

48

Graphics with GDI+

WHAT'S IN THIS CHAPTER?

- Principles of drawing
- Colors and the safety palette
- Pens and brushes
- Lines and simple shapes
- BMP images and other image files
- Drawing Text
- Fonts and font families
- Dealing with printing

You will notice that quite a number of chapters in this book deal with user interaction and the .NET Framework. Chapter 39, “Windows Forms,” focused on how to display either a dialog box or a Single Document Interface (SDI) or Multiple Document Interface (MDI) window, and how to place various controls such as buttons, text boxes, and list boxes. It also looked at how to work with data in Windows Forms using a number of the Windows Forms controls that work with the disparate data sources that you might encounter.

Although these standard controls are powerful and, by themselves, quite adequate for the complete user interface for many applications, some situations require more flexibility. For example, you might want to draw text in a given font in a precise position in a window, or display images without using a picture box control, or draw simple shapes or other graphics. None of this can be done with the controls discussed in Chapter 39. To display that kind of output, the application must instruct the operating system what to display and where in the window to display it.

In the process, you need to use a variety of helper objects, including pens (to define the characteristics of lines), brushes (to define how areas are filled in), and fonts (to define the shape of the characters of text). This chapter also goes into some detail about how devices interpret and display different colors.

The chapter starts, however, by discussing a technology called Graphics Device Interface or *GDI+*. GDI+ consists of the set of .NET base classes that are available to control custom drawing onscreen. These classes arrange for the appropriate instructions to be sent to graphics device drivers to ensure the correct output is placed onscreen (or printed to a hard copy).

UNDERSTANDING DRAWING PRINCIPLES

This section examines the basic principles that you need to start drawing to the screen. It starts by giving an overview of GDI and the underlying technology on which GDI+ is based. It also shows how GDI and GDI+ are related. Then some simple examples are given.

GDI and GDI+

In general, one of the strengths of Windows — and indeed of modern operating systems in general — lies in its ability to abstract the details of particular devices without input from the developer. For example, you do not need to understand anything about your hard drive device driver to programmatically read and write files to and from disk. You simply call the appropriate methods in the relevant .NET classes (or in pre-.NET days, the equivalent Windows API functions). This principle is also true when it comes to drawing. When the computer draws anything to the screen, it does so by sending instructions to the video card. However, hundreds of different video cards are on the market, most of which have different instruction sets and capabilities. If you had to take that into account and write specific code for each video driver, writing any such application would be an almost impossible task. The Windows Graphics Device Interface (GDI) has been around since the earliest versions of Windows because of these reasons.

GDI provides a layer of abstraction, hiding the differences between the different video cards. You simply call the Windows API function to do the specific task, and internally the GDI figures out how to get the client's particular video card to do whatever it is you want when the client runs your particular piece of code. Not only does GDI accomplish this, but if the client has several display devices — for example, monitors and printers — GDI achieves the remarkable feat of making the printer look the same as the screen, as far as the application is concerned. If the client wants to print something instead of displaying it, your application will simply inform the system that the output device is the printer, and then call the same API functions in exactly the same way.

As you can see, the device context (DC) object (covered shortly) is a very powerful object, and you won't be surprised to learn that under GDI *all* drawing had to be done through a device context. The DC was even used for operations that do not involve drawing to the screen or to any hardware device, such as modifying images in memory.

Although GDI exposes a relatively high-level API to developers, it is still an API that is based on the old Windows API, with C-style functions. GDI+, to a large extent, sits as a layer between GDI and your application, providing a more intuitive, inheritance-based object model. Although GDI+ is basically a wrapper around GDI, Microsoft has been able, through GDI+, to provide new features and performance improvements to some of the older features of GDI as well.

The GDI+ part of the .NET base class library is huge, and this chapter barely scratches the surface of its features because trying to cover more than a tiny fraction of the library would have turned this chapter into a huge reference guide that simply listed classes and methods. It is more important to understand the fundamental principles involved in drawing so that you are in a good position to explore the available classes. Full lists of all the classes and methods available in GDI+ are, of course, available in the SDK documentation.



Visual Basic 6 developers are likely to find the concepts involved in drawing quite unfamiliar because Visual Basic 6 focuses on controls that handle their own painting. C++/MFC developers are likely to be in more familiar territory because MFC requires developers to take control of more of the drawing process, using GDI. However, even if you have a strong background in the classic GDI, you will find that a lot of the material presented in this chapter is new.

GDI+ Namespaces

The following table provides an overview of the main namespaces you will need to explore to find the GDI+ base classes.

You should note that almost all the classes and structs used in this chapter are taken from the `System.Drawing` namespace.

NAMESPACE	DESCRIPTION
<code>System.Drawing</code>	Contains most of the classes, structs, enums, and delegates concerned with the basic functionality of drawing.
<code>System.Drawing.Drawing2D</code>	Provides most of the support for advanced 2-D and vector drawing, including anti-aliasing, geometric transformations, and graphics paths.
<code>System.Drawing.Imaging</code>	Contains various classes that assist in the manipulation of images (bitmaps, GIF files, and so on).
<code>System.Drawing.Printing</code>	Contains classes to assist when specifically targeting a printer or print preview window as the “output device.”
<code>System.Drawing.Design</code>	Contains some predefined dialog boxes, property sheets, and other user interface elements concerned with extending the design-time user interface.
<code>System.Drawing.Text</code>	Contains classes to perform more advanced manipulation of fonts and font families.

Device Contexts and the Graphics Object

In GDI, you identify which device you want your output to go to with an object known as the DC. The DC stores information about a particular device and is able to translate calls to the GDI API functions into whatever instructions need to be sent to that device. You can also query the device context to find out what the capabilities of the corresponding device are (for example, whether a printer prints in color or only in black and white), so the output can be adjusted accordingly. If you ask the device to do something it is not capable of, the DC will normally detect this and take appropriate action (which, depending on the situation, might mean throwing an exception or modifying the request to get the closest match that the device is actually capable of using).

However, the DC does not deal only with the hardware device. It acts as a bridge to Windows and is able to take account of any requirements or restrictions placed on the drawing by Windows. For example, if Windows knows that only a portion of your application’s window needs to be redrawn, the DC can trap and nullify attempts to draw outside that area. Because of the DC’s relationship with Windows, working through the device context can simplify your code in other ways.

For example, hardware devices need to be told where to draw objects, and they usually want coordinates relative to the top-left corner of the screen (or output device). Usually, however, your application will be thinking of drawing something at a certain position within the client area (the area reserved for drawing) of its own window, possibly using its own coordinate system. Because the window might be positioned anywhere on the screen, and a user might move it at any time, translating between the two coordinate systems is potentially a difficult task. However, the DC always knows where your window is and is able to perform this translation automatically.

With GDI+, the DC is wrapped up in the .NET base class `System.Drawing.Graphics`. Most drawing is done by calling methods on an instance of `Graphics`. In fact, because the `Graphics` class is the class that is responsible for handling most drawing operations, very little gets done in GDI+ that does not involve a `Graphics` instance somewhere, so understanding how to manipulate this object is the key to understanding how to draw to display devices with GDI+.

Drawing Shapes

This section starts with a short example, `DisplayAtStartup`, to illustrate drawing to an application's main window. The examples in this chapter are all created in Visual Studio 2010 as C# Windows applications. Recall that for this type of project the code wizard gives you a class called `Form1`, derived from `System.Windows.Forms.Form`, which represents the application's main window. Also generated for you is a class called `Program` (found in the `Program.cs` file), which represents the application's main starting point. Unless otherwise stated, in all code samples, new or modified code means code that you have added to the wizard-generated code. (You can download the sample code from the Wrox web site at www.wrox.com.)



In .NET usage, when we are talking about applications that display various controls, the terminology “form” has largely replaced “window” to represent the rectangular object that occupies an area of the screen on behalf of an application. In this chapter, we have tended to stick to the term “window” because in the context of manually drawing items it is more meaningful. We will also talk about the form when we are referring to the .NET class used to instantiate the form/window. Finally, we will use the terms “drawing” and “painting” interchangeably to describe the process of displaying some item onscreen or other display device.

The first example simply creates a form and draws to it in the constructor when the form starts up. Note that this is not actually the best or the correct way to draw to the screen — you will quickly find that this example has a problem because it is unable to redraw anything after starting up. However, this example illustrates quite a few points about drawing without your having to do very much work.

For this example, start Visual Studio 2010 and create a Windows Form Application project. First, set the background color of the form to white. In the example, this line comes after the `InitializeComponent()` method so that Visual Studio 2010 recognizes the line and is able to alter the design view appearance of the form. You can find the `InitializeComponent()` by clicking the arrow icon next to the `Form1.cs` file (this will expand the hierarchy of files related to the `Form1.cs` file). Here, you will find the `Form1.Designer.cs` file. It is in this file that you will find the `InitializeComponent()` method. You could have used the design view to set the background color, but this would have resulted in pretty much the same line being added automatically:



Available for
download on
Wrox.com

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
}
```

[code download DrawingShapes.sln](#)

Then you add code to the `Form1` constructor. You create a `Graphics` object using the form's `CreateGraphics()` method. This `Graphics` object contains the Windows DC that you need to draw with. The device context created is associated with the display device and also with this window:



Available for
download on
Wrox.com

```
public Form1()
{
    InitializeComponent();

    Graphics dc = CreateGraphics();
    Show();
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
}
```

```

    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}

```

[code download DrawingShapes.sln](#)

As you can see, you then call the `Show()` method to display the window. This is really done to force the window to display immediately because you cannot actually do any drawing until the window has been displayed. If the window is not displayed, there is nothing for you to draw onto.

Finally, you display a rectangle at coordinates (0,0) and with width and height 50, and an ellipse with coordinates (0,50) and with width 80 and height 50. Note that coordinates (x,y) translate to x pixels to the right and y pixels down from the top-left corner of the client area of the window — and these coordinates start from the top-left corner of the shape to be displayed.

The overloads that you are using of the `DrawRectangle()` and `DrawEllipse()` methods each take five parameters. The first parameter of each is an instance of the class `System.Drawing.Pen`. A `Pen` is one of a number of supporting objects to help with drawing — it contains information about how lines are to be drawn. Your first pen instructs the system that lines should be the color blue with a width of 3 pixels; the second pen instructs the system that the lines should be red and have a width of 2 pixels. The final four parameters are coordinates and size. For the rectangle, they represent the (x,y) coordinates of the top-left corner of the rectangle in addition to its width and height. For the ellipse, these numbers represent the same thing, except that you are talking about a hypothetical rectangle that the ellipse just fits into, rather than the ellipse itself. Figure 48-1 shows the result of running this code. Of course, because this book is not in color, you cannot see the colors.

Figure 48-1 demonstrates a couple of points. First, you can see clearly where the client area of the window is located. It is the white area — the area that has been affected by setting the `BackColor` property. Notice that the rectangle nestles up in the corner of this area, as you would expect when you specify the coordinates of (0,0) for it. Second, notice that the top of the ellipse overlaps the rectangle slightly, which you would not expect from the coordinates given in the code. The culprit here is Windows itself and where it places the lines that border the rectangle and ellipse. By default, Windows will try to center the line on the border of the shape — that is not always possible to do exactly because the line has to be drawn on pixels (obviously). Normally, the border of each shape theoretically lies between two pixels. The result is that lines that are 1 pixel thick will get drawn just *inside* the top and left sides of a shape, but just *outside* the bottom and right sides — which means that shapes that are next to each other have their borders overlapping by one pixel. You have specified wider lines; therefore, the overlap is greater. It is possible to change the default behavior by setting the `Pen.Alignment` property, as detailed in the SDK documentation, but for these purposes, the default behavior is adequate.

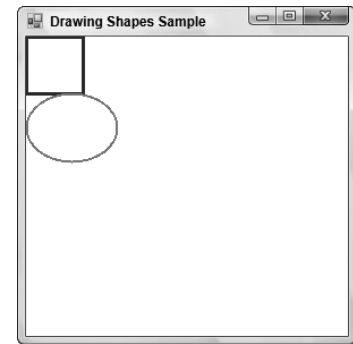


FIGURE 48-1

Unfortunately, if you actually run the sample, you will notice that the form behaves a bit strangely. It is fine if you just leave it there. It is also fine if you drag it around the screen with the mouse. However, if you try minimizing the window and then restoring it, then your carefully drawn shapes just vanish! The same thing happens if you drag another window across the sample so that it only obscures a portion of your shapes. When you drag the other window away again, you will find that the temporarily obscured portion has disappeared and you are left with half an ellipse or half a rectangle!

So what's going on? The problem arises when part of a window is hidden because Windows usually discards immediately all the information concerning exactly what has been displayed. This is something Windows has to do or else the memory usage for storing screen data would be astronomical. A typical computer might be running with the video card set to display 1024 × 768 pixels, perhaps in a 24-bit color mode, which implies that each pixel on the screen occupies 3 bytes — 2.25MB to display the screen (24-bit color

is covered later in this chapter). However, it is not uncommon for a user to work with 10 or 20 minimized windows in the taskbar. In a worst-case scenario, you might have 20 windows, each of which would occupy the whole screen if it was not minimized. If Windows actually stored the visual information those windows contained, ready for when the user restored them, then that would amount to some 45MB! These days, a decent graphics card might have 512MB of memory and be able to cope with that, but it was only a few years ago that 256MB was considered generous in a graphics card — and the excess would need to be stored in the computer's main memory. Many people still have old machines, some of them with only 256MB graphic cards. Clearly, it would not be practical for Windows to manage its user interface like that.

The moment any part of a window is hidden, the “hidden” pixels get lost because Windows frees the memory that was holding those pixels. It does, however, note that a portion of the window is hidden, and when it detects that it is no longer hidden, it asks the application that owns the window to redraw its contents. There are a couple of exceptions to this rule — generally for cases in which a small portion of a window is hidden very temporarily (a good example is when you select an item from the main menu and that menu item drops down, temporarily obscuring part of the window below). In general, however, you can expect that if part of your window is hidden, your application will need to redraw it later.

That is the source of the problem for the sample application. You placed your drawing code in the `Form1` constructor, which is called just once when the application starts up, and you cannot call the constructor again to redraw the shapes when required later on.

When working with Windows Forms server controls, there is no need to know anything about how to accomplish this task. This is because the standard controls are pretty sophisticated, and they are able to redraw themselves correctly whenever Windows asks them to. That is one reason why, when programming controls, you do not need to worry about the actual drawing process at all. If you are taking responsibility for drawing to the screen in your application, you also need to make sure that your application will respond correctly whenever Windows asks it to redraw all or part of its window. In the next section, you modify the sample to do just that.

Painting Shapes Using `OnPaint()`

If the preceding explanation has you worried that drawing your own user interface is going to be terribly complicated, do not worry. Getting your application to redraw itself when necessary is actually quite easy.

Windows notifies an application that some repainting needs to be done by raising a `Paint` event. Interestingly, the `Form` class has already implemented a handler for this event, so you do not need to add one yourself. The `Form1` handler for the `Paint` event will at some point in its processing call up a virtual method, `OnPaint()`, passing to it a single `PaintEventArgs` parameter. This means that all you need to do is override `OnPaint()` to perform your painting.

Although for this example you work by overriding `OnPaint()`, it is equally possible to achieve the same results by simply adding your own event handler for the `Paint` event (a `Form1_Paint()` method, say) — in much the same way as you would for any other Windows Forms event. This other approach is arguably more convenient because you can add a new event handler through the Visual Studio 2010 properties window, saving yourself from typing some code. However, the approach of overriding `OnPaint()` is slightly more flexible in terms of letting you control when the call to the base class window processing occurs, and it allows you to avoid attaching the control's event handler to its own event.

In this section, you create a new Windows Application called `DrawShapes` to do this. As before, you set the background color to white, using the properties window. You will also change the form's text to `DrawShapes Sample`. Then you add the following code to the generated code for the `Form1` class:



```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0,0,50,50);
}
```

```

    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}

```

[code download DrawingShapes.sln](#)

Notice that `OnPaint()` is declared as `protected`, because it is normally used internally within the class, so there is no reason for any other code outside the class to know about its existence.

`PaintEventArgs` is a class that is derived from the `EventArgs` class normally used to pass in information about events. `PaintEventArgs` has two additional properties, of which the more important one is a `Graphics` instance, already primed and optimized to paint the required portion of the window. This means that you do not have to call `CreateGraphics()` to get a DC in the `OnPaint()` method — you have already been provided with one. You will look at the other additional property soon. This property contains more detailed information about which area of the window actually needs repainting.

In your implementation of `OnPaint()`, you first get a reference to the `Graphics` object from `PaintEventArgs`, and then you draw your shapes exactly as you did before. When you start this, you call the base class's `OnPaint()` method. This step is important. You have overridden `OnPaint()` to do your own painting, but it is possible that Windows may have some additional work of its own to do in the painting process — any such work will be dealt with in an `OnPaint()` method in one of the .NET base classes.



For this example, you will find that removing the call to `base.OnPaint()` does not seem to have any effect. However, do not be tempted to leave this call out. You might be stopping Windows from doing its work properly, and the results could be unpredictable.

`OnPaint()` will also be called when the application first starts up and your window is displayed for the first time. Thus, there is no need to duplicate the drawing code in the constructor.

Running this code gives the same results initially as in the previous example, except that now your application behaves properly when you minimize it or hide parts of the window.

Using the Clipping Region

The `DrawShapes` sample from the previous section illustrates the main principles involved with drawing to a window, although the sample is not very efficient. The reason is that it attempts to draw everything in the window, regardless of how much needs to be drawn. Figure 48-2 shows the result of running the `DrawShapes` example and opening another window and moving it over the `DrawShapes` form so part of it is hidden.

However, when you move the overlapping window so that the `DrawShapes` window is fully visible again, Windows will, as usual, send a `Paint` event to the form, asking it to repaint itself. The rectangle and ellipse both lie in the top-left corner of the client area, and so were visible all the time. Therefore, there is actually nothing that needs to be done in this case apart from repainting the white background area. However, Windows does not know that, so it thinks it should raise the `Paint` event, resulting in your `OnPaint()` implementation being called. `OnPaint()` will then unnecessarily attempt to redraw the rectangle and ellipse.

Actually, in this case, the shapes will not be repainted because of the device context. Windows has pre-initialized the device context with information concerning what area actually needed repainting. In the

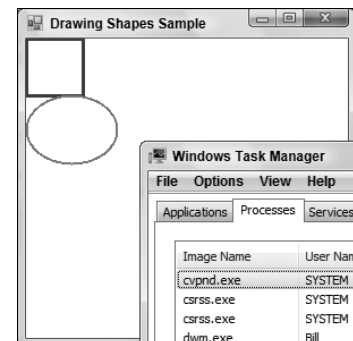


FIGURE 48-2

days of GDI, the region marked for repainting was known as the *invalidated region*, but with GDI+ the terminology has largely changed to *clipping region*. The device context recognizes this region. Therefore, it will intercept any attempts to draw outside this region and not pass the relevant drawing commands on to the graphics card. That sounds good, but there is still a potential performance hit here. You do not know how much processing the device context had to do before it figured out that the drawing was outside the invalidated region. In some cases, it might be quite a lot because calculating which pixels need to be changed to what color can be very processor-intensive (although a good graphics card will provide hardware acceleration to help with some of this).

The bottom line to this is that asking the `Graphics` instance to do some drawing outside the invalidated region is almost certainly wasting processor time and slowing your application down. In a well-designed application, your code will help the device context by carrying out a few simple checks to see if the proposed drawing work is likely to be needed before it calls the relevant `Graphics` instance methods. In this section, you code a new example, `DrawShapesWithClipping`, by modifying the `DisplayShapes` example to do just that. In your `OnPaint()` code, you will do a simple test to see whether the invalidated region intersects the area you need to draw in, and you will call the drawing methods only if it does.

First, you need to obtain the details of the clipping region. This is where an extra property, `ClipRectangle`, on `PaintEventArgs` comes in. `ClipRectangle` contains the coordinates of the region to be repainted and wrapped up in an instance of a struct, `System.Drawing.Rectangle.Rectangle`, which is quite a simple struct. It contains four properties of interest: `Top`, `Bottom`, `Left`, and `Right`. These respectively contain the vertical coordinates of the top and bottom of the rectangle and the horizontal coordinates of the left and right edges.

Next, you need to decide what test you will use to determine whether drawing should take place. You will go for a simple test here. Notice that in your drawing, the rectangle and ellipse are both entirely contained within the rectangle that stretches from point (0,0) to point (80,130) of the client area. Actually, use point (82,132) to be on the safe side because you know that the lines might stray a pixel or so outside this area. So, you will check whether the top-left corner of the clipping region is inside this rectangle. If it is, then you will go ahead and redraw. If it is not, then you won't bother.

The following is the code to do this:



```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 132 && e.ClipRectangle.Left < 82)
    {
        Pen bluePen = new Pen(Color.Blue, 3);
        dc.DrawRectangle(bluePen, 0,0,50,50);
        Pen redPen = new Pen(Color.Red, 2);
        dc.DrawEllipse(redPen, 0, 50, 80, 60);
    }
}
```

[code download DrawingShapes.sln](#)

Note that what is displayed is exactly the same as before. However, performance is improved now by the early detection of some cases in which nothing needs to be drawn. Notice also that the example uses a fairly crude test for whether to proceed with the drawing. A more refined test might be to check separately whether the rectangle or the ellipse needs to be redrawn. However, there is a balance here. You can make your tests in `OnPaint()` more sophisticated, improving performance, but you will also make your own `OnPaint()` code more complex. It is almost always worth putting some test in — because you have written the code, you understand far more about what is being drawn than the `Graphics` instance, which just blindly follows drawing commands.

MEASURING COORDINATES AND AREAS

In the previous example, you encountered the base struct, `Rectangle`, which is used to represent the coordinates of a rectangle. GDI+ actually uses several similar structures to represent coordinates or areas. The following table lists the structs that are defined in the `System.Drawing` namespace.

STRUCT	MAIN PUBLIC PROPERTIES
<code>Point</code> and <code>PointF</code>	<code>X</code> , <code>Y</code>
<code>Size</code> and <code>SizeF</code>	<code>Width</code> , <code>Height</code>
<code>Rectangle</code> and <code>RectangleF</code>	<code>Left</code> , <code>Right</code> , <code>Top</code> , <code>Bottom</code> , <code>Width</code> , <code>Height</code> , <code>X</code> , <code>Y</code> , <code>Location</code> , <code>Size</code>

Note that many of these objects have a number of other properties, methods, or operator overloads not listed here. This section just discusses some of the most important ones.

Point and PointF

`Point` is conceptually the simplest of these structs. Mathematically, it is equivalent to a 2-dimensional vector. It contains two public integer properties, which represent how far you move horizontally and vertically from a particular location (perhaps on the screen), as shown in Figure 48-3.

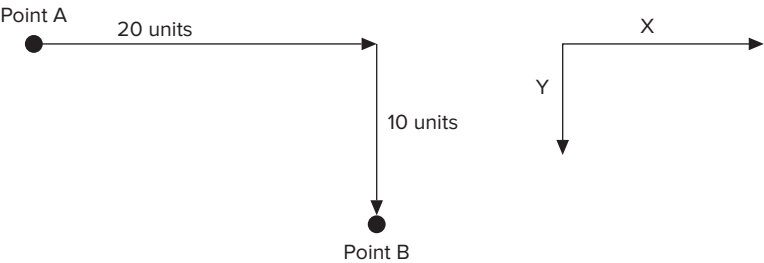


FIGURE 48-3

To get from point A to point B, you move 20 units across and 10 units down, marked as `x` and `y` on the diagram because this is how they are commonly referred to. The following `Point` struct represents that line:

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

`X` and `Y` are read-write properties, which means that you can also set the values in a `Point`, such as this:

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

Note that although conventionally horizontal and vertical coordinates are referred to as `x` and `y` coordinates (lowercase), the corresponding `Point` properties are `X` and `Y` (uppercase) because the usual convention in C# is for public properties to have names that start with an uppercase letter.

`PointF` is essentially identical to `Point`, except that `X` and `Y` are of type `float` instead of `int`. `PointF` is used when the coordinates are not necessarily integer values. A cast has been defined so that you can implicitly convert from `Point` to `PointF`. (Note that because `Point` and `PointF` are structs, this cast involves actually making a copy of the data.) There is no corresponding reverse case — to convert from `PointF` to `Point` you have to copy the values across, or use one of three conversion methods, `Round()`, `Truncate()`, or `Ceiling()`:

```
PointF abFloat = new PointF(20.5F, 10.9F);
// converting to Point
Point ab = new Point();
```

```

ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);
// but conversion back to PointF is implicit
PointF abFloat2 = ab;

```

You might be wondering what a unit is measured in. By default, GDI+ interprets units as pixels along the screen (or printer, whatever the graphics device is). This is how the `Graphics` object methods view any coordinates that they are passed as parameters. For example, the point `new Point(20,10)` represents 20 pixels across the screen and 10 pixels down. Usually these pixels are measured from the top-left corner of the client area of the window, as has been the case in the previous examples. However, that will not always be the case. For example, on some occasions you might want to draw relative to the top-left corner of the whole window (including its border), or even to the top-left corner of the screen. In most cases, however, unless the documentation tells you otherwise, you can assume that you are talking about pixels relative to the top-left corner of the client area.

You learn more on this subject later in this chapter, after scrolling is examined, when we discuss the three different coordinate systems in use — world, page, and device coordinates.

Size and SizeF

As with `Point` and `PointF`, sizes come in two varieties. The `Size` struct is for `int` types. `SizeF` is available if you need to use `float` types. Otherwise, `Size` and `SizeF` are identical. This section focuses on the `Size` struct.

In many ways, the `Size` struct is identical to the `Point` struct. It has two integer properties that represent a distance horizontally and vertically. The main difference is that instead of `x` and `y`, these properties are named `Width` and `Height`. You can represent the earlier diagram using this code:

```

Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width, ab.Height);

```

Although `Size` mathematically represents exactly the same thing as `Point`, conceptually, it is intended to be used in a slightly different way. `Point` is used when you are talking about where something is, and `Size` is used when you are talking about how big it is. However, because `Size` and `Point` are so closely related, there are even supported conversions between these two:

```

Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;

```

As an example, think about the rectangle you drew earlier, with top-left coordinate (0,0) and size (50,50). The size of this rectangle is (50,50) and might be represented by a `Size` instance. The bottom-right corner is also at (50,50), but would be represented by a `Point` instance. To see the difference, suppose that you draw the rectangle in a different location, so that its top-left coordinate is at (10,10):

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

Now the bottom-right corner is at coordinate (60,60), but the size is unchanged at (50,50).

The addition operator has been overloaded for `Point` and `Size` structs so that it is possible to add a `Size` to a `Point` struct, resulting in another `Point` struct:



```

static void Main(string[] args)
{
    Point topLeft = new Point(10,10);
    Size rectangleSize = new Size(50,50);
    Point bottomRight = topLeft + rectangleSize;
    Console.WriteLine("topLeft = " + topLeft);
}

```

```

        Console.WriteLine("bottomRight = " + bottomRight);
        Console.WriteLine("Size = " + rectangleSize);
    }

```

[code download PointsAndSizes.sln](#)

This code, running as a simple console application called `PointsAndSizes`, produces the output shown in Figure 48-4.

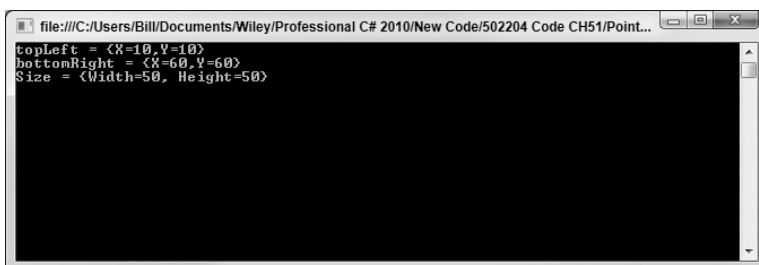


FIGURE 48-4

Note that this output also shows how the `ToString()` method has been overridden in both `Point` and `Size` to display the value in `{X,Y}` format.

It is also possible to subtract a `Size` from a `Point` struct to produce a `Point` struct, and you can add two `Size` structs together, producing another `Size`. It is not possible, however, to add a `Point` struct to another `Point`. Microsoft decided that adding `Point` structs does not conceptually make sense, and so it chose not to supply any overload to the `+` operator that would have allowed that.

You can also explicitly cast a `Point` to a `Size` struct and vice versa:

```

Point topLeft = new Point(10,10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;

```

With this cast, `s1.Width` is assigned the value `topLeft.X`, and `s1.Height` is assigned the value `topLeft.Y`. Hence, `s1` contains (10,10). `p1` will end up storing the same values as `topLeft`.

Rectangle and RectangleF

These structures represent a rectangular region (usually of the screen). Just as with `Point` and `Size`, only the `Rectangle` struct is considered here. `RectangleF` is basically identical except that the properties that represent dimensions all use `float`, whereas those of `Rectangle` use `int`.

A `Rectangle` struct can be thought of as composed of a point, representing the top-left corner of the rectangle, and a `Size` struct, representing how large it is. One of its constructors actually takes a `Point` struct and a `Size` struct as its parameters. You can see this by rewriting the earlier code from the `DrawShapes` sample that draws a rectangle:

```

Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);

```

This code also uses an alternative override of `Graphics.DrawRectangle()`, which takes a `Pen` and a `Rectangle` struct as its parameters.

You can also construct a `Rectangle` struct by supplying the top-left horizontal coordinate, top-left vertical coordinate, width and height separately, and in that order, as individual numbers:

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50)
```

`Rectangle` makes quite a few read-write properties available to set or extract its dimensions in different combinations. See the following table for details.

PROPERTY	DESCRIPTION
<code>int Left</code>	x-coordinate of left-hand edge
<code>int Right</code>	x-coordinate of right-hand edge
<code>int Top</code>	y-coordinate of top
<code>int Bottom</code>	y-coordinate of bottom
<code>int X</code>	Same as <code>Left</code>
<code>int Y</code>	Same as <code>Top</code>
<code>int Width</code>	Width of rectangle
<code>int Height</code>	Height of rectangle
<code>Point Location</code>	Top-left corner
<code>Size Size</code>	Size of rectangle

Note that these properties are not all independent. For example, setting `Width` also affects the value of `Right`.

Region

`Region` represents an area of the screen that has some complex shape. For example, the shaded area in Figure 48-5 could be represented by `Region`.

As you can imagine, the process of initializing a `Region` instance is itself quite complex. Broadly speaking, you can do it by indicating either what component simple shapes make up the region or what path you take as you trace around the edge of the region. If you need to start working with areas similar to this, it is worth looking up the `Region` class in the SDK documentation.



FIGURE 48-5

A NOTE ABOUT DEBUGGING

You are just about ready to do some more advanced types of drawing now. First, however, we just want to say a few things about debugging. If you have tried setting break points in the examples of this chapter, then you have noticed that debugging drawing routines is not quite as simple as debugging other parts of your program because entering and leaving the debugger often causes `Paint` messages to be sent to your application. As a result, setting a break point in your `OnPaint()` override can simply cause your application to keep painting itself over and over again, so it is basically unable to do anything else.

A typical scenario is as follows: You want to find out why your application is displaying something incorrectly, so you set a break point within the `OnPaint()` event. As expected, the application hits your break point and the debugger comes in, at which point your developer environment MDI window comes to the foreground. You more than likely have the developer environments set to full-screen display so that you can more easily view all the debugging information, which means it always completely hides the application you are debugging.

