one. If the difference is significant, the processed pixel will be the original plus a value that’s proportional to the magnitude of the detail. The sharpening algorithm can be expressed as follows:

\[
\text{new_value} = \text{original_value} + 0.5 \times \text{difference}
\]

If you simply add the difference to the original pixel, the algorithm brings out too much detail. You usually add a fraction of the difference; a 50 percent factor is common. You can also use different factors for different components (use a 60 percent factor for the green component and a 40 percent factor for the red component, for example).

**Embossing Images**

To sharpen an image, we add the difference between adjacent pixels to the pixel value. What do you think would happen to a processed image if you took the difference between adjacent pixels only? The flat areas of the image would be totally leveled, and only the edges would remain visible. The result would be an image like the one shown in Figure 4.24. This effect clearly sharpens the edges and flattens the smooth areas of the image. By doing so, it gives depth to the image. The processed image looks as if it’s raised and illuminated from the right side. This effect is known as **embossing** or **bas relief**.

**Figure 4.24**
The Emboss special effect

The actual algorithm is based on the difference between adjacent pixels. For most of the image, however, the difference between adjacent pixels is a small number, and the image will turn black. The Emboss algorithm adds a constant to the difference to bring some brightness to areas of the image that would otherwise be dark. The algorithm can be expressed as follows:

\[
\text{new_value} = \text{difference} + 128
\]

As usual, you can take the difference between adjacent pixels in the same row, adjacent pixels in the same column, or diagonally adjacent pixels. The code that implements the Emboss filter in the ImageProcessing application uses differences in the x and y directions. (Set one of the variables *DispX* or *DispY* to 0 to take the difference in one direction only.)
**Diffusing Images**

The Diffuse special effect is different from the previous ones, in the sense that it’s not based on the sums or the differences of pixel values. This effect uses the Random class to introduce some randomness to the image and give it a “painterly” look, as demonstrated in Figure 4.25. You can control the intensity of the effect by applying the same type of processing repeatedly to the image.

**Figure 4.25**
The Diffuse special effect gives the image a painterly look.

This time, we won’t manipulate the values of the pixels. Instead, the current pixel will assume the value of another one, selected randomly in its $5 \times 5$ neighborhood with the help of the Random class.

The Diffuse algorithm is the simplest one. It generates two random variables, $DX$ and $DY$, in the range $-3$ to $3$. These two variables are added to the coordinates of the current pixel to yield the coordinates of another pixel in the neighborhood. The original pixel is replaced by the value of the pixel that is $(DX, DY)$ pixels away.

Open the ImageProcessing application to explore the code that implements the various effects. Change the parameters of the various algorithms and see how they affect the processed image. You can easily implement new algorithms by inserting the appropriate code in the inner loop’s body. The rest of the code remains the same. Some simple ideas include clipping one or more colors (force the red color component of each pixel to remain within a range of values), substituting one component for another (replace the red component of each pixel with the green or blue component of the same pixel), inverting the colors of the image (subtract all three color components of each pixel from 255), and so on. With a little imagination, you can create interesting effects for your images.