Programmer to Programmer ™

Professional C# 2nd Edition

Written and tested for final release of .NET v1.0

Simon Robinson, K. Scott Allen, Ollie Cornes, Jay Glynn, Zach Greenvoss, Burton Harvey, Christian Nagel, Morgan Skinner, Karli Watson

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

What you need to use this book

The following list is the recommended operating system requirements for running the C# code in this book:

- Windows 2000 Professional or higher with IIS installed
- Windows XP Professional with IIS installed
- Visual Studio .NET Professional or higher

The book is intended for experienced developers, probably from a VB 6, C++, or Java background. Although previous experience of C# or .NET programming is useful, it is not required.

Summary of Contents

19

Graphics with GDI+

This is the second of the two chapters in this book that cover the elements of interacting directly with the user; displaying information on the screen and accepting user input. In Chapter 7 we focused on Windows Forms, where we learned how to display a dialog box or SDI or MDI window, and how to place various controls on it such as buttons, textboxes, and listboxes. We used these familiar, predefined controls at a high level and relied on the fact that they are able to take full responsibility for getting themselves drawn on the display device.

Although these standard controls are powerful, and are by themselves quite adequate for the complete user interface for many applications, there are situations in which you need more flexibility in your user interface. For example, you may want to draw text in a given font in a precise position in a window, display images without using a picture box control, or draw simple shapes or other graphics. None of this can be done with the controls from Chapter 7. To display that kind of output, the application must take direct responsibility for telling the operating system precisely what needs to be displayed where in its window.

Therefore, in this chapter we're going to show you how to draw a variety of items including:

- Lines and simple shapes
- Images from bitmap and other image files
- Text

In the process, we'll also 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). We'll also go into some detail on how devices interpret and display different colors.

We'll start, however, by discussing a technology called **GDI+**. GDI+ consists of the set of .NET base classes that are available to carry out custom drawing on the screen. These classes arrange for the appropriate instructions to be sent to graphics device drivers to ensure the correct output is placed on the monitor screen (or printed to a hard copy).

Understanding Drawing Principles

In this section, we'll examine the basic principles that we need to understand in order to start drawing to the screen. We'll start by giving an overview of GDI, the underlying technology on which GDI+ is based, and see how it and GDI+ are related. Then we'll move on to a couple of simple examples.

GDI and GDI+

In general, one of the strengths of Windows – and indeed of modern operating systems in general – lies in their ability to abstract the details of particular devices away from the developer. For example, you don't need to understand anything about your hard drive device driver in order to programmatically read and write files to 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 very 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, there are many hundreds of different video cards 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. This is why the Windows **Graphical Device Interface** (**GDI**) has always been around since the earliest versions of Windows.

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 your particular video card to do whatever it is you want. Not only this, but if you have several display devices – monitors and printers, say – GDI achieves the remarkable feat of making your printer look the same as your screen as far as your application is concerned. If you want to print something instead of displaying it, you 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 DC 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 don't 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 claims to have made some performance improvements.

The GDI+ part of the .NET base class library is huge, and we will scarcely scratch the surface of its features in this chapter. That's a deliberate decision, because trying to cover more than a tiny fraction of the library would have effectively turned this chapter into a huge reference guide that simply listed classes and methods. It's more important to understand the fundamental principles involved in drawing, so that you will be in a good position to explore the classes available yourself. Full lists of all the classes and methods available in GDI+ are of course available in the MSDN documentation.

Developers coming from a VB background, in particular, are likely to find the concepts involved in drawing quite unfamiliar, since VB's focus lies so strongly in controls that handle their own painting. Those coming from a C++/MFC background are likely to be in more comfortable territory since MFC does require developers to take control of more of the drawing process, using GDI. However, even if you have a good background in GDI, you'll find a lot of the material is new.

GDI+ Namespaces

Here's an overview of the main namespaces you'll need to look in to find the GDI+ base classes:

Namespace	Contains
`System.Drawing`	Most of the classes, structs, enums, and delegates concerned with the basic functionality of drawing
`System.Drawing.Drawing2D`	Provides most of the support for advanced 2D and vector drawing, including antialiasing, geometric transformations, and graphics paths
`System.Drawing.Imaging`	Various classes that assist in the manipulation of images (bitmaps, GIF files, and so on)
`System.Drawing.Printing`	Classes to assist when specifically targeting a printer or print preview window as the "output device"
`System.Drawing.Design`	Some predefined dialog boxes, property sheets, and other user interface elements concerned with extending the design-time user interface
`System.Drawing.Text`	Classes to perform more advanced manipulation of fonts and font families

You should note that almost all of the classes and structs that we use in this chapter will be taken from the `System.Drawing` namespace.

Device Contexts and the Graphics Object

In GDI, the way that you identify which device you want your output to go to is through an object known as the **device context (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 black and white), so you can adjust your output accordingly. If you ask the device to do something it's not capable of, the DC will normally detect this, and take appropriate action (which depending on the situation might mean throwing an error or modifying the request to get the closest match that the device is actually capable of).

However, the DC doesn't only deal 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. Due to 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. Since 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 device context 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, since the `Graphics` class is the class that is responsible for actually handling most drawing operations, very little gets done in GDI+ that doesn't 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

We're going to start off 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.NET as C# Windows applications. Recall that for this type of project the code wizard gives us a class called `Form1`, derived from `System.Windows.Form`, which represents the application's main window. Unless otherwise stated, in all code samples, new or modified code means code that we've added to the wizard-generated code.

> *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've tended to stick to the term window, since in the context of manually drawing items it's rather more meaningful. We'll also talk about the Form when we're referring to the .NET class used to instantiate the form/window. Finally, we'll use the terms drawing and painting interchangeably to describe the process of displaying some item on the screen or other display device.*

The first example will simply create a form and draw to it in the constructor, when the form starts up. I should say at the start that this is not actually the best or the correct way to draw to the screen – we'll quickly find that this example has a problem in that it is unable to redraw anything when it needs to after starting up. However the sample will illustrate quite a few points about drawing without our having to do very much work.

For this sample, we start Visual Studio .NET and create a Windows application. We first set the background color of the form to white. We've put this line in the `InitializeComponent()` method so that Visual Studio .NET recognizes the line and is able to alter the design view appearance of the form. We 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:

```
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Display At Startup";

   this.BackColor = Color.White;
```

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

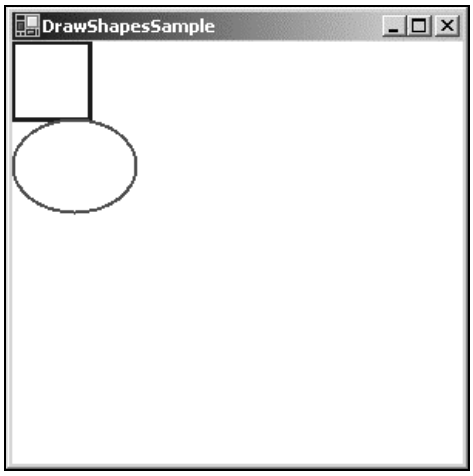
```
public Form1()
{
   InitializeComponent();

   Graphics dc = this.CreateGraphics();
   this.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);
}
```

As you can see, we then call the `Show()` method to display the window. This is really a fudge to force the window to display immediately, because we can't actually do any drawing until the window has been displayed – there's nothing to draw onto.