Moving on, you examine the values of some variables and hopefully discover something useful. Then you press F5 to tell the application to continue, so that you can go on to see what happens when the application displays something else after some processing. Unfortunately, the first thing that happens is that the application comes to the foreground, and Windows efficiently detects that the form is visible again and promptly sends it a `Paint` event. This means, of course, that your break point is hit again. If that is what you want, fine. More commonly, what you really want is to hit the break point *later*, when the application is drawing something more interesting, perhaps after you have selected some menu option to read in a file or in some other way changed what is displayed. It looks like you are stuck. Either you do not have a break point in `OnPaint()` at all, or your application can never get beyond the point where it is displaying its initial startup window.

There is a workaround to this problem.

With a big screen, the easiest way is simply to keep your developer environment window tiled rather than maximized. In addition, you want to keep it well away from your application window, so that your application is never hidden in the first place. Unfortunately, in most cases that is not a practical solution because your developer environment window would be too small (you can also get a second monitor). An alternative that uses the same principle is to have your application declare itself as the topmost application while you are debugging. You do this by setting a property in the `Form` class, `TopMost`, which you can easily do in the `InitializeComponent()` method:

```
private void InitializeComponent()
{
    this.TopMost = true;
```

You can also set this property through the properties window in Visual Studio 2010.

Being a `TopMost` window means your application can never be hidden by other windows (except other topmost windows). It always remains above other windows even when another application has the focus. This is how the Task Manager behaves.

Even with this technique, you have to be careful because you can never be certain when Windows might decide for some reason to raise a `Paint` event. If you really want to trap some problem that occurs in `OnPaint()` in some specific circumstance (for example, the application draws something after you select a certain menu option, and something goes wrong at that point), then the best way to do this is to place some dummy code in `OnPaint()` that tests a condition, which will only be true in the specified circumstances. Then place the break point inside the `if` block, as shown here:

```
protected override void OnPaint( PaintEventArgs e )
{
    // Condition() evaluates to true when we want to break
    if (Condition())
    {
        int ii = 0;    // <-SET BREAKPOINT HERE!!!
    }
}
```

This is a quick-and-easy way of setting a conditional break point.

DRAWING SCROLLABLE WINDOWS

The earlier `DrawShapes` example worked very well because everything you needed to draw fit into the initial window size. This section covers what you need to do if that is not the case.

For this example, you expand the `DrawShapes` sample to demonstrate scrolling. To make things a bit more realistic, you start by creating an example, `BigShapes`, in which you make the rectangle and ellipse a bit bigger. Also, while you are at it, you will see how to use the `Point`, `Size`, and `Rectangle` structs by using

them to assist in defining the drawing areas. With these changes, the relevant part of the `Form1` class looks like this:

```
// member fields
private readonly Point rectangleTopLeft = new Point(0, 0);
private readonly Size rectangleSize = new Size(200,200);
private readonly Point ellipseTopLeft = new Point(50, 200);
private readonly Size ellipseSize = new Size(200, 150);
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
    {
        Rectangle rectangleArea =
            new Rectangle (rectangleTopLeft, rectangleSize);
        Rectangle ellipseArea =
            new Rectangle (ellipseTopLeft, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

Note that you have also turned the `Pen`, `Size`, and `Point` objects into member fields. This is more efficient than creating a new `Pen` every time you need to draw anything, as you have been doing so far.

The result of running this example looks like Figure 48-6.

You can see a problem instantly. The shapes do not fit in your 300×300 pixel drawing area.

Normally, if a document is too large to display, an application will add scrollbars to let you scroll the window and look at a chosen part of it. This is another area in which if you were building Windows Forms using standard controls, you would simply allow the .NET runtime and the base classes to handle everything for you. If your form has various controls attached to it, then the `Form` instance will normally know where these controls are, and it will therefore know if its window becomes so small that scrollbars are necessary. The `Form` instance automatically adds the scrollbars for you. It is also able to draw correctly the portion of the screen you have scrolled to. In that case, there is nothing you need to do in your code. In this chapter, however, you are taking responsibility for drawing to the screen. Therefore, you need to help the `Form` instance out when it comes to scrolling.

Adding the scrollbars is actually very easy. The `Form` can still handle all that for you because the `Form` does not know how big an area you want to draw in. (The reason it did not do so in the earlier `BigShapes` example is that Windows does not know they are needed.) You need to determine whether the size of a rectangle that stretches from the top-left corner of the document (or equivalently, the top-left corner of the client area before you have done any scrolling) is big enough to contain the entire document. In this chapter, this area is called the *document area*. As shown in Figure 48-7, the document area for this example is (250×350) pixels.

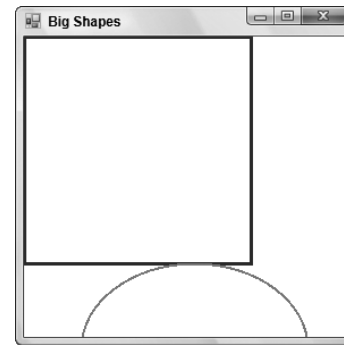


FIGURE 48-6

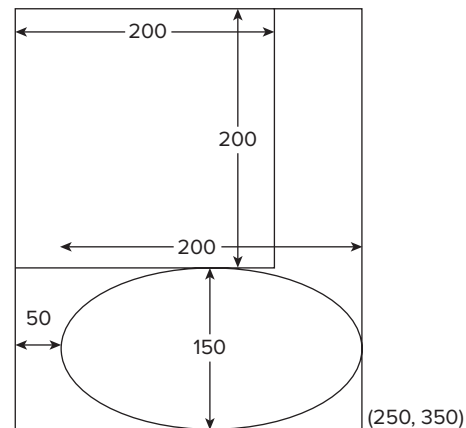


FIGURE 48-7

It is easy to tell the form how big the document is. You use the relevant property, `Form.AutoScrollMinSize`. Therefore, you can add this code to either the `InitializeComponent()` method or the `Form1` constructor:

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
    this.AutoScrollMinSize = new Size(250, 350);
}
```

Alternatively, the `AutoScrollMinSize` property can be set using the Visual Studio 2010 properties window. Note that to gain access to the `Size` class, you need to add the following using statement:

```
using System.Drawing;
```

Setting the minimum size at application startup and leaving it thereafter is fine in this particular example because you know that is how big the screen area will always be. Your document never changes size while this particular application is running. Keep in mind, however, that if your application does things like display contents of files or something else for which the area of the screen might change, you will need to set this property at other times (and in that case you will have to sort out the code manually — the Visual Studio 2010 properties window can help you only with the initial value that a property has when the form is constructed).

Setting `AutoScrollMinSize` is a start, but it is not yet quite enough. Figure 48-8 shows what the example application looks like now — initially you get the screen that correctly displays the shapes.

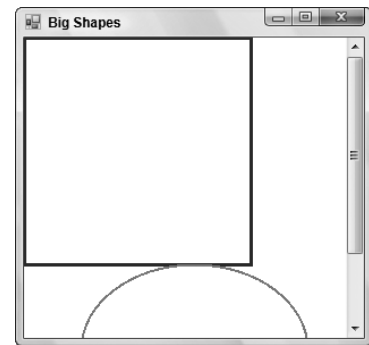


FIGURE 48-8

Notice that not only has the form correctly set the scrollbars, but also it has correctly sized them to indicate what proportion of the document is currently displayed. You can try resizing the window while the sample is running — you will find the scrollbars respond properly, and even disappear if you make the window big enough so that they are no longer needed.

However, look at what happens when you actually use one of the scrollbars to scroll down a bit (see Figure 48-9). Clearly, something has gone wrong!

What's wrong is that you haven't taken into account the position of the scrollbars in the code in your `OnPaint()` override. You can see this very clearly if you force the window to repaint itself completely by minimizing and restoring it (see Figure 48-10).

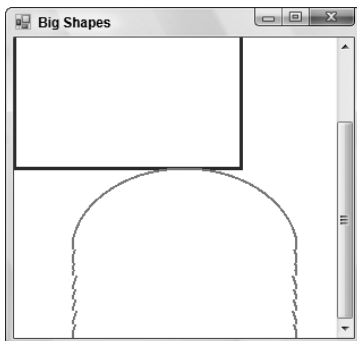


FIGURE 48-9

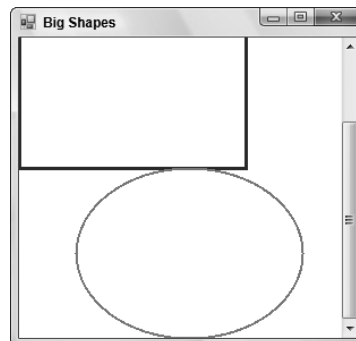


FIGURE 48-10

The shapes have been painted, just as before, with the top-left corner of the rectangle nestled into the top-left corner of the client area — as if you hadn't moved the scrollbars at all.

Before you see how to correct this problem, take a closer look at precisely what is happening in these screenshots.

Start with the `BigShapes` sample, shown in Figure 48-8. In this example, the entire window has just been repainted. Reviewing your code, you learn that it instructs the graphics instance to draw a rectangle with top-left coordinates (0,0) — relative to the top-left corner of the client area of the window — which is what has been drawn. The problem is that the graphics instance by default interprets coordinates as relative to the client window and is unaware of the scrollbars. Your code, as yet, does not attempt to adjust the coordinates for the scrollbar positions. The same goes for the ellipse.

Now, you can tackle the screenshot in Figure 48-9. After you scroll down, you notice that the top half of the window looks fine because it was drawn when the application first started up. When you scroll windows, Windows does not ask the application to redraw what was already on the screen. Windows is smart enough to determine which currently displayed bits can be smoothly moved around to match where the scrollbars are now located. This is a much more efficient process because it may be able to use some hardware acceleration to do that, too. The bit in this screenshot that is wrong is the bottom third of the window. This part of the window was not drawn when the application first appeared because before you started scrolling, it was outside the client area. This means that Windows asks your `BigShapes` application to draw this area. It will raise a `Paint` event passing in just this area as the clipping rectangle. And that is exactly what your `OnPaint()` override has done.

One way to look at the problem is that you are, at the moment, expressing your coordinates relative to the top-left corner of the start of the document — you need to convert them to express them relative to the top-left corner of the client area instead (see Figure 48-11).

To make the diagram clearer, the document is actually extended further downward and to the right, beyond the boundaries of the screen, but this does not change our reasoning. It also assumes a small horizontal scroll as well as a vertical one.

In Figure 48-11, the thin rectangles mark the borders of the screen area and of the entire document. The thick lines mark the rectangle and ellipse that you are trying to draw. P marks some arbitrary point that you are drawing and that is being used as an example. When calling the drawing methods, the graphics instance was supplied with the vector from point B to (say) point P, expressed as a `Point` instance. You actually need to give it the vector from point A to point P.

The problem is that you do not know what the vector from A to P is. You know what B to P is; that is just the coordinates of P relative to the top-left corner of the document — the position where you want to draw point P in the document. You also know that the vector from B to A is just the amount you have scrolled by. This is stored in a property of the `Form` class called `AutoScrollPosition`. However, you do not know the vector from A to P.

To solve this problem, you subtract one vector from the other. Say, for example, to get from B to P you move 150 pixels across and 200 pixels down, whereas to get from B to A you move 10 pixels across and 57 pixels down. That means to get from A to P you have to move 140 (150 minus 10) pixels across and 143 (200 minus 57) pixels down. To make it even simpler, the `Graphics` class actually implements a method that will do these calculations for you. It is called `TranslateTransform()`. You pass it the horizontal and vertical coordinates that say where the top left of the client area is relative to the top-left corner of the document (your `AutoScrollPosition` property, that is, the vector from B to A in the diagram). The `Graphics` device will now work out all its coordinates, taking into account where the client area is relative to the document.

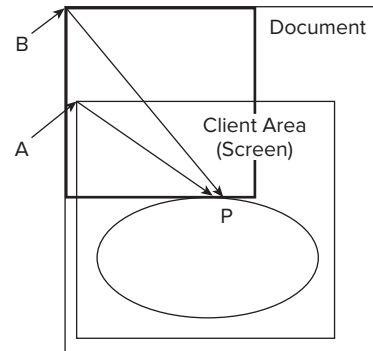


FIGURE 48-11

If we translate this long explanation into code, all you typically need to do is add the following line to your drawing code:

```
dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

However, in this example, it is a little more complicated because you are also separately testing whether you need to do any drawing by looking at the clipping region. You need to adjust this test to take the scroll position into account, too. When you have done that, the full drawing code for the sample looks like this:

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Size scrollOffset = new Size(this.AutoScrollPosition);
    if (e.ClipRectangle.Top+scrollOffset.Width < 350 ||
        e.ClipRectangle.Left+scrollOffset.Height < 250)
    {
        Rectangle rectangleArea = new Rectangle
            (rectangleTopLeft+scrollOffset, rectangleSize);
        Rectangle ellipseArea = new Rectangle
            (ellipseTopLeft+scrollOffset, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

Now you have your scroll code working perfectly. You can at last obtain a correctly scrolled screenshot (see Figure 48-12).

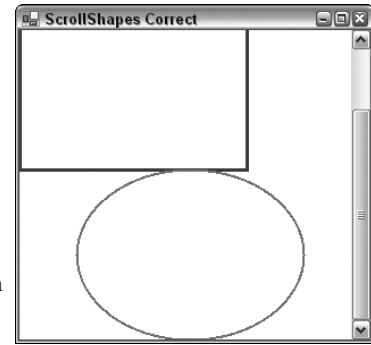


FIGURE 48-12

WORLD, PAGE, AND DEVICE COORDINATES

The distinction between measuring position relative to the top-left corner of the document and measuring it relative to the top-left corner of the screen (desktop) is so important that GDI+ has special names for these coordinate systems:

- **World coordinates** specify the position of a point measured in pixels from the top-left corner of the document.
- **Page coordinates** specify the position of a point measured in pixels from the top-left corner of the client area.



Developers familiar with GDI will note that world coordinates correspond to what in GDI were known as logical coordinates. Page coordinates correspond to what were known as device coordinates. As a developer familiar with GDI, you should also note that the way you code conversion between logical and device coordinates has changed in GDI+. In GDI, conversions took place via the device context, using the `LPtoDP()` and `DPtoLP()` Windows API functions. In GDI+, it is the `Control` class, from which both `Form` and all the various Windows Forms controls derive, that maintains the information needed to carry out the conversion.

GDI+ also distinguishes a third coordinate system, which is now known as *device coordinates*. Device coordinates are similar to page coordinates, except that you do not use pixels as the unit of measurement. Instead, you use some other unit that can be specified by the user by calling the `Graphics.PageUnit` property. Possible units, besides the default of pixels, include inches and millimeters. Although you will not use the `PageUnit` property in this chapter, you might find it useful as a way of getting around the different pixel densities of devices. For example, 100 pixels on most monitors will occupy approximately an inch. However, laser printers can have 1,200 or more dpi (dots per inch), which means that a shape specified

as 100 pixels wide will look a lot smaller when printed. By setting the units to, say, inches and specifying that the shape should be 1 inch wide, you can ensure that the shape will look the same size on the different devices. This is illustrated in the following:

```
Graphics dc = this.CreateGraphics();
dc.PageUnit = GraphicsUnit.Inch;
```

Possible units available via the `GraphicsUnit` enumeration include the following:

- **Display** — Defines the display's unit measure.
- **Document** — Defines the document unit ($\frac{1}{300}$ inch) as the unit of measure.
- **Inch** — Defines the inch measurement as the unit of measure.
- **Millimeter** — Defines the millimeter measurement as the unit of measure.
- **Pixel** — Defines the pixel measurement as the unit of measure.
- **Point** — Defines the printer point ($\frac{1}{72}$ inch) as the unit of measure.
- **World** — Defines the world coordinate system as the unit of measure.

COLORS

This section discusses the ways that you can specify what color you want something to be drawn in.

Colors in GDI+ are represented by instances of the `System.Drawing.Color` struct. Generally, after you have instantiated this struct, you won't do much with the corresponding `Color` instance — you just pass it to whatever other method you are calling that requires a `Color`. You have encountered this struct before, when you set the background color of the client area of the window in each of the examples, as well as when you set the colors of the various shapes you were displaying. The `Form.BackColor` property actually returns a `Color` instance. This section looks at this struct in more detail. In particular, it examines several different ways that you can construct a `Color`.

Red-Green-Blue Values

The total number of colors that can be displayed by a monitor is huge — more than 16 million. To be exact, the number is 2 to the power of 24, which works out to 16,777,216. Obviously, you need some way of indexing those colors so that you can indicate which one is the color you want to display at any given pixel.

The most common way of indexing colors is by dividing them into the red, green, and blue components. This idea is based on the theory that any color that the human eye can distinguish can be constructed from a certain amount of red light, a certain amount of green light, and a certain amount of blue light. These colors are known as *components*. In practice, dividing the amount of each component light into 256 possible intensities yields a sufficiently fine gradation to be able to display images that are perceived by the human eye to be of photographic quality. You, therefore, specify colors by giving the amounts of these components on a scale of 0 to 255 where 0 means that the component is not present and 255 means that it is at its maximum intensity.

This gives you your first way of telling GDI+ about a color. You can indicate a color's red, green, and blue values by calling the static function `Color.FromArgb()`. Microsoft has chosen not to supply a constructor to do this task. The reason is that there are other ways, besides the usual RGB components, to indicate a color. Because of this, Microsoft felt that the meaning of parameters passed to any constructor they defined would be open to misinterpretation:

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor = Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

The three parameters are, respectively, the quantities of red, green, and blue. This function has a number of other overloads, some of which also allow you to specify something called an alpha-blend (that is the *A* in the name of the method, `FromArgb()`). Alpha blending is beyond the scope of this chapter, but it allows you to paint a color semi-transparently by combining it with whatever color was already on the screen. This can give some beautiful effects and is often used in games.

The Named Colors

Constructing a `Color` using `FromArgb()` is the most flexible technique because it literally means you can specify any color that the human eye can see. However, if you want a simple, standard, well-known color such as red or blue, it is a lot easier to name the color you want. Hence, Microsoft has also provided a large number of static properties in `Color`, each of which returns a named color. It was one of these properties that you used when you set the background color of your windows to white in the examples:

```
this.BackColor = Color.White;
// has the same effect as:
// this.BackColor = Color.FromArgb(255, 255, 255);
```

Several hundred such colors exist. The full list is given in the SDK documentation. They include all the simple colors: Red, White, Blue, Green, Black, and so on, as well as such delights as `MediumAquaMarine`, `LightCoral`, and `DarkOrchid`. There is also a `KnownColor` enumeration, which lists the named colors.



Each of these named colors represents a precise set of RGB values. They were originally chosen many years ago for use on the Internet. The idea was to provide a useful set of colors across the spectrum whose names would be recognized by web browsers, thus saving you from having to write explicit RGB values in your HTML code. A few years ago, these colors were also important because early browsers could not necessarily display very many colors accurately, and the named colors were supposed to provide a set of colors that would be displayed correctly by most browsers. These days, that aspect is less important because modern web browsers are quite capable of displaying any RGB value correctly. Web-safe color palettes are also available that provide developers with a comprehensive list of colors that work with most browsers.

Graphics Display Modes and the Safety Palette

Although in principle monitors can display any of the more than 16 million RGB colors, in practice this depends on how you have set the display properties on your computer. In Windows, there are traditionally three main color options (although some machines might provide other options depending on the hardware): true color (24 bit), high color (16 bit), and 256 colors. (On some graphics cards these days, true color is actually marked as 32 bit. This has to do with optimizing the hardware, though in that case only 24 bits of the 32 bits are used for the color itself.)

Only true color mode allows you to display all the RGB colors simultaneously. This sounds like the best option, but it comes at a cost: 3 bytes are needed to hold a full RGB value, which means that 3 bytes of graphics card memory are needed to hold each pixel that is displayed. If graphics card memory is at a premium (a restriction that is less common now than it used to be), then you might want to choose one of the other modes. High color mode gives you 2 bytes per pixel, which is enough to give 5 bits for each RGB component. Therefore, instead of 256 gradations of red intensity, you get just 32 gradations. The same applies to blue and green, which produce a total of 65,536 colors. That is just about enough to give apparent photographic quality on a casual inspection, although areas of subtle shading tend to be broken up a bit.

The 256-color mode gives you even fewer colors. However, in this mode, you get to choose the colors. The system sets up something known as a *palette*. This is a list of 256 colors chosen from the 16 million RGB

colors. After you have specified the colors in the palette, the graphics device will be able to display just those colors. The palette can be changed at any time, but the graphics device can only display 256 different colors on the screen at any one time. The 256-color mode is used only when high performance is necessary and video memory is at a premium. Most computer games use this mode. They can still achieve decent-looking graphics because of a very careful choice of palette.

In general, if a display device is in high-color or 256-color mode and a particular RGB color is requested, then it will pick the nearest mathematical match from the pool of colors that it is able to display. It is for this reason that it is important to be aware of the color modes. If you are drawing something that involves subtle shading or photographic-quality images, and the user does not have 24-bit color mode selected, she might not see the image the same way you intended it. Therefore, if you are doing that kind of work with GDI+, then you should test your application in different color modes. (It is also possible for your application to programmatically set a given color mode, although that is not discussed in this chapter for lack of space.)

The Safety Palette

For reference, this section quickly mentions the safety palette, which is a commonly used default palette. To use the safety palette, you set six equally spaced possible values for each color component: 0, 51, 102, 153, 204, and 255. In other words, the red component can have any of these values. The green component can have any of these values and so can the blue component. Possible colors from the safety palette include (0, 0, 0), black; (153, 0, 0), a fairly dark shade of red; (0, 255, 102), green with a smattering of blue added; and so on. This gives you a total of $6^3 = 216$ colors. The idea is that this provides an easy way of creating a palette that contains colors from right across the spectrum and of all degrees of brightness. In practice, however, this does not actually work that well because equal mathematical spacing of color components does not mean equal perception of color differences by the human eye.

If you set Windows to 256-color mode, you will find that the default palette is the safety palette, with 20 Windows-standard colors added to it, and 20 spare colors.

PENS AND BRUSHES

This section reviews two helper classes that are needed to draw shapes. You have already encountered the `Pen` class, which you used to instruct the graphics instance how to draw lines. A related class is `System.Drawing.Brush`, which instructs the graphics instance how to fill regions. For example, the `Pen` is needed to draw the outlines of the rectangle and ellipse in the previous examples. If you had needed to draw these shapes as solid, you would have used a brush to specify how to fill them. One aspect of both of these classes is that you will hardly ever call any methods on them. You simply construct a `Pen` or `Brush` instance with the required color and other properties, and then pass it to drawing methods that require a `Pen` or `Brush`.



If you have programmed using GDI before, you may have noticed from the first few examples that pens are used in a different way in GDI+. In GDI, the normal practice was to call a Windows API function, `SelectObject()`, which actually associated a pen with the device context. That pen was then used in all drawing operations that required a pen until you informed the device context otherwise, by calling `SelectObject()` again. The same principle held for brushes and other objects such as fonts or bitmaps. With GDI+, Microsoft has opted for a stateless model in which there is no default pen or other helper object. Rather, you simply specify with each method call the appropriate helper object to be used for that particular method.

Brushes

GDI+ has several different kinds of brushes — more than there is space to go into in this chapter, so this section just explains the simpler ones to give you an idea of the principles. Each type of brush is represented by an instance of a class derived from the abstract class `System.Drawing.Brush`. The simplest brush, `System.Drawing.SolidBrush`, indicates that a region is to be filled with a solid color:

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
Brush solidFunnyOrangyBrownBrush = new SolidBrush(Color.FromArgb(255,155,100));
```

Alternatively, if the brush is one of the web-safe colors, then you can construct the brush using another class, `System.Drawing.Brushes.Brushes`. `Brushes` is one of those classes that you never actually instantiate (it has a private constructor to stop you from doing that). It simply has a large number of static properties, each of which returns a brush of a specified color. You can use `Brushes` like this:

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

The next level of complexity is a hatch brush, which fills a region by drawing a pattern. This type of brush is considered more advanced, so it is in the `Drawing2D` namespace, represented by the class `System.Drawing.Drawing2D.HatchBrush`. The `Brushes` class cannot help you with hatch brushes; you will need to construct one explicitly by supplying the hatch style and two colors — the foreground color followed by the background color. (Note, you can omit the background color, in which case it defaults to black). The hatch style comes from an enumeration, `System.Drawing.Drawing2D.HatchStyle`. You can choose from a large number of `HatchStyle` values (see the SDK documentation for the full list). To give you an idea, typical styles include `ForwardDiagonal`, `Cross`, `DiagonalCross`, `SmallConfetti`, and `ZigZag`. Examples of constructing a hatch brush include these:

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross, Color.Azure);
// background color of CrossBrush is black
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
```

Solid and hatch brushes are the only brushes available with GDI. GDI+ has added a couple of new styles of brushes:

- `System.Drawing.Drawing2D.LinearGradientBrush` fills in an area with a color that varies across the screen.
- `System.Drawing.Drawing2D.PathGradientBrush` is similar, but in this case, the color varies along a path around the region to be filled.

Note that both brushes can render some spectacular effects if used carefully.

Pens

Unlike brushes, pens are represented by just one class: `System.Drawing.Pen`. However, the pen is slightly more complex than the brush because it needs to indicate how thick lines should be (how many pixels wide) and, for a wide line, how to fill the area inside the line. Pens can also specify a number of other properties, which are beyond the scope of this chapter, but which include the `Alignment` property mentioned earlier. This property indicates where in relation to the border of a shape a line should be drawn, as well as what shape to draw at the end of a line (whether to round off the shape).

The area inside a thick line can be filled with a solid color or by using a brush. Hence, a `Pen` instance might contain a reference to a `Brush` instance. This is quite powerful because it means that you can draw lines that are colored in by using, say, hatching or linear shading. There are four different ways to construct a `Pen` instance that you have designed yourself. One is by passing a color; a second is by passing in a brush. Both of these constructors will produce a pen with a width of one pixel. Alternatively, a third way is to pass in a

color or a brush, and additionally a `float`, which represents the width of the pen. (It needs to be a `float` in case you are using non-default units such as millimeters or inches for the `Graphics` object that will do the drawing, so you can, for example, specify fractions of an inch.) For example, you can construct pens similar to this:

```
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
Pen solidBluePen = new Pen(Color.FromArgb(0,0,255));
Pen solidWideBluePen = new Pen(Color.Blue, 4);
Pen brickPen = new Pen(brickBrush);
Pen brickWidePen = new Pen(brickBrush, 10);
```

Additionally, a fourth way offers the quick construction of pens by using the class `System.Drawing.Pens`, which, like the `Brushes` class, contains a number of stock pens. These pens all have a 1-pixel width and come in the usual sets of web-safe colors. This allows you to construct pens in this way:

```
Pen solidYellowPen = Pens.Yellow;
```

DRAWING SHAPES AND LINES

You have almost finished the first part of the chapter, and you have seen all the basic classes and objects required to draw specified shapes and so on to the screen. This section starts by reviewing some of the drawing methods the `Graphics` class makes available and presents a short example that illustrates the use of several brushes and pens.

`System.Drawing.Graphics` has a large number of methods that allow you to draw various lines, outline shapes, and solid shapes. Again, there are too many to provide a comprehensive list here, but the following table lists the main ones and should give you some idea of the variety of shapes you can draw.

METHOD	TYPICAL PARAMETERS	WHAT IT DRAWS
<code>DrawLine</code>	Pen, start and end points	A single straight line.
<code>DrawRectangle</code>	Pen, position, and size	Outline of a rectangle.
<code>DrawEllipse</code>	Pen, position, and size	Outline of an ellipse.
<code>FillRectangle</code>	Brush, position, and size	Solid rectangle.
<code>FillEllipse</code>	Brush, position, and size	Solid ellipse.
<code>DrawLines</code>	Pen, array of points	Series of lines, connecting each point to the next one in the array.
<code>DrawBezier</code>	Pen, four points	A smooth curve through the two endpoints, with the remaining two points used to control the shape of the curve.
<code>DrawCurve</code>	Pen, array of points	A smooth curve through the points.
<code>DrawArc</code>	Pen, rectangle, two angles	Portion of circle within the rectangle defined by the angles.
<code>DrawClosedCurve</code>	Pen, array of points	Like <code>DrawCurve</code> but also draws a straight line to close the curve.
<code>DrawPie</code>	Pen, rectangle, two angles	Wedge-shaped outline within the rectangle.
<code>FillPie</code>	Brush, rectangle, two angles	Solid wedge-shaped area within the rectangle.
<code>DrawPolygon</code>	Pen, array of points	Like <code>DrawLines</code> but also connects first and last points to close the figure drawn.

Before we leave the subject of drawing simple objects, this section rounds off with a simple example that demonstrates the kinds of visual effects you can achieve using brushes. The example is called `ScrollMoreShapes`, and it is essentially a revision of `ScrollShapes`. Besides the rectangle and ellipse, you will add a thick line and fill in the shapes with various custom brushes. You have already learned the principles of drawing, so the code speaks for itself. First, because of your new brushes, you need to indicate that you are using the `System.Drawing.Drawing2D` namespace:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

Next are some extra fields in your `Form1` class, which contain details of the locations where the shapes are to be drawn, as well as various pens and brushes you will use:



Available for
download on
Wrox.com

```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
                                                    new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
                                                new Size(200,150));
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
private readonly Brush solidAzureBrush = Brushes.Azure;
private readonly Brush solidYellowBrush = new SolidBrush(Color.Yellow);
private static readonly Brush brickBrush = new
    HatchBrush(HatchStyle.DiagonalBrick, Color.DarkGoldenrod, Color.Cyan);
private readonly Pen brickWidePen = new Pen(brickBrush, 10);
```

[*code download ScrollMoreShapes.sln*](#)

The `brickBrush` field has been declared as static so that you can use its value to initialize the `brickWidePen` field. C# will not let you use one instance field to initialize another instance field because it has not defined which one will be initialized first. However, declaring the field as static solves the problem. Because only one instance of the `Form1` class will be instantiated, it is immaterial whether the fields are static or instance fields.

The following is the `OnPaint()` override:



Available for
download on
Wrox.com

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Point scrollOffset = AutoScrollPosition;
    dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);
    if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
        e.ClipRectangle.Left+scrollOffset.Y < 250)
    {
        dc.DrawRectangle(bluePen, rectangleBounds);
        dc.FillRectangle(solidYellowBrush, rectangleBounds);
        dc.DrawEllipse(redPen, ellipseBounds);
        dc.FillEllipse(solidAzureBrush, ellipseBounds);
        dc.DrawLine(brickWidePen, rectangleBounds.Location,
                    ellipseBounds.Location+ellipseBounds.Size);
    }
}
```

[*code download ScrollMoreShapes.sln*](#)

As before, you also set the `AutoScrollMinSize` to (250,350). Figure 48-13 shows the new results.

Notice that the thick diagonal line has been drawn on top of the rectangle and ellipse because it was the last item to be painted.

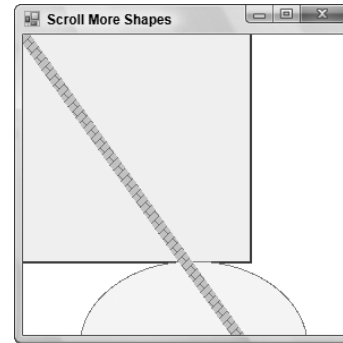


FIGURE 48-13

DISPLAYING IMAGES

One of the most common things you might want to do with GDI+ is display an image that already exists in a file. This is actually a lot simpler than drawing your own user interface because the image is already pre-drawn. Effectively, all you have to do is load the file and instruct GDI+ to display it. The image can be a simple line drawing, an icon, or a complex image such as a photograph. You can also manipulate the image by stretching or rotating it, or simply displaying only a portion of it.