Finally, we 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) means x pixels to the right and y pixels down from the top left corner of the client area of the window – and these are the coordinates of the top left corner of the shape being displayed:

The overloads that we 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. Our first pen says that lines should be blue and with a width of 3 pixels, the second says that 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 hand corner of the rectangle, and its width and height. For the ellipse these numbers represent the same thing, except that we are talking about a hypothetical rectangle that the ellipse just fits into, rather than the ellipse itself.

Running this code gives this result:



I know – the book's printed in grayscale. As with all the screenshots in this chapter, you'll just have to take my word for it that the colors are correct. Or you can always try running the examples yourself!

This screenshot demonstrates a couple of points. First, you can see clearly what the client area of the window means. It's the white area – the area that has been affected by our setting the `BackColor` property. And notice that the rectangle nestles up in the corner of this area, as you'd expect when we specified coordinates of (0,0) for it. Second, notice how the top of the ellipse overlaps the rectangle slightly, which you wouldn't expect from the coordinates we gave in the code. That results from where Windows places the lines that border the rectangle and ellipse. By default, Windows will try to center the line on where the border of the shape is – that's not always possible to do exactly, because the line has to be drawn on pixels (obviously), but 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 strictly speaking are next to each other will have their borders overlap by one pixel. We've 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 MSDN documentation, but for our purposes the default behavior is adequate.

Unfortunately, if you actually run the sample you'll notice the form behaves a bit strangely. It's fine if you just leave it there, and it's fine if you drag it around the screen with the mouse. Try minimizing it then restoring it, however, and our carefully drawn shapes just vanish! The same thing happens if you drag another window across the sample. If you drag another window across it so that it only obscures a portion of our shapes, then drag the other window away again, you'll find the temporarily obscured portion has disappeared and you're left with half an ellipse or half a rectangle!

So what's going on? The problem arises when part of a window gets hidden, because Windows usually immediately discards all the information concerning exactly what was being displayed there. It has to – otherwise the memory usage for storing screen data would be astronomical. A typical computer might be running with the video card set to display 1024 x 768 pixels, perhaps with 24-bit color mode. We'll cover what 24-bit color means later in the chapter, but for now I'll say that implies that each pixel on the screen occupies 3 bytes. That means 2.25MB to display the screen. However, it's not uncommon for a user to sit there working, with 10 or 20 minimized windows in the taskbar. Let's do a worst-case scenario: 20 windows, each of which would occupy the whole screen if it wasn't minimized. If Windows actually stored the visual information those windows contained, ready for when the user restored them, you'd be talking about 45MB! These days, a good graphics card might have 64MB of memory and be able to cope with that, but it's only a couple of years ago that 4MB was considered generous in a graphics card – and the excess would need to be stored in the computer's main memory. A lot of people still have old machines – for example, my backup computer that has a 4 MB graphics card. Clearly it wouldn't be practical for Windows to manage its user interface like that.

The moment any part of a window gets 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 gets hidden, your application will need to redraw it later.

That's the source of the problem for our sample application. We placed our drawing code in the `Form1` constructor, which is called just once when the application starts up, and you can't call the constructor again to redraw the shapes when required later on.

In Chapter 7, when we covered controls, we didn't need to know about any of that. This is because the standard controls are pretty sophisticated and they are able to redraw themselves correctly whenever Windows asks them to. That's one reason why when programming controls you don't need to worry about the actual drawing process at all. If we are taking responsibility for drawing to the screen in our application then we also need to make sure our application will respond correctly whenever Windows asks it to redraw all or part of its window. In the next section, we will modify our sample to do just that.

Painting Shapes Using OnPaint()

If the above explanation has made you worried that drawing your own user interface is going to be terribly complicated, don't 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 don't 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 we need to do is override `OnPaint()`to perform our painting.

Although we've chosen to work by overriding `OnPaint()`, it's equally possible to achieve the same results by simply adding our 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, since you can add a new event handler through the VS .NET properties window, saving yourself from typing some code. However, our approach, of overriding `OnPaint()`, is slightly more flexible in terms of letting us control when the call to the base class window processing occurs, and is the approach recommended in the documentation. We suggest you use this approach for consistency.

We'll create a new Windows Application called `DrawShapes` to do this. As before, we set the background color to white using the Properties Window. We'll also change the Form's text to '`DrawShapes sample`'. Then we 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);
}
```

Notice that `OnPaint()` is declared as `protected`, because it is normally used internally within the class, so there's 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 is a `Graphics` instance, already primed and optimized to paint the required portion of the window. This means that you don't have to call `CreateGraphics()` to get a DC in the `OnPaint()` method – you've already been provided with one. We'll look at the other additional property soon – it contains more detailed information about which area of the window actually needs repainting.

In our implementation of `OnPaint()`, we first get a reference to the `Graphics` object from `PaintEventArgs`, then we draw our shapes exactly as we did before. At the end we call the base class's `OnPaint()` method. This step is important. We've overridden `OnPaint()` to do our own painting, but it's 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.

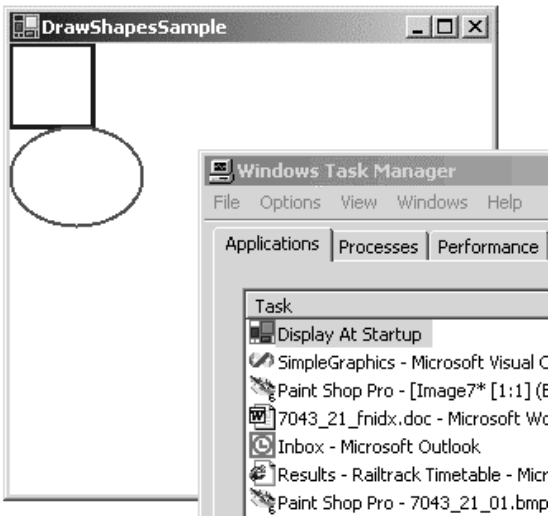
> *For this example, you'll find that removing the call to `base.OnPaint()` doesn't seem to have any effect, but don't ever be tempted to leave this call out. You might be stopping Windows from doing its work properly and the results could be unpredictable.*

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

Running this code gives the same results initially as for our previous example – except that now our application behaves itself properly when you minimize it or hide parts of the window.

Using the Clipping Region

Our `DrawShapes` sample from the last section illustrates the main principles involved with drawing to a window, although it's not very efficient. The reason is that it attempts to draw everything in the window, irrespective of how much needs to be drawn. Consider the situation shown in this screenshot. I ran the `DrawShapes` example, but while it was on the screen I opened another window and moved it over the `DrawShapes` form, so it hid part of it.



So far, so good. However, when I 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's actually nothing that needs to be done in this case apart from repaint the white background area. However, Windows doesn't know that, so it thinks it should raise the `Paint` event, resulting in our `OnPaint()` implementation being called. `OnPaint()` will then unnecessarily attempt to redraw the rectangle and ellipse.

Actually, in this case, the shapes will not get repainted. The reason is to do with the device context. Windows has pre-initialized the device context with information concerning what area actually needed repainting. In the days of GDI, the region that is marked for repainting used to be known as the **invalidated region**, but with GDI+ the terminology has largely changed to **clipping region**. The device context knows what this region is; 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's still a potential performance hit here. We don't 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, since 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 actively help the device context out by carrying out a few simple checks, to see if the proposed drawing work is likely to be actually needed, before it calls the relevant `Graphics` instance methods. In this section we're going to code up a new example – `DrawShapesWithClipping` – by modifying the `DisplayShapes` example to do just that. In our `OnPaint()` code, we'll do a simple test to see whether the invalidated region intersects the area we need to draw in, and only call the drawing methods if it does.

First, we need to obtain the details of the clipping region. This is where an extra property, `ClipRectangle`, on the `PaintEventArgs` comes in. `ClipRectangle` contains the coordinates of the region to be repainted, wrapped up in an instance of a struct, `System.Drawing.Rectangle`. `Rectangle` 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, we need to decide what test we'll use to determine whether drawing should take place. We'll go for a simple test here. Notice, that in our 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, point (82,132) to be on the safe side, since we know that the lines may stray a pixel or so outside this area. So we'll check whether the top left corner of the clipping region is inside this rectangle. If it is, we'll go ahead and redraw. If it isn't, we won't bother.

Here 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);
   }
}
```