This section, just for a change, presents the sample first. Then it discusses some of the issues you need to be aware of when displaying images. Presenting it this way is possible because the code needed to display an image is so simple.

The class you need is the .NET base class, `System.Drawing.Image`. An instance of `Image` represents one image. Reading in an image simply takes one line of code:

```
Image myImage = Image.FromFile("FileName");
```

`FromFile()` is a static member of `Image` and is the usual way of instantiating an image. The file can be any of the commonly supported graphics file formats, including `.bmp`, `.jpg`, `.gif`, and `.png`.

Displaying an image is also very simple, assuming that you have a suitable `Graphics` instance at hand — a call to either `Graphics.DrawImageUnscaled()` or `Graphics.DrawImage()` suffices. There are quite a few overloads of these methods, allowing you a lot of flexibility in the information you supply in terms of where the image is located and how big it is to be drawn. However, this example uses `DrawImage()`, like this:

```
dc.DrawImage(myImage, points);
```

In this line of code, `dc` is assumed to be a `Graphics` instance, and `myImage` is the `Image` to be displayed. `points` is an array of `Point` structs, where `points[0]`, `points[1]`, and `points[2]` are the coordinates of the top-left, top-right, and bottom-left corner of the image.



Images are probably the area in which developers familiar with GDI will notice the biggest difference between GDI and GDI+. In GDI, displaying an image involved several nontrivial steps. If the image was a bitmap, then loading it was reasonably simple. Nevertheless, if it were any other file type, then loading it would involve a sequence of calls to OLE objects. Actually, getting a loaded image onto the screen required getting a handle to it, selecting it into a memory device context, and then performing a block transfer between device contexts. Although the device contexts and handles are still there behind the scenes and will be needed if you want to start doing sophisticated editing of the images from your code, simple tasks have now been extremely well wrapped up in the GDI+ object model.

The process of displaying an image is illustrated with an example called `DisplayImage`. The example simply displays a `.jpg` file in the application's main window. To keep things simple, the path of the `.jpg` file is hard-coded into the application (so if you run the example, then you need to change it to reflect the location of the file in your system). The `.jpg` file you will display is a sunset picture in St. Petersburg.

As with the other examples, the `DisplayImage` project is a standard C# Visual Studio 2010-generated Windows application. You add the following fields to your `Form1` class:

```
readonly Image piccy;
private readonly Point [] piccyBounds;
```

You then load the file in the `Form1()` constructor:



```
public Form1()
{
    InitializeComponent();
    piccy =
        Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.jpg");
    AutoScrollMinSize = piccy.Size;
    piccyBounds = new Point[3];
    piccyBounds[0] = new Point(0,0); // top left
    piccyBounds[1] = new Point(piccy.Width,0); // top right
    piccyBounds[2] = new Point(0,piccy.Height); // bottom left
}
```

[*code download DisplayPicture.sln*](#)

Note that the size in pixels of the image is obtained as its `Size` property, which you use to set the document area. You also set up the `piccyBounds` array, which is used to identify the position of the image on the screen. You have chosen the coordinates of the three corners to draw the image in its actual size and shape here, but if you had wanted the image to be resized, stretched, or even sheared into a nonrectangular parallelogram, then you could do so simply by changing the values of the `Points` in the `piccyBounds` array.

The image is displayed in the `OnPaint()` override:



```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.ScaleTransform(1.0f, 1.0f);
    dc.TranslateTransform(AutoScrollPosition.X, AutoScrollPosition.Y);
    dc.DrawImage(piccy, piccyBounds);
}
```

[*code download DisplayPicture.sln*](#)

Finally, note the modification made to the IDE-generated `Form1.Dispose()` method:



```
protected override void Dispose(bool disposing)
{
    piccy.Dispose();
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
```

[*code download DisplayPicture.sln*](#)

Disposing of the image as soon as possible when it is no longer needed is important because images generally take up a lot of memory while in use. After `Image.Dispose()` has been called, the `Image` instance no longer refers to any actual image, and so it can no longer be displayed (unless you load a new image).

Figure 48-14 shows the result of running this code.



FIGURE 48-14

ISSUES WHEN MANIPULATING IMAGES

Although displaying images is very simple, it still pays to have some understanding of what is going on behind the scenes.

The most important point to understand about images is that they are always rectangular. That is not just a convenience; it is because of the underlying technology. All modern graphics cards have hardware built in that can efficiently copy blocks of pixels from one area of memory to another area of memory, provided that the block of pixels represents a rectangular region. This hardware-accelerated operation can occur virtually as one single operation, and as such, is extremely fast. Indeed, it is the key to modern high-performance graphics. This operation is known as a *bitmap block transfer* (or *BitBlt*). `Graphics.DrawImageUnscaled()` internally uses a `BitBlt`, which is why you can see a huge image, perhaps containing as many as a million pixels, appearing almost instantly. If the computer had to copy the image to the screen pixel by pixel, you would see the image gradually being drawn over a period of up to several seconds.

`BitBlts` are very efficient; therefore, almost all drawing and manipulation of images is carried out using them. Even some editing of images will be done by manipulating portions of images with `BitBlts` between DCs that represent areas of memory. In the days of GDI, the Windows 32 API function `BitBlt()` was arguably the most important and widely used function for image manipulation, although with GDI+, the `BitBlt` operations are largely hidden by the GDI+ object model.

It's not possible to `BitBlt` areas of images that are not rectangular, although similar effects can be easily simulated. One way is to mark a certain color as transparent for the purposes of a `BitBlt`, so that areas of that color in the source image will not overwrite the existing color of the corresponding pixel in the destination device. It is also possible to specify that in the process of a `BitBlt`, each pixel of the resultant image will be formed by some logical operation (such as a bitwise AND) on the colors of that pixel in the source image and in the destination device before the `BitBlt`. Such operations are supported by hardware acceleration and can be used to give a variety of subtle effects. Note that the `Graphics` object implements another method, `DrawImage()`. This is similar to `DrawImageUnscaled()` but comes in a large number of overloads that allow you to specify more complex forms of `BitBlt` be used in the drawing process. `DrawImage()` also allows you to draw (using `BitBlt`) only a specified part of the image, or to perform certain other operations on it such as scaling it (expanding or reducing it in size) as it is drawn.

DRAWING TEXT

We have chosen to cover the very important topic of displaying text late in this chapter because drawing text to the screen is (in general) more complex than drawing simple graphics. Although displaying a line or two of text when you don't care about the appearance is extremely easy (it takes one single call to the `Graphics.DrawString()` method), if you are trying to display a document that has a fair amount of text in it, then you will rapidly find that things become a lot more complex. This is for two reasons:

- If you are concerned about getting the appearance just right, then you must understand fonts. Whereas shape drawing requires brushes and pens as helper objects, the process of drawing text requires fonts as helper objects. Moreover, understanding fonts is not a trivial undertaking.
- Text needs to be very carefully laid out in the window. Users generally expect words to follow naturally from one word to another and to be lined up with clear spaces in between. Doing that is harder than you might think. For starters, you do not usually know in advance how much space on the screen a word is going to take up. That has to be calculated (using the `Graphics.MeasureString()` method). In addition, the space a word occupies on the screen affects where in the document every subsequent word is placed. If your application does any line wrapping, then it will need to assess word sizes carefully before deciding where to place the line break. The next time you run Microsoft Word, look carefully at the way Word is continually repositioning text as you do your work; there is a lot of complex processing going on there. Chances are that any GDI+ application you work on will not be nearly as complex as Word. However, if you need to display any text, many of the same considerations apply.

In short, high-quality text processing is tricky to get right. However, putting a line of text on the screen, assuming that you know the font and where you want it to go, is actually very simple. Therefore, the next section presents a quick example that shows you how to display some text, followed by a short review of the principles of fonts and font families and a more realistic (and involved) text-processing example, `CapsEditor`.

SIMPLE TEXT EXAMPLE

This example, `DisplayText`, is your usual Windows Forms effort. This time you override `OnPaint()` and add member fields as follows:



```
private readonly Brush blackBrush = Brushes.Black;
private readonly Brush blueBrush = Brushes.Blue;
private readonly Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private readonly Font boldTimesFont = new Font("Times New Roman", 10,
    FontStyle.Bold);
private readonly Font italicCourierFont = new Font("Courier", 11,
    FontStyle.Italic | FontStyle.Underline);

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.DrawString("This is a groovy string", haettenschweilerFont, blackBrush,
        10, 10);
    dc.DrawString("This is a groovy string " +
        "with some very long text that will never fit in the box",
        boldTimesFont, blueBrush,
        new Rectangle(new Point(10, 40), new Size(100, 40)));
    dc.DrawString("This is a groovy string", italicCourierFont, blackBrush,
        new Point(10, 100));
}
```

[*code download DisplayText.sln*](#)

Figure 48-15 shows the result of running this example.

The example demonstrates the use of the `Graphics.DrawString()` method to draw items of text. The method `DrawString()` comes in a number of overloads, three of which are demonstrated here. The different overloads require parameters that indicate the text to be displayed, the font that the string should be drawn in, and the brush that should be used to construct the various lines and curves that make up each character of text. A few alternatives exist for the remaining parameters. In general, however, it is possible to specify either a `Point` (or equivalently, two numbers) or a `Rectangle`.

If you specify a `Point`, then the text will start with its top-left corner at that `Point` and simply stretch out to the right. If you specify a `Rectangle`, then the `Graphics` instance will lay out the string inside that rectangle. If the text does not fit within the boundaries of the rectangle, it will be cut off (see the fourth line of text in Figure 48-15). Passing a rectangle to `DrawString()` means that the drawing process will take longer because `DrawString()` will need to figure out where to put line breaks, but the result may look nicer — provided the string fits in the rectangle!

This example also shows a few ways to construct fonts. You always need to include the name of the font and its size (height). You can also optionally pass in various styles that modify how the text is to be drawn (bold, underline, and so on).

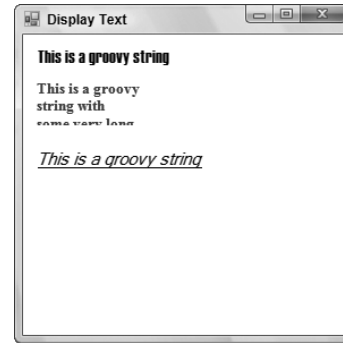


FIGURE 48-15

FONTS AND FONT FAMILIES

A font describes exactly how each letter should be displayed. Selection of the appropriate font and providing a reasonable variety of fonts within a document are important factors in improving readability.

Most people, if asked to name a font, might mention Arial or Times New Roman (if they are Windows users) or Times or Helvetica (if they are Mac OS users). In fact, these are not fonts at all — they are *font families*. The font family tells you, in generic terms, the visual style of the text and is a key factor in the overall appearance of your application. Most of us recognize the styles of the most common font families, even if we are not consciously aware of it.

An actual *font* would be something like Arial 9-point italic. In other words, the size and other modifications to the text are specified as well as the font family. These modifications might include whether text is **bold**, *italic*, underlined, or displayed in SMALL CAPS or as _{subscript}; this is technically referred to as the *style*, although in some ways, the term is misleading because the visual appearance is determined as much by the font family.

The size of the text is measured by specifying its height. The height is measured in *points* — a traditional unit that represents $\frac{1}{72}$ of an inch (0.351 mm). So letters in a 10-point font are roughly $\frac{1}{7}$ of an inch, or 3.5 mm high. However, you will not get seven lines of 10-point text into one inch of vertical screen or paper space, as you need to allow for the spacing between the lines as well.



Strictly speaking, measuring the height is not quite as simple as that because there are several different heights that you must consider. For example, there is the height of tall letters such as the A or F (this is the measurement that we are referring to when we talk about the height), the additional height occupied by any accents on letters such as Å or Ñ (the internal leading), and the extra height below the baseline needed for the tails of letters such as y and g (the descent). However, for this chapter we will not worry about that. When you specify the font family and the main height, these subsidiary heights are determined automatically.

When you are dealing with fonts, you might also encounter some other terms commonly used to describe certain font families:

- **Serif** font families have feet at the ends of many of the lines that make up the characters (these ticks are known as serifs). Times New Roman is a classic example of this.
- **Sans serif** font families, by contrast, do not have these feet. Good examples of sans serif fonts are Arial and Verdana. The lack of feet often gives text a blunt, in-your-face appearance, so sans serif fonts are often used for important text.
- A **TrueType** font family is one that is defined by expressing the shapes of the curves that make up the characters in a precise mathematical manner. This means that the same definition can be used to calculate how to draw fonts of any size within the family. These days, virtually all the fonts you might use are TrueType fonts. Some older font families from the days of Windows 3.1 were defined by individually specifying the bitmap for each character separately for each font size, but the use of these fonts is now discouraged.

Microsoft has provided two main classes that you need to deal with when selecting or manipulating fonts:

- `System.Drawing.Font`
- `System.Drawing.FontFamily`

You have already seen the main use of the `Font` class. When you want to draw text, you instantiate an instance of `Font` and pass it to the `DrawString()` method to indicate how the text should be drawn. A `FontFamily` instance is used to represent a family of fonts.

You can use the `FontFamily` class, for example, if you know you want a font of a particular type (serif, sans serif, or TrueType), but do not have a preference for which font. The static properties `GenericSerif`, `GenericSansSerif`, and `GenericMonospace` return default fonts that satisfy these criteria:

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

However, if you are writing a professional application, then you need to choose your font in a more sophisticated way. Most likely, you will implement your drawing code so that it checks the font families available and selects the appropriate one, perhaps by taking the first available one on a list of preferred fonts. Moreover, if you want your application to be very user-friendly, then the first choice on the list will probably be the one that users selected the last time they ran your software. Usually, if you are dealing with the most popular font families, such as Arial and Times New Roman, you are safe. However, if you try to display text using a font that does not exist, the results are not always predictable. You are quite likely to find that Windows just substitutes the standard system font, which is very easy for the system to draw, but that it does not look very pleasant — and if it does appear in your document, then it is likely to give the impression of software that is of poor quality.

You can find out what fonts are available on your system using a class called `InstalledFontCollection`, which is in the `System.Drawing.Text` namespace. This class implements a property, `Families`, which is an array of all the fonts that are available to use on your system:

```
InstalledFontCollection insFont = new InstalledFontCollection();
FontFamily [] families = insFont.Families;

foreach (FontFamily family in families)
{
    // do processing with this font family
}
```

ENUMERATING FONT FAMILIES EXAMPLE

In this section, you work through a quick example, `EnumFontFamilies`, which lists all the font families available on the system and illustrates them by displaying the name of each family using an appropriate font (the 12-point regular version of that font family). Figure 48-16 shows the result of running `EnumFontFamilies`.

Of course, the results that you get will depend on the fonts you have installed on your computer.

For this example, you create a standard C# Windows application, `ListFonts`. You start by adding an extra namespace to be searched. You will be using the `InstalledFontCollection` class, which is defined in `System.Drawing.Text`.

```
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;
```

You then add the following constant to the `Form1` class:

```
private const int margin = 10;
```

`margin` is the size of the left and top margin between the text and the edge of the document — it stops the text from appearing right at the edge of the client area.

This is designed as a quick-and-easy way of showing off font families; therefore, the code is crude and in many instances does not do things the way you ought to in a real application. For example, here you hard-code an estimated value for the document size of (200, 1500) and set the `AutoScrollMinSize` property to this value using the Visual Studio 2010 properties window. Typically, you would have to examine the text to be displayed to work out the document size. You do that in the next section.