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

Measuring Coordinates and Areas

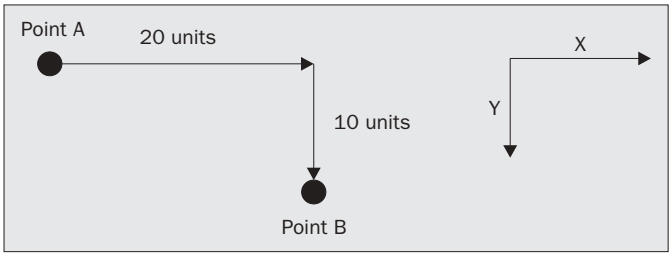
In our last example, we encountered the base struct, `Rectangle`, which is used to represent the coordinates of a rectangle. GDI+ actually uses several similar structures to represent coordinates or areas, and we're at a convenient point in the chapter to go over the main ones. We'll look at the following structs, which are all defined in the `System.Drawing` namespace:

Struct	Main Public Properties
`struct Point` `struct PointF`	`X, Y`
`struct Size` `struct SizeF`	`Width, Height`
`struct Rectangle` `struct RectangleF`	`Left`, `Right`, `Top`, `Bottom`, `Width`, `Height`, `X`, `Y`, `Location`, `Size`

Note that many of these objects have a number of other properties, methods, or operator overloads not listed here. In this section we'll just discuss the most important ones.

Point and PointF

We'll look at `Point` first. `Point` is conceptually the simplest of these structs. Mathematically, it's completely equivalent to a 2D vector. It contains two public integer properties, which represent how far you move horizontally and vertically from a particular location (perhaps on the screen). In other words, look at this diagram:



In order 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 as this is how they are commonly referred to. We could create a `Point` struct that represents that as follows:

```
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 you can also set the values in a `Point` like 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 explicitly copy the values across, or use one of three conversion methods, `Round()`, `Truncate()`, and `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 may be wondering what a "unit" is measured in. By default, GDI+ will interpret units as pixels along the screen (or printer, whatever the graphics device is) – so that's how the `Graphics` object methods will view any coordinates that they get passed as parameters. For example, the point `new Point(20,10)` represents 20 pixels across the screen and 10 pixels down. Usually these pixels will be measured from the top left corner of the client area of the window, as has been the case in our examples up to now. However, that won't always be the case – for example, on some occasions you may wish 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 you're talking pixels relative to the top left corner of the client area.

We'll have more to say on this subject later on, after we've examined scrolling, when we mention the three different coordinate systems in use, world, page, and device coordinates.

Size and SizeF

Like `Point` and `PointF`, sizes come in two varieties. The `Size` struct is for when you are using `int`s; `SizeF` is available if you need to use `float`s. Otherwise `Size` and `SizeF` are identical. We'll focus on the `Size` struct here.

In many ways the `Size` struct is identical to the `Point` struct. It has two integer properties that represent a distance horizontally and a distance vertically – the main difference is that instead of `X` and `Y`, these properties are named `Width` and `Height`. We can represent our earlier diagram by:

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

Although strictly speaking, a `Size` mathematically represents exactly the same thing as a `Point`; conceptually it is intended to be used in a slightly different way. A `Point` is used when we are talking about where something is, and a `Size` is used when we are talking about how big it is. However, because `Size` and `Point` are so closely related, there are even supported explicit 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 we 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 that would be represented by a `Point` instance. To see the difference, suppose we drew the rectangle in a different location, so it's top left coordinate was 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 – that's still (50,50).

The addition operator has been overloaded for `Point`s and `Size`s, so that it is possible to add a `Size` to a `Point` giving another `Point`:

```
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);
}
```

This code, running as a simple console application, called `PointsAndSizes`, produces this output:

Notice 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` to give a `Point`, and you can add two `Sizes` together, giving another `Size`. It is not possible, however, to add a `Point` to another `Point`. Microsoft decided that adding `Point`s doesn't conceptually make sense, and so chose not to supply any overload to the + operator that would have allowed that.

You can also explicitly cast a `Point` to a `Size` 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 of `topLeft.X`, and `s1.Height` is assigned the value of `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`, we'll only consider the `Rectangle` struct here. `RectangleF` is basically identical except that those of its properties that represent dimensions all use `float`, whereas those of `Rectangle` use `int`.

A `Rectangle` can be thought of as composed of a point, representing the top left corner of the rectangle, and a `Size`, which represents how large it is. One of its constructors actually takes a `Point` and a `Size` as its parameters. We can see this by rewriting our 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` 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)
```

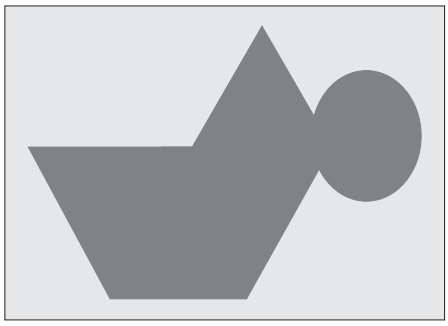
`Rectangle` makes quite a few read-write properties available to set or extract its dimensions in different combinations:

Property	Description
`int Left`	x-coordinate of left-hand edge
`int Right`	x-coordinate of right-hand edge
`int Top`	y-coordinate of top
`int Bottom`	y-coordinate of bottom
`int X`	same as `Left`
`int Y`	same as `Top`
`int Width`	width of rectangle
`int Height`	height of rectangle
`Point Location`	top left corner
`Size Size`	size of rectangle

Note that these properties are not all independent – for example setting `Width` will also affect the value of `Right`.

Region

We'll mention the existence of the `System.Drawing.Region` class here, though we don't have space to go details in this book. `Region` represents an area of the screen that has some complex shape. For example the shaded area in the diagram 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 round the edge of the region. If you do need to start working with areas like this, then it's worth looking up the `Region` class.

A Note about Debugging

We're just about ready to do some more advanced drawing now. First,however, I just want to say a few things about debugging. If you have a go at setting break points in the examples in this chapter you will quickly notice that debugging drawing routines isn't quite as simple as debugging other parts of your program. This is because entering and leaving the debugger often causes `Paint` messages to be sent to your application. The result can be that setting a breakpoint in your `OnPaint()` override simply causes your application to keep painting itself over and over again, so it's 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 in `OnPaint()`. As expected, the application hits the break point and the debugger comes in, at which point your developer environment MDI window comes to the foreground. If you're anything like me, you probably have the developer environments set to full screen display so 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 find out something useful. Then you hit *F5* to tell the application to continue, so that you can go on to see what happens when the application displays something else, after it's done 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 gets hit again straight away. If that's what you want fine, but more commonly what you really want is to hit the breakpoint *later,* when the application is drawing something more interesting, perhaps after you've selected some menu option to read in a file or in some other way changed what gets displayed. It looks like you're stuck. Either you don't have a break point in `OnPaint()` at all, or your application can never get beyond the point where it's displaying its initial startup window.

There are a couple of ways around this problem.

If you have a big screen the easiest way is simply to keep your developer environment window tiled rather than maximized and keep it well away from your application window – so your application never gets hidden in the first place. Unfortunately, in most cases that is not a practical solution, because that would make your developer environment window too small. 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 .NET.

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 quite 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()` for 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 some condition, which will only be `true` in the specified circumstances – and then place the break point inside the `if` block, like this:

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

This is effectively a quick-and-easy way of putting in a conditional break point.

Drawing Scrollable Windows

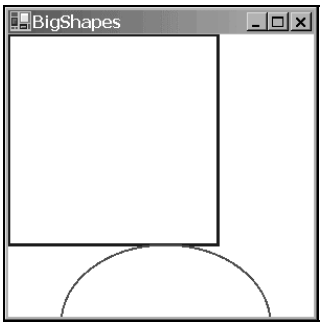
Our earlier `DrawShapes` sample worked very well, because everything we needed to draw fitted into the initial window size. In this section we're going to look at what we need to do if that's not the case.

We shall expand our `DrawShapes` sample to demonstrate scrolling. To make things a bit more realistic, we'll start by creating an example, `BigShapes`, in which we will make the rectangle and ellipse a bit bigger. Also, while we're at it we'll demonstrate 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 Point rectangleTopLeft = new Point(0, 0);
private Size rectangleSize = new Size(200,200);
private Point ellipseTopLeft = new Point(50, 200);
private Size ellipseSize = new Size(200, 150);
private Pen bluePen = new Pen(Color.Blue, 3);
private 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);
   }
}
```

Notice, that we've also turned the `Pen`, `Size`, and `Point` objects into member fields – this is more efficient than creating a new `Pen` every time we need to draw anything, as we have been doing up to now.

The result of running this example looks like this:

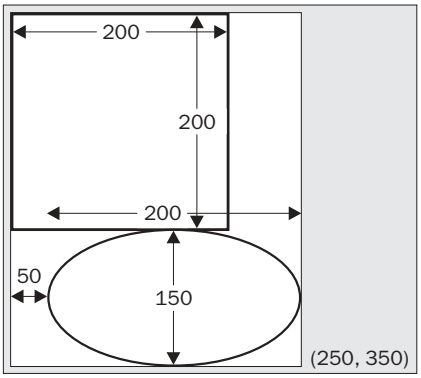


We can see a problem instantly. The shapes don't fit in our 300x300 pixel drawing area.

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, with the kind of user interface that we were dealing with in Chapter 7, we'd let the .NET runtime and the base classes handle everything. 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 become necessary. The `Form` instance will also automatically add the scrollbars for you, and not only that, but it's also able to correctly draw whichever portion of the screen you've scrolled to. In that case there is nothing you need to explicitly do in your code. In this chapter, however, we're taking responsibility for drawing to the screen; therefore, we're going to have to help the `Form` instance out when it comes to scrolling.

> *In the last paragraph we said "if a document is too large to display". This probably made you think in terms of something like a Word or Excel document. With drawing applications, however, it's better to think of the document as whatever data the application is manipulating that it needs to draw. For our current example, the rectangle and ellipse between them constitute the document.*

Getting the scrollbars added is actually very easy. The `Form` can still handle all that for us – the reason it hasn't in the above `ScrollShapes` sample is that Windows doesn't know they are needed – because it doesn't know how big an area we will want to draw in. How big an area is that? More accurately, what we need to figure out is 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 we've done any scrolling), and which is just big enough to contain the entire document. In this chapter, we'll refer to this area as the document area. Looking at the diagram of the 'document' we can see that for this example the document area is (250, 350) pixels.



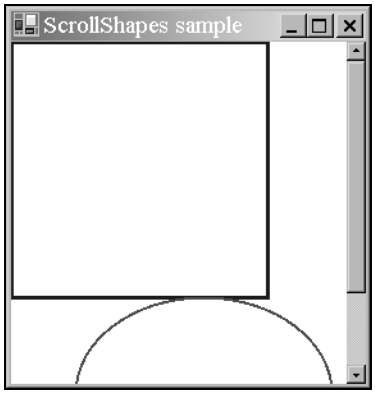
It is easy to tell the form how big the document is. We use the relevant property, `Form.AutoScrollMinSize`. Therefore we can add this code to either the `InitializeComponent()` method or the `Form1` constructor:

```
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Scroll Shapes";
   this.BackColor = Color.White;
   this.AutoScrollMinSize = new Size(250, 350);
}
```

Alternatively the `AutoScrollMinSize` property can be set through the Visual Studio .NET properties window.

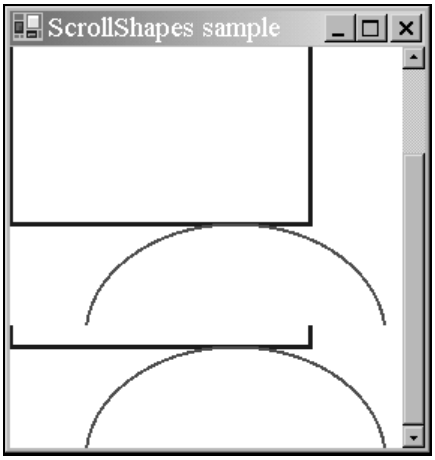
Setting the minimum size at application startup and leaving it thereafter is fine in this particular application, because we know that is how big the screen area will always be. Our "document" never changes size while this particular application is running. Bear in mind, however, that if your application 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'll have to sort out the code manually – the Visual Studio .NET Properties window can only help you with the initial value that a property has when the form is constructed).

Setting `AutoScrollMinSize` is a start, but it's not yet quite enough. To see that, let's look at what our sample – which in this version is downloadable as the `ScrollShapes` sample – looks like now. Initially we get the screen that correctly displays the shapes:



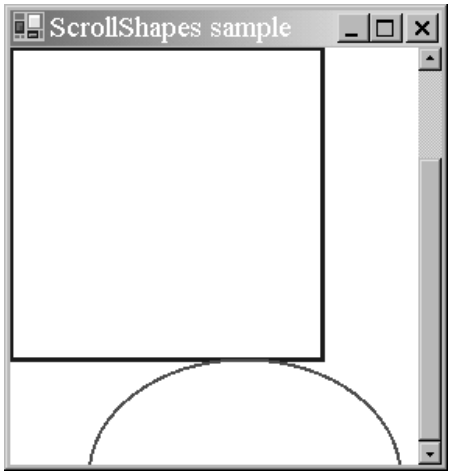
Notice that, not only has the form correctly set the scrollbars, but it's even correctly sized them to indicate what proportion of the document is currently displayed. You can try resizing the window while the sample is running – you'll find the scrollbars respond correctly, and even disappear if we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

However, now look at what happens if we actually use one of the scrollbars and scroll down a bit:



Clearly something has gone wrong!

In fact, what's gone wrong is that we haven't taken into account the position of the scrollbars in the code in our `OnPaint()` override. We can see this very clearly if we force the window to completely repaint itself by minimizing and restoring it. The result looks like this:



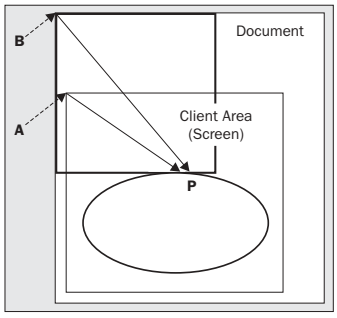
The shapes have been painted, just as before, with the top left corner of the rectangle nestled into the top left corner of the client area – just as if we hadn't moved the scrollbars at all.

Before we go over how to correct this problem, we'll take a closer look at precisely what is happening in these screenshots. Doing so is quite instructive, both because it'll help us to understand exactly how the drawing is done in the presence of scrollbars and because it'll be quite good practice. If you start using GDI+, I promise you that sooner or later, you'll find yourself presented with a strange drawing like one of those above, and having to try to figure out what has gone wrong.

We'll look at the last screenshot first since that one is easy to deal with. The `ScrollShapes` sample has just been restored so the entire window has just been repainted. Looking back at our code 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 – it doesn't know anything about the scrollbars. Our code as yet does not attempt to adjust the coordinates for the scrollbar positions. The same goes for the ellipse.

Now, we can tackle the earlier screenshot, from immediately after we'd scrolled down. We notice that here the top two-thirds or so of the window look fine. That's because these were drawn when the application first started up. When you scroll windows, Windows doesn't ask the application to redraw what was already on the screen. Windows is smart enough to figure out for itself which bits of what's currently being displayed on the screen can be smoothly moved around to match where the scrollbars now are. That's a much more efficient process, since it may be able to use some hardware acceleration to do that too. The bit in this screenshot that's wrong is the bottom third of the window. This part of the window didn't get drawn when the application first appeared, since before we started scrolling it was outside the client area. This means that Windows asks our `ScrollShapes` application to draw this area. It'll raise a `Paint` event passing in just this area as the clipping rectangle. And that's exactly what our `OnPaint()` override has done.

One way of looking at the problem is that we are at the moment expressing our coordinates relative to the top left corner of the start of the 'document' – we need to convert them to express them relative to the top left corner of the client area instead. The following diagram should make this clear:



To make the diagram clearer we've actually extended the document further downwards and to the right, beyond the boundaries of the screen, but this doesn't change our reasoning. We've also assumed a small horizontal scroll as well as a vertical one.

In the diagram the thin rectangles mark the borders of the screen area and of the entire document. The thick lines mark the rectangle and ellipse that we are trying to draw. P marks some arbitrary point that we are drawing, which we're going to take as an example. When calling the drawing methods we've supplied the graphics instance with the vector from point B to (say) point P, expressed as a `Point` instance. We actually need to give it the vector from point A to point P.

The problem is that we don't know what the vector from A to P is. We know what B to P is – that's just the coordinates of P relative to the top left corner of the document – the position where we want to draw point P in the document. We also know what the vector from B to A is – that's just the amount we've scrolled by; this is stored in a property of the `Form` class called `AutoScrollPosition`. However, we don't know the vector from A to P.

Now, if you were good at math at school, you might remember what the solution to this is – you just have to subtract vectors. Say, for example, to get from B to P you move 150 pixels across and 200 pixels down, while to get from B to A you have to 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. The `Graphics` class actually implements a method that will do these calculations for us. It's 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, (our `AutoScrollPosition` property, that is the vector from B to A in the diagram). Then the `Graphics` device will from then on work out all its coordinates taking into account where the client area is relative to the document.

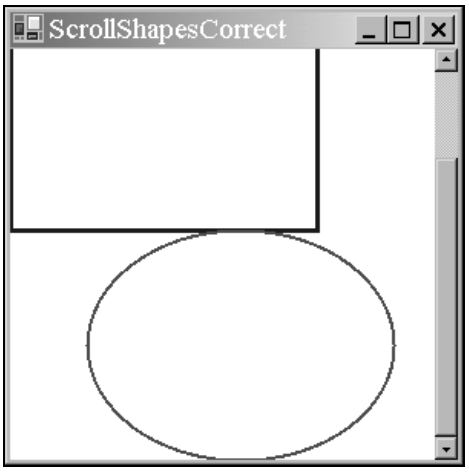
After all that explanation, all we need to do is add this line to our drawing code:

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

In fact in our example, it's a little more complicated because we are also separately testing whether we need to do any drawing by looking at the clipping region. We need to adjust this test to take the scroll position into account too. When we've done that, the full drawing code for the sample (downloadable from the Wrox Press web site as `ScrollShapes`) 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 we have our scroll code working perfectly, we can at last obtain a correctly scrolled screenshot!



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** are the position of a point measured in pixels from the top left corner of the document
- **Page coordinates** are 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 used to be known as device coordinates. Those developers should also note that the way you code up 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's 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 we do not use pixels as the unit of measurement – instead we 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 we won't use the `PageUnit` property in this chapter, it can be useful as a way of getting around the different pixel densities of devices. For example, 100 pixels on most monitors will occupy something like an inch. However, laser printers can have anything up to thousands of dpi (dots per inch) – which means that a shape specified to be 100 pixels wide will look a lot smaller when printed on it. 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.

Colors

In this section, we're going to look at 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, once you've instantiated this struct, you won't do much with the corresponding `Color` instance – just pass it to whatever other method you are calling that requires a `Color`. We've encountered this struct before – when we set the background color of the client area of the window in each of our samples, as well as when we set the colors of the various shapes we were displaying. The `Form.BackColor` property actually returns a `Color` instance. In this section, we'll look at this struct in more detail. In particular, we'll examine several different ways that you can construct a `Color`.

Red-Green-Blue (RGB) Values

The total number of colors that can be displayed by a monitor is huge – over 16 million. To be exact the number is 2 to the power 24, which works out at 16,777,216. Obviously we need some way of indexing those colors so we can indicate which of these is the color we want to display at a 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 principle that any color that the human eye can distinguish can be constructed from a certain amount of red light, a certain amount of the green light, and a certain amount of blue light. These colors are known as **components**. In practice, it's found that if we divide the amount of each component light into 256 possible intensities, then that gives a sufficiently fine gradation to be able to display images that are perceived by the human eye to be of photographic quality. We 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.

> *We can now see where are quoted figure of 16,777,216 colors comes from, since that number is just 256 cubed.*

This gives us our 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. There are a number of other overloads to this function, some of which also allow you to specify something called an alpha-blend (that's 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, since 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's a lot easier to just be able 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 is one of these properties that we used when we set the background color of our windows to white in our samples:

```
            this.BackColor = Color.White;

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

There are several hundred such colors. The full list is given in the MSDN 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.

> *Incidentally, although it might look that way, these named colors have not been chosen at random. Each one represents a precise set of RGB values, and they were originally chosen many years ago for use on the Internet. The idea was to provide a useful set of colors right 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 couldn't 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 since modern web browsers are quite capable of displaying any RGB value correctly.*

Graphics Display Modes and the Safety Palette

Although we've said that in principle monitors can display any of the over 16 million RGB colors, in practice this depends on how you've set the display properties on your computer. In Windows, there are traditionally three main color options (although some machines may 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 for reasons 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 of the RGB colors simultaneously. This sounds the best option, but it comes at a cost: 3 bytes are needed to hold a full RGB value which means 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's less common now than it used to be) you may choose one of the other modes. High color mode gives you 2 bytes per pixel. That's enough to give 5 bits for each RGB component. So instead of 256 gradations of red intensity you just get 32 gradations; the same for blue and green, which gives a total of 65,536 colors. That is just about enough to give apparent photographic quality on a casual inspection, though areas of subtle shading tend to be broken up a bit.

256-color mode gives you even fewer colors. However, in this mode, you get to choose which colors. What happens is that the system sets up something known as a **palette**. This is a list of 256 colors chosen from the 16 million RGB colors. Once you've 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 still only display 256 different colors on the screen at any one time. 256-color mode is only really used when high performance and video memory is at a premium. Most games will use this mode – and 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 it is asked to display a particular RGB color, it will pick the nearest mathematical match from the pool of colors that it is able to display. It's for this reason that it's 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, they may not see the image the same way you intended it. So if you're doing that kind of work with GDI+, you should test your application in different color modes. (It is also possible for your application to programmatically set a given color mode, though we won't go into that in this chapter.)

The Safety Palette

For reference, we'll quickly mention the safety palette, which is a very commonly-used default palette. The way it works is that we set six equally spaced possible values for each color component. Namely, the values 0, 51, 102, 153, 204, and 255. In other words, the red component can have any of these values. So can the green component. So can the blue component. So 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 us a total of 6 cubed = 216 colors. The idea is that this gives us an easy way of having a palette that contains colors from right across the spectrum and of all degrees of brightness, although in practice this doesn't actually work that well because equal mathematical spacing of color components doesn't mean equal perception of color differences by the human eye. Because the safety palette used to be widely used, however, you'll still find a fair number of applications and images exclusively use colors from the safety palette.

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

Pens and Brushes

In this section, we'll review two helper classes that are needed in order to draw shapes. We've already encountered the `Pen` class, used to tell the graphics instance how to draw lines. A related class is `System.Drawing.Brush`, which tells it how to fill regions. For example, the `Pen` is needed to draw the outlines of the rectangle and ellipse in our previous samples. If we'd needed to draw these shapes as solid, it would have been a brush that would have been used to specify how to fill them in. 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`.

We will look at brushes first, then pens.