Here is the `OnPaint()` method:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    int verticalCoordinate = margin;
    InstalledFontCollection insFont = new InstalledFontCollection();
    FontFamily [] families = insFont.Families;
    e.Graphics.Transform(AutoScrollPosition.X,
                        AutoScrollPosition.Y);
    foreach (FontFamily family in families)
    {
        if (family.IsStyleAvailable(FontStyle.Regular))
        {
            Font f = new Font(family.Name, 10);
            Point topLeftCorner = new Point(margin, verticalCoordinate);
            verticalCoordinate += f.Height;
            e.Graphics.DrawString (family.Name, f,
                                Brushes.Black, topLeftCorner);
            f.Dispose();
        }
    }
}
```



FIGURE 48-16



[code download ListFonts.sln](#)

In this code, you start by using an `InstalledFontCollection` object to obtain an array that contains details of all the available font families. For each family, you instantiate a 10-point `Font`. You use a simple

constructor for `Font` — there are many more that allow additional options to be specified. The constructor takes two parameters, the name of the family and the size of the font:

```
Font f = new Font(family.Name, 10);
```

This constructor builds a font that has the regular style. To be on the safe side, however, you first check that this style is available for each font family before attempting to display anything using that font. This is done using the `FontFamily.IsStyleAvailable()` method. This check is important because not all fonts are available in all styles:

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

`FontFamily.IsStyleAvailable()` takes one parameter, a `FontStyle` enumeration. This enumeration contains a number of flags that might be combined with the bitwise OR operator. The possible flags are `Bold`, `Italic`, `Regular`, `Strikeout`, and `Underline`.

Finally, note that you use a property of the `Font` class, `Height`, which returns the height needed to display text of that font, to work out the line spacing:

```
Font f = new Font(family.Name, 10);
Point topLeftCorner = new Point(margin, verticalCoordinate);
verticalCoordinate += f.Height;
```

Again, to keep things simple, this version of `OnPaint()` reveals some bad programming practices. For example, you have not bothered to check what area of the document actually needs drawing — you just tried to display everything. Also, instantiating a `Font` is, as remarked earlier, a computationally intensive process, so you really ought to save the fonts rather than instantiating new copies every time `OnPaint()` is called. Because of the way the code has been designed, you might note that this example actually takes a noticeable amount of time to paint itself. To try to conserve memory and help the garbage collector, you do, however, call `Dispose()` on each font instance after you have finished with it. If you did not, after 10 or 20 paint operations, there would be a lot of wasted memory storing fonts that are no longer needed.

EDITING A TEXT DOCUMENT: THE CAPSEDITOR EXAMPLE

You now come to the extended example in this chapter. The `CapsEditor` example is designed to demonstrate how the principles of drawing that you have learned so far have to be applied in a more realistic context. The `CapsEditor` example does not require any new material, apart from responding to user input via the mouse, but it shows how to manage the drawing of text so that the application maintains performance while ensuring that the contents of the client area of the main window are always kept up-to-date.

The `CapsEditor` program allows the user to read in a text file, which is then displayed line by line in the client area. If the user double-clicks any line, then that line will be changed to all uppercase. That is literally all the example does. Even with this limited set of features, you will find that the work involved in making sure everything is displayed in the right place while considering performance issues is quite complex. In particular, you have a new element here: the contents of the document can change — either when the user selects the menu option to read a new file, or when she double-clicks to capitalize a line. In the first case, you need to update the document size so the scrollbars still work correctly, and you have to redisplay everything. In the second case, you need to check carefully whether the document size has changed, and what text needs to be redisplayed.

This section starts by reviewing the appearance of `CapsEditor`. When the application is first run, it has no document loaded and resembles Figure 48-17.

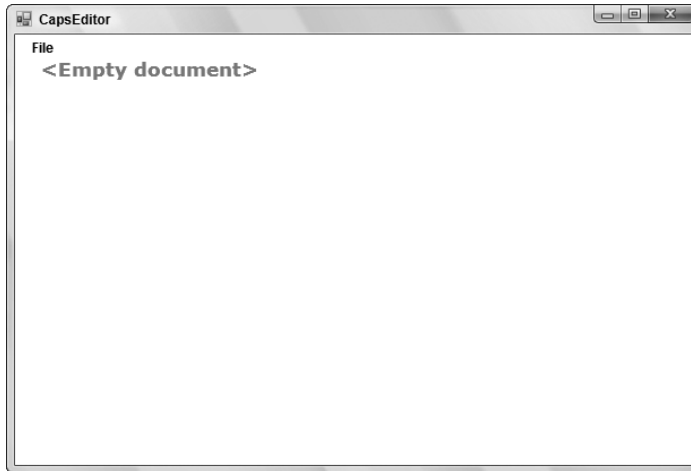


FIGURE 48-17

The File menu has two options: Open, which evokes `OpenFileDialog` when selected and reads in whatever file the user clicks, and Exit, which closes the application when clicked. Figure 48-18 shows `CapsEditor` displaying its own source file, `Form1.cs`. (A few lines have been double-clicked in this image to convert them to uppercase.)

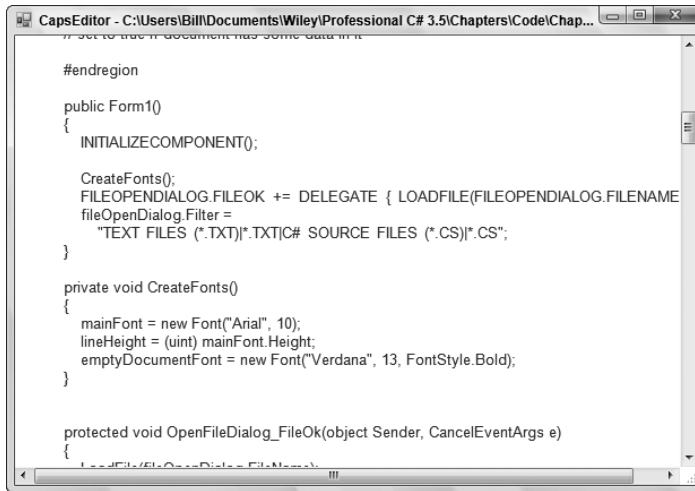


FIGURE 48-18

The sizes of the horizontal and vertical scrollbars are correct. The client area will scroll just enough to view the entire document. `CapsEditor` does not try to wrap lines of text — the example is already complicated enough as is. It just displays each line of the file exactly as it is read in. There are no limits to the size of the file, but you are assuming that it is a text file and does not contain any nonprintable characters.

Begin by adding a `using` command:

```
using System;
using System.Collections;
using System.ComponentModel;
```



```
using System.Drawing;
using System.IO;
using System.Windows.Forms;
```

You will be using the `StreamReader` class, which is in the `System.IO` namespace. Next, you add some fields to the `Form1` class:



```
#region Constant fields
private const string standardTitle = "CapsEditor";
// default text in titlebar

private const uint margin = 10;
// horizontal and vertical margin in client area

#endregion
#region Member fields
// The 'document'
private readonly List<TextLineInformation> documentLines =
    new List<TextLineInformation>();
private uint lineHeight; // height in pixels of one line
private Size documentSize; // how big a client area is needed to
// display document

private uint nLines; // number of lines in document
private Font mainFont; // font used to display all lines
private Font emptyDocumentFont; // font used to display empty message
private readonly Brush mainBrush = Brushes.Blue;
// brush used to display document text
private readonly Brush emptyDocumentBrush = Brushes.Red;
// brush used to display empty document message
private Point mouseDoubleClickPosition;
// location mouse is pointing to when double-clicked
private readonly OpenFileDialog fileOpenDialog = new OpenFileDialog();
// standard open file dialog
private bool documentHasData = false;
// set to true if document has some data in it
#endregion
```

[*code download CapsEditor.sln*](#)

Most of these fields should be self-explanatory. The `documentLines` field is a `List<TextLineInformation>` that contains the actual text of the file that has been read in. Actually, this is the field that contains the data in the document. Each element of `documentLines` contains information for one line of text that has been read in. Because it is a `List<TextLineInformation>` rather than a plain array, you can dynamically add elements to it as you read in a file.

As previously mentioned, each `documentLines` element contains information about a line of text. This information is actually an instance of another class, `TextLineInformation`:

```
class TextLineInformation
{
    public string Text;
    public uint Width;
}
```

`TextLineInformation` looks like a classic case where you would normally use a struct rather than a class because it is just there to group a couple of fields. However, its instances are always accessed as elements of a `List<TextLineInformation>`, which expects its elements to be stored as reference types.

Each `TextLineInformation` instance stores a line of text — and that can be thought of as the smallest item that is displayed as a single item. In general, for each similar item in a GDI+ application, you would probably want to store the text of the item, as well as the world coordinates of where it should be displayed and its size. (The page coordinates will change frequently, whenever the user scrolls, whereas world coordinates will normally change only when other parts of the document are modified in some way.) In this case, you have stored only the `Width` of the item because the height in this case is just the height of whatever

your selected font is. It is the same for all lines of text so there is no point storing the height separately for each one; you store it once, in the `Form1.LineHeight` field. As for the position, well, in this case, the x coordinate is just equal to the margin, and the y coordinate is easily calculated here:

```
margin + lineHeight*(however many lines are above this one)
```

If you had been trying to display and manipulate, say, individual words instead of complete lines, then the x position of each word would have to be calculated using the widths of all the previous words on that line of text, but the intent is to keep it simple here, which is why you are treating each line of text as one single item.

Let's turn to the main menu now. This part of the application is more the realm of Windows Forms (see Chapter 39) than of GDI+. Add the menu options using the design view in Visual Studio 2010, but rename them `menuFile`, `menuFileOpen`, and `menuFileExit`. Next, add event handlers for the File Open and File Exit menu options using the Visual Studio 2010 properties window. The event handlers have their Visual Studio 2010-generated names of `menuFileOpen_Click()` and `menuFileExit_Click()`.

Add some extra initialization code in the `Form1()` constructor:



```
public Form1()
{
    InitializeComponent();
    CreateFonts();
    fileOpenDialog.FileOk += delegate { LoadFile(fileOpenDialog.FileName); };
    fileOpenDialog.Filter =
        "Text files (*.txt)|*.txt|C# source files (*.cs)|*.cs";
}
```

code download CapsEditor.sln

You add the event handler here for instances when the user clicks OK in the File Open dialog box. You have also set the filter for the Open File dialog box, so that you can load text files only. The example in this case only uses .txt files, in addition to the C# source files, so you can use the application to examine the source code for the samples.

`CreateFonts()` is a helper method that sorts out the fonts you intend to use:

```
private void CreateFonts()
{
    mainFont = new Font("Arial", 10);
    lineHeight = (uint)mainFont.Height;
    emptyDocumentFont = new Font("Verdana", 13, FontStyle.Bold);
}
```

The actual definitions of the handlers are pretty standard:

```
protected void menuFileOpen_Click(object sender, EventArgs e)
{
    fileOpenDialog.ShowDialog();
}
protected void menuFileExit_Click(object sender, EventArgs e)
{
    Close();
}
```

Next, take a look at the `LoadFile()` method. It handles the opening and reading of a file (as well as ensuring a Paint event is raised to force a repaint with the new file):



```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
```

```

nLines = 0;
TextLineInformation nextLineInfo;
while ( (nextLine = sr.ReadLine()) != null)
{
    nextLineInfo = new TextLineInformation();
    nextLineInfo.Text = nextLine;
    documentLines.Add(nextLineInfo);
    ++nLines;
}
sr.Close();
documentHasData = (nLines>0) ? true: false;
CalculateLineWidths();
CalculateDocumentSize();
Text = standardTitle + "-" + FileName;
Invalidate();
}

```

[*code download CapsEditor.sln*](#)

Most of this function is just standard file-reading (see Chapter 29, “Manipulating Files and the Registry”). Note that as the file is read, you progressively add lines to `documentLines` `ArrayList`, so this array ends up containing information for each of the lines in order. After you have read in the file, you set the `documentHasData` flag, which indicates whether there is actually anything to display. Your next task is to work out where everything is to be displayed, and, having done that, how much client area you need to display the file as well as the document size that will be used to set the scrollbars. Finally, you set the title bar text and call `Invalidate()`. `Invalidate()` is an important method supplied by Microsoft, so the next section discusses its use first, before examining the code for the `CalculateLineWidths()` and `CalculateDocumentSize()` methods.

The Invalidate() Method

`Invalidate()` is a member of `System.Windows.Forms.Form`. It marks an area of the client window as invalid and, therefore, in need of repainting, and then makes sure a `Paint` event is raised. `Invalidate()` has a couple of overrides: you can pass it a rectangle that specifies (in page coordinates) precisely which area of the window needs repainting. If you do not pass any parameters, it just marks the entire client area as invalid.

If you know that something needs painting, why don't you just call `OnPaint()` or some other method to do the painting directly? The answer is that, in general, calling painting routines directly is regarded as bad programming practice — if your code decides it wants some painting done, you should call `Invalidate()`. Here is why:

- Drawing is almost always the most processor-intensive task a GDI+ application will carry out, so doing it in the middle of other work holds up the other work. With the example, if you had directly called a method to do the drawing from the `LoadFile()` method, then the `LoadFile()` method would not return until that drawing task was complete. During that time, your application cannot respond to any other events. However, by calling `Invalidate()`, you are simply getting Windows to raise a `Paint` event before immediately returning from `LoadFile()`. Windows is then free to examine the events that are in line to be handled. How this works internally is that the events sit as what are known as *messages* in a *message queue*. Windows periodically examines the queue, and if there are events in it, then it picks one and calls the corresponding event handler. Although the `Paint` event might be the only one sitting in the queue (so `OnPaint()` is called immediately anyway), in a more complex application there might be other events that ought to get priority over your `Paint` event. In particular, when the user has decided to quit the application, this will be marked by a message known as `WM_QUIT`.
- If you have a more complicated, multithreaded application, then you will probably want just one thread to handle all the drawing. Using `Invalidate()` to route all drawing through the message

queue provides a good way of ensuring that the same thread does all the drawing, no matter what other thread requested the drawing operation. (Whatever thread is responsible for the message queue will be the thread that called `Application.Run()`.)

- There is an additional performance-related reason. Suppose that a couple of different requests to draw part of the screen come in at about the same time. Maybe your code has just modified the document and wants to ensure the updated document is displayed, while at the same time the user has just moved another window that was covering part of the client area out of the way. By calling `Invalidate()`, you are giving Windows a chance to notice that this has occurred. Windows can then merge the `Paint` events if appropriate, combining the invalidated areas, so that the painting is only done once.
- The code to do the painting is probably going to be one of the most complex parts of the code in your application, especially if you have a very sophisticated user interface. The people who have to maintain your code in a couple of years time will thank you for having kept your painting code all in one place and as simple as you reasonably can — something that is easier to do if you do not have too many pathways into it from other parts of the program.

The bottom line from all this is that it is good practice to keep all your painting in the `OnPaint()` routine, or in other methods called from that method. However, you have to strike a balance; if you want to replace just one character on the screen and you know perfectly well that it won't affect anything else that you have drawn, then you might decide that it's not worth the overhead of going through `Invalidate()` and just write a separate drawing routine.



In a very complicated application, you might even write a full class that takes responsibility for drawing to the screen. A few years ago when MFC was the standard technology for GDI-intensive applications, MFC followed this model with a C++ class, `C<ApplicationName>View`, which was responsible for painting. However, even in this case, this class had one member function, `OnDraw()`, which was designed to be the entry point for most drawing requests.

Calculating Item Sizes and Document Size

This section returns to the `CapsEditor` example and examines the `CalculateLineWidths()` and `CalculateDocumentSize()` methods called from `LoadFile()`:



Available for
download on
Wrox.com