> *Incidentally, if you've programmed using GDI before you have noticed from the first couple of 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+, as mentioned earlier, Microsoft has instead gone 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 brush – more than we have space to go into in this chapter, so we'll just explain 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`, simply indicates that a region is to be filled with 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 Internet named colors you can construct the brush more simply using another class, `System.Drawing.Brushes`. `Brushes` is one of those classes that you never actually instantiate (it's got a private constructor to stop you doing that). It simply has a large number of static properties, each of which returns a brush of a specified color. You'd 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's in the `Drawing2D` namespace, represented by the class `System.Drawing.Drawing2D.HatchBrush`. The `Brushes` class can't help you with hatch brushes – you'll need to construct one explicitly, by supplying the hatch style and two colors – the foreground color followed by the background color (you can omit the background color, in which case it defaults to black). The hatch style comes from an enumeration, `System.Drawing.Drawing2D.HatchStyle`. There are a large number of `HatchStyle` values available, so it's easiest to refer to the MSDN 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:

```
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 under GDI. GDI+ has added a couple of new styles of brush:

- `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

We won't go into these brushes in this chapter. We'll note though that both can give some spectacular effects if used carefully.

Pens

Unlike brushes, pens are represented by just one class – `System.Drawing.Pen`. The pen is, however, actually 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 that we mentioned earlier, which 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 solid color, or it can be filled using a brush. Hence, a `Pen` instance may contain a reference to a `Brush` instance. This is quite powerful, as it means you can draw lines that are colored in by using – say – hatching or linear shading. There are four different ways that you can construct a `Pen` instance that you have designed yourself. You can do it by passing a color, or you can do it by passing in a brush. Both of these constructors will produce a pen with a width of one pixel. Alternatively, you can 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 we are using non-default units such as millimeters or inches for the `Graphics` object that will do the drawing – so we can for example specify fractions of an inch.) So for example, you can construct pens like 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, for the quick construction of pens, you can use the class `System.Drawing.Pens` which, like the `Brushes` class, simply contains a number of stock pens. These pens all have width one pixel and come in the usual sets of Internet named colors. This allows you to construct pens in this way:

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

Drawing Shapes and Lines

We've almost finished the first part of the chapter, in which we've covered all the basic classes and objects required in order to draw specified shapes and so on to the screen. We'll round off by reviewing some of the drawing methods the `Graphics` class makes available, and presenting 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. Once again there are too many to provide a comprehensive list here, but the following table gives the main ones and should give you some idea of the variety of shapes you can draw.

Method	Typical parameters	What it draws
`DrawLine`	Pen, start and end points	A single straight line
`DrawRectangle`	Pen, position, and size	Outline of a rectangle
`DrawEllipse`	Pen, position, and size	Outline of an ellipse
`FillRectangle`	Brush, position, and size	Solid rectangle
`FillEllipse`	Brush, position, and size	Solid ellipse
`DrawLines`	Pen, array of points	Series of lines, connecting each point to the next one in the array
`DrawBezier`	Pen, 4 points	A smooth curve through the two end points, with the remaining two points used to control the shape of the curve
`DrawCurve`	Pen, array of points	A smooth curve through the points
`DrawArc`	Pen, rectangle, two angles	Portion of circle within the rectangle defined by the angles
`DrawClosedCurve`	Pen, array of points	Like `DrawCurve` but also draws a straight line to close the curve
`DrawPie`	Pen, rectangle, two angles	Wedge-shaped outline within the rectangle
`FillPie`	Brush, rectangle, two angles	Solid wedge-shaped area within the rectangle
`DrawPolygon`	Pen, array of points	Like `DrawLines` but also connects first and last points to close the figure drawn

Before we leave the subject of drawing simple objects, we'll round off with a simple example that demonstrates the kinds of visual effect you can achieve by use of brushes. The example is called `ScrollMoreShapes`, and it's essentially a revision of `ScrollShapes`. Besides the rectangle and ellipse, we'll add a thick line and fill the shapes in with various custom brushes. We've already explained the principles of drawing so we'll present the code without too many comments. First, because of our new brushes, we need to indicate we are using the `System.Drawing.Drawing2D` namespace:

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

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

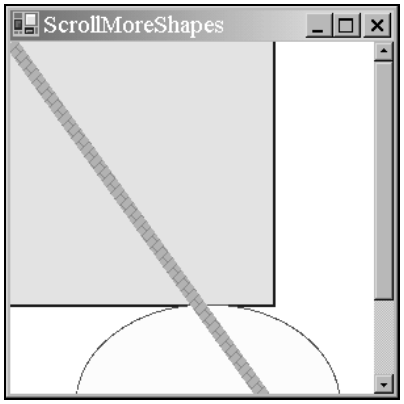
```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
                                                  new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
                                                new Size(200,150));
private Pen bluePen = new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

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

Here is the `OnPaint()` override:

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

As before we also set the `AutoScrollMinSize` to (250,350). Now the 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.

Displaying Images

One of the most common things you may 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. It's also possible to perform some manipulations on the image, such as stretching it or rotating it, and you can choose to display only a portion of it.

In this section, just for a change, we'll present the sample first. Then we'll discuss some of the issues you need to be aware of when displaying images. We can do this, because the code needed to display an image really is so simple.

The class we need is the .NET base class, `System.Drawing.Image`. An instance of `Image` represents one image – if you like, one picture. Reading in an image 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 you have a suitable `Graphics` instance to hand – a simple call to either `Graphics.DrawImageUnscaled()` or `Graphics.DrawImage()` will suffice. 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. But we will use `DrawImage()`, like this:

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

In this line of code, `dc` is assumed to be a `Graphics` instance, while `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 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 with GDI+. In GDI, displaying an image involved several nontrivial steps. If the image was a bitmap, loading it was reasonably simple, but if it was any other file type, loading it would involve a sequence of calls to OLE objects. Actually getting a loaded image onto the screen involved getting a handle to it, selecting it into a memory device context, 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.*

We'll illustrate the process of displaying an image 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 you'll need to change it to reflect the location of the file in your system). The `.jpg` file we'll display is a group photograph of attendees from a recent COMFest event.

COMFest (www.comfest.co.uk) is an informal group of developers in the UK who meet to discuss the latest technologies and swap ideas. The picture includes all the attendees at COMFest 4, except for the author of this chapter who was (conveniently) taking the picture!

As usual for this chapter, the `DisplayImage` project is a standard C# Visual Studio .NET-generated Windows application. We add the following fields to our `Form1` class:

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

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

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

   piccy =
      Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\CF4Group.bmp");
   this.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
}
```

Note that the size in pixels of the image is obtained as its `Size` property, which we use to set the document area. We also set up the `piccyBounds` array, which is used to identify the position of the image on the screen. We have chosen the coordinates of the three corners to draw the image in its actual size and shape here, but if we'd wanted the image to be resized, stretched, or even sheared into a non-rectangular parallelogram, we could do so simply by changing the values of the `Point`s 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(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
   dc.DrawImage(piccy, piccyBounds);
}
```

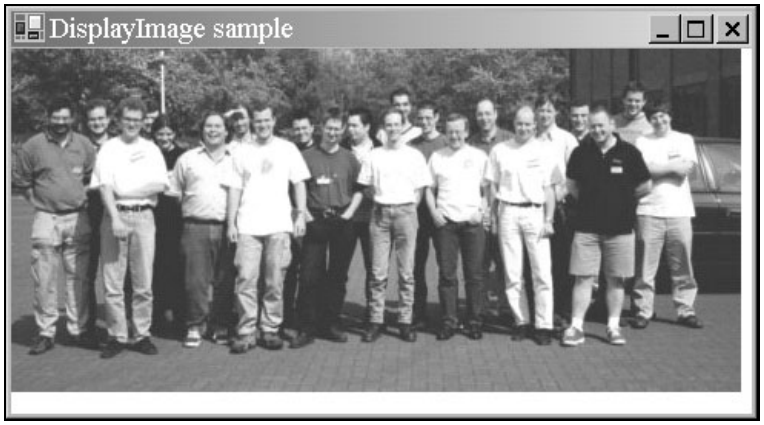
Finally, we'll take particular note of the modification made to the code wizard-generated `Form1.Dispose()` method:

```
        protected override void Dispose(bool disposing)
        {
           piccy.Dispose();
```

```
        if( disposing )
        {
           if (components != null)
           {
              components.Dispose();
           }
        }
        base.Dispose( disposing );
     }
```

Disposing of the image as soon as possible when it's no longer needed is important, because images generally eat 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 can no longer be displayed (unless you load a new image).