```
private void CalculateLineWidths()
{
    Graphics dc = this.CreateGraphics();
    foreach (TextLineInformation nextLine in documentLines)
    {
        nextLine.Width = (uint)dc.MeasureString(nextLine.Text,
            mainFont).Width;
    }
}
```

code download CapsEditor.sln

This method simply runs through each line that has been read in and uses the `Graphics.MeasureString()` method to work out and store how much horizontal screen space the string requires. You store the value because `MeasureString()` is computationally intensive. If the `CapsEditor` sample had not been simple enough to easily work out the height and location of each item, then this method would almost certainly have needed to be implemented in such a way as to compute all those quantities, too.

Now that you know how big each item on the screen is and you can calculate where each item goes, you are in a position to work out the actual document size. The height is the number of lines multiplied by the height of each line. The width will need to be worked out by iterating through the lines to find the

longest. For both height and width, you will also want to make an allowance for a small margin around the displayed document to make the application look more attractive.

The following is the method that calculates the document size:



```
private void CalculateDocumentSize()
{
    if (!documentHasData)
    {
        documentSize = new Size(100, 200);
    }
    else
    {
        documentSize.Height = (int)(nLines*lineHeight) + 2*(int)margin;
        uint maxLineLength = 0;
        foreach (TextLineInformation nextWord in documentLines)
        {
            uint tempLineLength = nextWord.Width;
            if (tempLineLength > maxLineLength)
            {
                maxLineLength = tempLineLength;
            }
        }
        maxLineLength += 2*margin;
        documentSize.Width = (int)maxLineLength;
    }
    AutoScrollMinSize = documentSize;
}
```

[code download CapsEditor.sln](#)

This method first checks whether there is any data to be displayed. If there is not, then you cheat a bit and use a hard-coded document size, which is big enough to display the big red <Empty Document> warning. If you had wanted to really do it properly, you would have used `MeasureString()` to check how big that warning actually is.

When you have worked out the document size, you tell the `Form` instance what the size is by setting the `Form.AutoScrollMinSize` property. When you do this, something interesting happens behind the scenes. In the process of setting this property, the client area is invalidated and a `Paint` event is raised, for the very sensible reason that changing the size of the document means scrollbars will need to be added or modified and the entire client area will almost certainly be repainted. Why is that interesting? If you look back at the code for `LoadFile()`, you will realize that the call to `Invalidate()` in that method is actually redundant. The client area will be invalidated anyway when you set the document size. The explicit call to `Invalidate()` was left in the `LoadFile()` implementation to illustrate how you should normally do things. In fact, in this case, calling `Invalidate()` again will only needlessly request a duplicate `Paint` event. However, this in turn illustrates how `Invalidate()` gives Windows the chance to optimize performance. The second `Paint` event will not, in fact, get raised: Windows will see that there is a `Paint` event already sitting in the queue and will compare the requested invalidated regions to see if it needs to do anything to merge them. In this case, both `Paint` events will specify the entire client area, so nothing needs to be done, and Windows will quietly drop the second `Paint` request. Of course, going through that process will take up a little bit of processor time, but it will be a negligible amount of time compared to how long it takes to actually do some painting.

OnPaint()

Now that you have seen how `CapsEditor` loads the file, it's time to look at how the painting is done:



```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
```

```

int scrollPositionX = AutoScrollPosition.X;
int scrollPositionY = AutoScrollPosition.Y;
dc.TranslateTransform(scrollPositionX, scrollPositionY);
if (!documentHasData)
{
    dc.DrawString("<Empty document>", emptyDocumentFont,
        emptyDocumentBrush, new Point(20,20));
    base.OnPaint(e);
    return;
}
// work out which lines are in clipping rectangle
int minLineInClipRegion =
    WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
        scrollPositionY);

if (minLineInClipRegion == -1)
{
    minLineInClipRegion = 0;
}
int maxLineInClipRegion =
    WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom -
        scrollPositionY);

if (maxLineInClipRegion >= documentLines.Count ||
    maxLineInClipRegion == -1)
{
    maxLineInClipRegion = documentLines.Count-1;
}
TextLineInformation nextLine;
for (int i=minLineInClipRegion; i<=maxLineInClipRegion; i++)
{
    nextLine = (TextLineInformation)documentLines[i];
    dc.DrawString(nextLine.Text, mainFont, mainBrush,
        LineIndexToWorldCoordinates(i));
}
}

```

[*code download CapsEditor.sh*](#)

At the heart of this `OnPaint()` override is a loop that goes through each line of the document, calling `Graphics.DrawString()` to paint each one. The rest of this code is mostly concerned with optimizing the painting — figuring out what exactly needs painting instead of rushing in and telling the graphics instance to redraw everything.

You begin by checking if there is any data in the document. If there is not, then you draw a quick message saying so, call the base class's `OnPaint()` implementation, and exit. If there is data, then you start looking at the clipping rectangle by calling another method, `WorldYCoordinateToLineIndex()`. This method is examined next, but essentially it takes a given y position relative to the top of the document, and works out what line of the document is being displayed at that point.

The first time you call the `WorldYCoordinateToLineIndex()` method, you pass it the coordinate value (`e.ClipRectangle.Top - scrollPositionY`). This is just the top of the clipping region, converted to world coordinates. If the return value is `-1`, you play it safe and assume that you need to start at the beginning of the document (this is the case if the top of the clipping region is within the top margin).

After you have done all that, you essentially repeat the same process for the bottom of the clipping rectangle to find the last line of the document that is inside the clipping region. The indices of the first and last lines are respectively stored in `minLineInClipRegion` and `maxLineInClipRegion`, so then you can just run a `for` loop between these values to do your painting. Inside the painting loop, you actually need to do roughly the reverse transformation to the one performed by `WorldYCoordinateToLineIndex()`. You are given the index of a line of text, and you need to check where it should be drawn. This calculation is actually quite simple, but you have wrapped it up in another method, `LineIndexToWorldCoordinates()`, which returns

the required coordinates of the top-left corner of the item. The returned coordinates are world coordinates, but that is fine because you have already called `TranslateTransform()` on the `Graphics` object so that you need to pass it as a world coordinate, rather than a page, when asking it to display items.

Coordinate Transforms

This section examines the implementation of the helper methods that are written in the `CapsEditor` sample to help you with coordinate transforms. These are the `WorldYCoordinateToLineIndex()` and `LineIndexToWorldCoordinates()` methods referred to in the previous section, as well as a couple of other methods.

First, `LineIndexToWorldCoordinates()` takes a given line index, and works out the world coordinates of the top-left corner of that line, using the known margin and line height:



```
private Point LineIndexToWorldCoordinates(int index)
{
    Point TopLeftCorner = new Point(
        (int)margin, (int)(lineHeight*index + margin));
    return TopLeftCorner;
}
```

[code download CapsEditor.sln](#)

You also use a method that roughly does the reverse transform in `OnPaint()`.

`WorldYCoordinateToLineIndex()` works out the line index, but it takes into account only a vertical world coordinate because it is used to work out the line index corresponding to the top and bottom of the clip region:

```
private int WorldYCoordinateToLineIndex(int y)
{
    if (y < margin)
    {
        return -1;
    }
    return (int)((y-margin)/lineHeight);
}
```

There are three more methods, which will be called from the handler routine that responds to the user double-clicking the mouse. First, you have a method that works out the index of the line being displayed at given world coordinates. Unlike `WorldYCoordinateToLineIndex()`, this method takes into account the x and y positions of the coordinates. It returns -1 if there is no line of text covering the coordinates passed in:

```
private int WorldCoordinatesToLineIndex(Point position)
{
    if (!documentHasData)
    {
        return -1;
    }
    if (position.Y < margin || position.X < margin)
    {
        return -1;
    }
    int index = (int)(position.Y-margin)/((int)this.lineHeight;
    // check position is not below document
    if (index >= documentLines.Count)
    {
        return -1;
    }
    // now check that horizontal position is within this line
    TextLineInformation theLine =
        (TextLineInformation)documentLines[index];
```

```
        if (position.X > margin + theLine.Width)
        {
            return -1;
        }
        // all is OK. We can return answer
        return index;
    }
}
```

Finally, on occasion, you also need to convert between line index and page, rather than world, coordinates. The following methods achieve this:

```
private Point LineIndexToPageCoordinates(int index)
{
    return LineIndexToWorldCoordinates(index) +
           new Size(AutoScrollPosition);
}
private int PageCoordinatesToLineIndex(Point position)
{
    return WorldCoordinatesToLineIndex(position-new
                                       Size(AutoScrollPosition));
}
```

Note that when converting to page coordinates, you add the `AutoScrollPosition`, which is negative.

Although these methods by themselves do not look particularly interesting, they do illustrate a general technique that you will probably need to use often. With GDI+, you will often find yourself in a situation where you have been given specific coordinates (for example the coordinates of where the user has clicked the mouse), and you will need to figure out what item is being displayed at that point. Or it could happen the other way around — where should a particular display item be displayed? Hence, if you are writing a GDI+ application, you will probably find it useful to write methods that do the equivalent of the coordinate transformation methods illustrated here.

Responding to User Input

So far, with the exception of the File menu in the `CapsEditor` sample, everything you have done in this chapter has been one way: the application has talked to the user by displaying information on the screen. Almost all software of course works both ways: the user can talk to the software as well. You are now going to add that functionality to `CapsEditor`.

Getting a GDI+ application to respond to user input is actually a lot simpler than writing the code to draw to the screen. (Chapter 39 covers how to handle user input.) Essentially, you override methods from the `Form` class that are called from the relevant event handler, in much the same way that `OnPaint()` is called when a `Paint` event is raised.

The following table lists the methods you might want to override when the user clicks or moves the mouse.

METHOD	CALLED WHEN ...
<code>OnClick(EventArgs e)</code>	Mouse is clicked.
<code>OnDoubleClick(EventArgs e)</code>	Mouse is double-clicked.
<code>OnMouseDown(MouseEventArgs e)</code>	Left mouse button is pressed.
<code>OnMouseHover(MouseEventArgs e)</code>	Mouse stays still somewhere after moving.
<code>OnMouseMove(MouseEventArgs e)</code>	Mouse is moved.
<code>OnMouseUp(MouseEventArgs e)</code>	Left mouse button is released.

If you want to detect when the user types in any text, then you will probably want to override the methods listed in the following table.

METHOD	CALLED WHEN ...
OnKeyDown (EventArgs e)	A key is pressed.
OnKeyPress (KeyPressEventArgs e)	A key is pressed and released.
OnKeyUp (EventArgs e)	A pressed key is released.

Note that some of these events overlap. For example, when the user presses a mouse button, the `MouseDown` event is raised. If the button is immediately released again, then this will raise the `MouseUp` event and the `Click` event. In addition, some of these methods take an argument that is derived from `EventArgs` rather than an instance of `EventArgs` itself. These instances of derived classes can be used to give more information about a particular event. `MouseEventArgs` has two properties, `X` and `Y`, which give the device coordinates of the mouse at the time it was pressed. Both `EventArgs` and `KeyPressEventArgs` have properties that indicate which key or keys the event concerns.

That is all there is to it. It is up to you to think about the logic of precisely what you want to do. The only point to note is that you will probably find yourself doing a bit more logic work with a GDI+ application than you would have with a `Windows.Forms` application. That is because in a `Windows.Forms` application you are typically responding to high-level events (`TextChanged` for a text box, for example). By contrast, with GDI+, the events tend to be more elementary — the user clicks the mouse or presses the H key. The action your application takes is likely to depend on a sequence of events rather than on a single event. For example, say your application works similar to Microsoft Word for Windows: to select some text, the user clicks the left mouse button, and then moves the mouse and releases the left mouse button. Your application receives the `MouseDown` event, but there is not much you can do with this event except record that the mouse was clicked with the cursor in a certain position. Then, when the `MouseMove` event is received, you need to check from the record whether the left button is currently down, and if so, highlight text as the user selects it. When the user releases the left mouse button, your corresponding action (in the `OnMouseUp()` method) will need to check whether any dragging took place while the mouse button was down and act accordingly within the method. Only at this point is the sequence complete.

Another point to consider is that, because certain events overlap, you will often have a choice of which event you want your code to respond to.

The golden rule is to think carefully about the logic of every combination of mouse movement or click and keyboard event that the user might initiate, and ensure that your application responds in a way that is intuitive and in accordance with the expected behavior of applications in *every* case. Most of your work here will be in thinking rather than in coding, although the coding you do will be tricky because you might need to take into account many combinations of user input. For example, what should your application do if the user starts typing in text while one of the mouse buttons is held down? It might sound like an improbable combination, but eventually some user is going to try it!

The `CapsEditor` example keeps things very simple, so you do not really have any combinations to think about. The only thing you are going to respond to in the example is when the user double-clicks, in which case you capitalize whatever line of text the mouse pointer is hovering over.

This should be a simple task, but there is one snag. You need to trap the `DoubleClick` event, but the previous table shows that this event takes an `EventArgs` parameter, not a `MouseEventArgs` parameter. The trouble is that you need to know where the mouse is when the user double-clicks if you are to correctly identify the line of text to be capitalized — and you need a `MouseEventArgs` parameter to do that. There are two workarounds. One is to use a static method implemented by the `Form1` object `Control.MousePosition` to find the mouse position:


Available for
download on
Wrox.com

```
protected override void OnDoubleClick(EventArgs e)
{
    Point MouseLocation = Control.MousePosition;
    // handle double click
}
```

[code download CapsEditor.sln](#)

In most cases, this will work. However, there could be a problem if your application (or even some other application with a high priority) is doing some computationally intensive work at the moment the user double-clicks. It just might happen in that case that the `OnDoubleClick()` event handler does not get called until perhaps half a second or so *after* the user has double-clicked. You do not want such delays because they usually annoy users intensely, but even so, occasionally it does happen and sometimes for reasons beyond the control of your application (a slow computer, for instance). The trouble is that half a second is easily enough time for the mouse to be moved halfway across the screen, in which case your call to `Control.MousePosition` will return the completely wrong location!

A better approach here is to rely on one of the many overlaps between mouse event meanings. The first part of double-clicking a mouse involves pressing the left button down. This means that if `OnDoubleClick()` is called, you know that `OnMouseDown()` has also just been called, with the mouse at the same location. You can use the `OnMouseDown()` override to record the position of the mouse, ready for `OnDoubleClick()`. This is the approach taken in `CapsEditor`:

```
protected override void OnMouseDown(MouseEventArgs e)
{
    base.OnMouseDown(e);
    mouseDoubleClickPosition = new Point(e.X, e.Y);
}
```

Now look at the `OnDoubleClick()` override. There is quite a bit more work to do here:

```
protected override void OnDoubleClick(EventArgs e)
{
    int i = PageCoordinatesToLineIndex(mouseDoubleClickPosition);
    if (i >= 0)
    {
        TextLineInformation lineToBeChanged =
            (TextLineInformation)documentLines[i];
        lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
        Graphics dc = this.CreateGraphics();
        uint newWidth = (uint)dc.MeasureString(lineToBeChanged.Text,
            mainFont).Width;

        if (newWidth > lineToBeChanged.Width)
            lineToBeChanged.Width = newWidth;
        if (newWidth+2*margin > this.documentSize.Width)
        {
            documentSize.Width = (int)newWidth;
            AutoScrollMinSize = this.documentSize;
        }
        Rectangle changedRectangle = new Rectangle(
            LineIndexToPageCoordinates(i),
            new Size((int)newWidth,
                (int)this.lineHeight));

        Invalidate(changedRectangle);
    }
    base.OnDoubleClick(e);
}
```

[code download CapsEditor.sln](#)

You start off by calling `PageCoordinatesToLineIndex()` to work out which line of text the mouse pointer was hovering over when the user double-clicked. If this call returns `-1`, then you weren't over any text, so there is nothing to do — except, of course, call the base class version of `OnDoubleClick()` to let Windows do any default processing.

Assuming that you have identified a line of text, you can use the `string.ToUpper()` method to convert it to uppercase. That was the easy part. The hard part is figuring out what needs to be redrawn where. Fortunately, because this example is simple, there are not too many combinations. You can assume that



converting to uppercase will always either leave the width of the line on the screen unchanged or increase it. Capital letters are bigger than lowercase letters; therefore, the width will never go down. You also know that because you are not wrapping lines, your line of text will not overflow to the next line and push out other text below. Your action of converting the line to uppercase will not, therefore, actually change the locations of any of the other items being displayed. That is a big simplification!

The next thing the code does is use `Graphics.MeasureString()` to work out the new width of the text. There are now just two possibilities:

- The new width might make your line the longest line and cause the width of the entire document to increase. If that is the case, then you will need to set `AutoScrollMinSize` to the new size so that the scrollbars are correctly placed.
- The size of the document might be unchanged.

In either case, you need to get the screen redrawn by calling `Invalidate()`. Only one line has changed; therefore, you do not want to have the entire document repainted. Rather, you need to work out the bounds of a rectangle that contains just the modified line, so that you can pass this rectangle to `Invalidate()`, ensuring that just that line of text is repainted. That is precisely what the previous code does. Your call to `Invalidate()` initiates a call to `OnPaint()` when the mouse event handler finally returns. Keeping in mind the earlier comments about the difficulty in setting a break point in `OnPaint()`, if you run the sample and set a break point in `OnPaint()` to trap the resultant painting action, then you will find that the `PaintEventArgs` parameter to `OnPaint()` does indeed contain a clipping region that matches the specified rectangle. And because you have overloaded `OnPaint()` to take careful account of the clipping region, only one required line of text will be repainted.

PRINTING

So far, the chapter has focused exclusively on drawing to the screen. However, at some point you will probably also want to be able to produce a hard copy of the data. That is the topic of this section. You are going to extend the `CapsEditor` sample so that it is able to print preview and print the document that is being edited.

Unfortunately, there is not enough space to go into too much detail about printing here, so the printing functionality you will implement is very basic. Typically, when you are implementing the ability for an application to print data, you will need to add three items to the application's main File menu:

- **Page Setup**, which allows the user to choose options such as which pages to print, which printer to use, and so on.
- **Print Preview**, which opens a new Form that displays a mock-up of what the printed copy should look like.
- **Print**, which prints the document.

In this case, to keep things simple, you will not implement a Page Setup menu option. Printing will only be possible using default settings. Note, however, that if you want to implement Page Setup, Microsoft has already written a page setup dialog class for you to use: `System.Windows.Forms.PrintDialog`. You will normally want to write an event handler that displays this form and saves the settings chosen by the user.

In many ways, printing is just the same as displaying to a screen. You will be supplied with a device context (`Graphics` instance) and call all the usual display commands against that instance. Microsoft has written a number of classes to assist you in doing this; the two main ones are `System.Drawing.Printing.PrintDocument` and `System.Windows.Forms.PrintPreviewDialog`. These two classes handle the process of making sure that drawing instructions passed to a device context are handled appropriately for printing, leaving you to think about the logic of what to print where.

Some important differences exist between printing or print previewing on the one hand, and displaying to the screen on the other hand. Printers cannot scroll; instead, they turn out pages. Therefore, you will need to make sure that you find a sensible way of dividing your document into pages and draw each page as requested. Among other things, that means calculating how much of your document will fit onto a single page and, therefore, how many pages you will need and which page each part of the document needs to be written to.

Despite these complications, the process of printing is quite simple. Programmatically, the steps you need to go through look roughly like this:

- **Printing** — You instantiate a `PrintDocument` object and call its `Print()` method. This method signals the `PrintPage` event to print the first page. `PrintPage` takes a `PrintPageEventArgs` parameter, which supplies information concerning paper size and setup, as well as a `Graphics` object used for the drawing commands. You should therefore have written an event handler for this event, and have implemented this handler to print a page. This event handler should also set a Boolean property of the `PrintPageEventArgs` called `HasMorePages` to either `true` or `false` to indicate whether there are more pages to be printed. The `PrintDocument.Print()` method will repeatedly raise the `PrintPage` event until it sees that `HasMorePages` has been set to `false`.
- **Print Previewing** — In this case, you instantiate both a `PrintDocument` object and a `PrintPreviewDialog` object. You attach the `PrintDocument` to the `PrintPreviewDialog` (using the property `PrintPreviewDialog.Document`) and then call the dialog's `ShowDialog()` method. This method modally displays the dialog, which turns out to be a standard Windows print preview form and which displays pages of the document. Internally, the pages are displayed again by repeatedly raising the `PrintPage` event until the `HasMorePages` property is `false`. There is no need to write a separate event handler for this; you can use the same event handler as used for printing each page because the drawing code ought to be identical in both cases. (After all, whatever is print previewed ought to look identical to the printed version!)

Implementing Print and Print Preview

Now that this process has been outlined in broad strokes, in this section you see how this works in code terms. You can download the code as the `PrintingCapsEdit` project at www.wrox.com; it consists of the `CapsEditor` project with the changes displayed in the following snippet.

You begin by using the Visual Studio 2010 design view to add two new items to the File menu: `Print` and `Print Preview`. You also use the properties window to name these items `menuFilePrint` and `menuFilePrintPreview`, and to set them to be disabled when the application starts up (you cannot print anything until a document has been opened!). You arrange for these menu items to be enabled by adding the following code to the main form's `LoadFile()` method, which is responsible for loading a file into the `CapsEditor` application:



Available for
download on
Wrox.com

```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
}
```

```

    if (nLines > 0)
    {
        documentHasData = true;
        menuFilePrint.Enabled = true;
        menuFilePrintPreview.Enabled = true;
    }
    else
    {
        documentHasData = false;
        menuFilePrint.Enabled = false;
        menuFilePrintPreview.Enabled = false;
    }
    CalculateLineWidths();
    CalculateDocumentSize();
    Text = standardTitle + "-" + FileName;
    Invalidate();
}

```

[code download Printing.sln](#)

The above code is the new code added to this method. Next, you add a member field to the `Form1` class:

```

public partial class Form1: Form
{
    private int pagesPrinted = 0;

```

This field will be used to indicate which page you are currently printing. You are making it a member field because you will need to remember this information between calls to the `PrintPage` event handler.

Next, you will find the event handlers that handle the selection of the Print or Print Preview menu options:



```

private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintPreviewDialog ppd = new PrintPreviewDialog();
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += this.pd_PrintPage;
    ppd.Document = pd;
    ppd.ShowDialog();
}

private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += new PrintPageEventHandler
        (this.pd_PrintPage);
    pd.Print();
}

```

[code download Printing.sln](#)

You have already seen the steps involved in printing, and you can see that these event handlers are simply implementing that procedure. In both cases, you are instantiating a `PrintDocument` object and attaching an event handler to its `PrintPage` event. In the case of printing, you call `PrintDocument.Print()`, whereas for print previewing, you attach the `PrintDocument` object to a `PrintPreviewDialog` and call the preview dialog box object's `ShowDialog()` method. The real work to the `PrintPage` event is done in the event handler. Here is how this handler looks:



```

private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
    float yPos = 0;
    float leftMargin = e.MarginBounds.Left;

```

```

float topMargin = e.MarginBounds.Top;
string line = null;
// Calculate the number of lines per page.
int linesPerPage = (int)(e.MarginBounds.Height /
    mainFont.GetHeight(e.Graphics));
int lineNo = pagesPrinted * linesPerPage;
// Print each line of the file.
int count = 0;
while(count < linesPerPage && lineNo < this.nLines)
{
    line = ((TextLineInformation)this.documentLines[lineNo]).Text;
    yPos = topMargin + (count * mainFont.GetHeight(e.Graphics));
    e.Graphics.DrawString(line, mainFont, Brushes.Blue,
        leftMargin, yPos, new StringFormat());
    lineNo++;
    count++;
}
// If more lines exist, print another page.
if(this.nLines > lineNo)
    e.HasMorePages = true;
else
    e.HasMorePages = false;
pagesPrinted++;
}

```

[*code download Printing.sln*](#)

After declaring a couple of local variables, the first thing you do is work out how many lines of text can be displayed on one page, which will be the height of a page divided by the height of a line and rounded down. The height of the page can be obtained from the `PrintPageEventArgs.MarginBounds` property. This property is a `RectangleF` struct that has been initialized to give the bounds of the page. The height of a line is obtained from the `Form1.mainFont` field, which is the font used for displaying the text. There is no reason here for not using the same font for printing too. Note that for the `PrintingCapsEditor` sample, the number of lines per page is always the same, so you arguably could have cached the value the first time you calculated it. However, the calculation is not too hard, and in a more sophisticated application, the value might change, so it is not bad practice to recalculate it every time you print a page.

You also initialize a variable called `lineNo`. This gives the zero-based index of the line of the document that will be the first line of this page. This information is important because, in principle, the `pd_PrintPage()` method could have been called to print any page, not just the first page. `lineNo` is computed as the number of lines per page times the number of pages that have so far been printed.

Next, you run through a loop, printing each line. This loop will terminate either when you find that you have printed all the lines of text in the document, or when you find that you have printed all the lines that will fit on this page, whichever condition occurs first. Finally, you check whether there is any more of the document to be printed, and set the `HasMorePages` property of your `PrintPageEventArgs` accordingly. You also increment the `pagesPrinted` field so that you know to print the correct page the next time the `PrintPage` event handler is invoked.

One point to note about this event handler is that you do not worry about where the drawing commands are being sent. You simply use the `Graphics` object that was supplied with the `PrintPageEventArgs`. The `PrintDocument` class that Microsoft has written will internally take care of making sure that, if you are printing, the `Graphics` object has been hooked up to the printer; if you are print previewing, then the `Graphics` object has been hooked up to the print preview form on the screen.

Finally, you need to ensure that the `System.Drawing.Printing` namespace is searched for type definitions:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

All that remains is to compile the project and check that the code works. Figure 48-19 shows what happens when you run `CapsEdit`, load a text document (as before, you have picked the C# source file for the project), and select `Print Preview`.

In Figure 48-19, the document is scrolled to page 5 and the preview is set to display normal size. The `PrintPreviewDialog` has supplied quite a lot of features, as you can see by looking at the toolbar at the top of the form. The options available include printing the document, zooming in or out, and displaying two, three, four, or six pages together. These options are all fully functional, without your having to do any work. Figure 48-20 shows the result of changing the zoom to auto and clicking to display four pages (third toolbar button from the right).

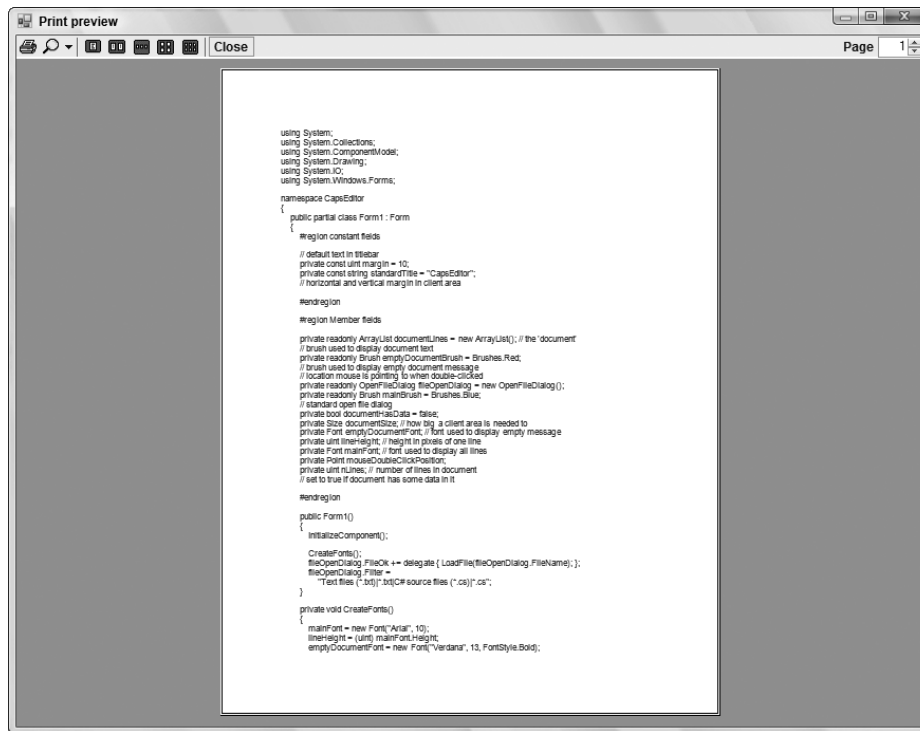


FIGURE 48-19

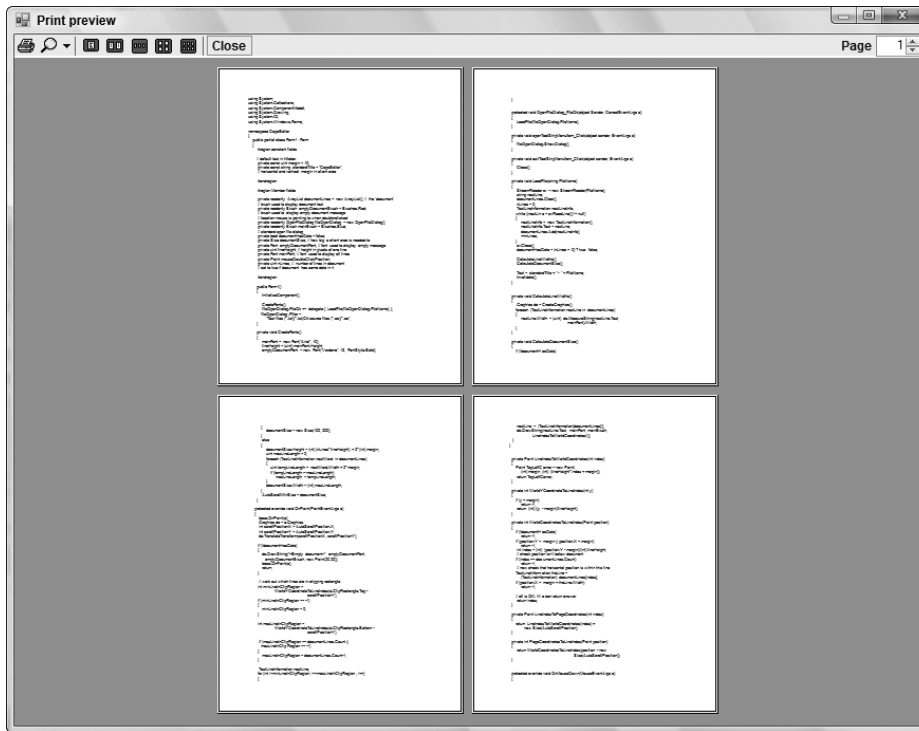


FIGURE 48-20

SUMMARY

This chapter covered the realm of GDI+ — the area of drawing to a display device, where the drawing is done by your code rather than by some predefined control or dialog box. GDI+ is a powerful tool, and there are many .NET base classes available to help you draw to a device. You have seen that the process of drawing is actually relatively simple. In most cases, you can draw text or sophisticated figures or display images with just a couple of C# statements. However, managing your drawing — the behind-the-scenes work involving working out what to draw, where to draw it, and what does or does not need repainting in any given situation — is far more complex and requires careful algorithm design. For this reason, it is also important to have a good understanding of how GDI+ works and what actions Windows takes to get something drawn. In particular, because of the architecture of Windows, it is important that, where possible, drawing should be done by invalidating areas of the window and relying on Windows to respond by issuing a `Paint` event.

Many more .NET classes can be used for drawing than there is space to cover in this chapter. However, if you have worked through it and understood the principles involved in drawing, then you will be in an excellent position to explore these classes by looking at their lists of methods in the SDK documentation and instantiating instances of them to see what they do. In the end, drawing, as with almost any other aspect of programming, requires logic, careful thought, and clear algorithms if you want to go beyond the standard controls. Your software will benefit in both user-friendliness and visual appearance if it is well thought out. Many applications out there rely entirely on controls for their user interface. Although this can be effective, such applications very quickly end up resembling each other. By adding some GDI+ code to do some custom drawing you can mark out your software as distinct and make it appear more original, which can only help increase your sales!