Running this code produces these results:



Issues When Manipulating Images

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

The most important point to understand about images is that they are always rectangular. That's not just a convenience, but because of the underlying technology. It's because all modern graphics cards have hardware built-in that can very 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 individually pixel by pixel, you'd see the image gradually being drawn over a period of up to several seconds.

`BitBlt`s 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 `BitBlt`ing portions of images 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, though 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. We're not going to go into details of this here. We'll remark however, 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` to be used in the drawing process. `DrawImage()` also allows you to draw (`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've left the very important topic of displaying text until this late in the 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're not that bothered 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, you rapidly find that things become a lot more complex. This is for two reasons:

- If you're concerned about getting the appearance just right, you need to understand fonts. Where shape drawing requires brushes and pens as helper objects, the process of drawing text correspondingly requires fonts as helper objects. And understanding fonts is not trivial task.
- Text needs to be very carefully laid out in the window. Users generally expect words to follow naturally from one another – to be lined up with clear spaces in between. Doing that is harder than you'd think. For a start, unlike the case for shapes, you don't 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). Also, how much space on the screen a word occupies will affect whereabouts on the screen every subsequent word in the document gets placed. If your application does line wrapping then it'll need to carefully assess word sizes before deciding where to place the break. The next time you run Word for Windows, look carefully at the way Word is continually repositioning text as you do your work: there's a lot of complex processing going on there. The chances are that any GDI+ application you work on won't be anything like as complex as Word, but if you need to display any text then many of the same considerations still apply.

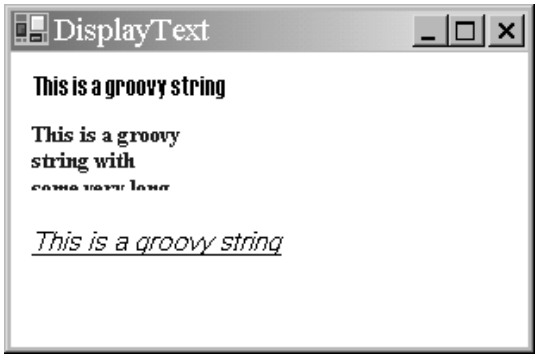
So, good quality text processing is tricky to get right, but putting a line of text on the screen, assuming you know the font and where you want it to go, is actually very simple. Therefore, the next thing we'll do is present a quick example that shows how to display a couple of pieces of text. After that, the plan for the rest of the chapter is to review some of the principles of fonts and font families before moving on to our more realistic text-processing example, the `CapsEditor` sample, which will demonstrate some of the issues involved when you're trying to control text layout on-screen, and will also show how to handle user input.

Simple Text Example

This example, `DisplayText`, is our usual Windows Forms effort. This time we've overridden `OnPaint()` and added member fields as follows:

```
private System.ComponentModel.Container components = null;
private Brush blackBrush = Brushes.Black;
private Brush blueBrush = Brushes.Blue;
private Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private Font boldTimesFont = new Font("Times New Roman", 10, FontStyle.Bold);
private 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));
}
```

Running this example produces this:



The example demonstrates the use of the `Graphics.DrawString()` method to draw items of text. `DrawString()` comes in a number of overloads, of which we demonstrate three. The different overloads all, however, 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. There are a couple of alternatives 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`, 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 the string out inside that rectangle. If the text doesn't fit into the bounds of the rectangle, then it'll be cut off, as you see from the screenshot. Passing a rectangle to `DrawString()` means that the drawing process will take longer, as `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 couple of ways of constructing fonts. You always need 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).

Fonts and Font Families

We all think intuitively that we have a fairly good understanding of fonts; after all we look at them almost all the time. 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.

Oddly, our intuitive understanding usually isn't quite correct. Most people, if asked to name a font, will say things like 'Arial' or 'Times New Roman' or 'Courier'. 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 will have become used to recognizing the styles of the most common font families, even if we're not consciously aware of this.

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 it is **bold,** *italic,* underlined, or displayed in SMALL CAPS or as a $_{\text{subscript}}$; this is technically referred to the **style**, though in some ways the term is misleading since the visual appearance is determined as much by the font family.

The way the size of the text is measured is by specifying its height. The height is measured in **points** – a traditional unit, which represents 1/72 of an inch (0.351 mm). So letters in a 10-point font are roughly 1/7" or 3.5 mm high. However, you won't get seven lines of 10 point text into one inch of vertical screen or paper space, because you need to allow for the spacing between the lines as well.

> *Strictly speaking, measuring the height isn't quite as simple as that, since there are several different heights that you need to consider. For example, there is the height of tall letters like the A or F (this is the measurement that we really mean when we talk about the height), the additional height occupied by any accents on letters like Å or Ñ (the **internal leading**), and the extra height below the base line needed for the tails of letters like y and g (the **descent**). However, for this chapter we won't worry about that. Once you specify the font family and the main height, these subsidiary heights are determined automatically – you can't independently choose their values.*

When you're dealing with fonts you may also encounter some other terms that are commonly used to describe certain font families.

- A **serif** font family is one that has little tick marks 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, don't have these ticks. Good examples of sans serif fonts are Arial, and Verdana. The lack of tick marks often gives text a blunt, in-your-face appearance, so sans serif fonts are often used for important text.
- A **True Type** 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 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 will use are true type 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 we need to deal with when selecting or manipulating fonts. These are:

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

We have already seen the main use of the `Font` class. When we wish to draw text we 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 (surprisingly enough) to represent a family of fonts.

One use of the `FontFamily` class is if you know you want a font of a particular type (Serif, Sans Serif or Monospace), but don't mind which font. The static properties `GenericSerif`, `GenericSansSerif`, and `GenericMonospace` return default fonts that satisfy these criteria:

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

Generally speaking, however, if you're writing a professional application, you will want to choose your font in a more sophisticated way than this. 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. And if you want your application to be very user-friendly, the first choice on the list will probably be the one that the user selected last time they ran your software. Usually, if you're dealing with the most popular font families, such as Arial and Times New Roman, you'll be safe. However, if you do try to display text using a font that doesn't exist the results aren't always predictable and you're quite likely to find that Windows just substitutes the standard system font, which is very easy for the system to draw but it doesn't look very pleasant – and if it does appear in your document it's likely to give the impression of very poor-quality software.

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

}
```

Example: Enumerating Font Families

In this section, we will 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 10-point regular version of that font family). When the sample is run it will look something like this:

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

For this sample we have as usual created a standard C# Windows Application – this time named `EnumFontFamilies`. We start off by adding an extra namespace to be searched. We will be using the `InstalledFontCollection` class, which is defined in `System.Drawing.Text`.

```
using System;
using System.Drawing;
using System.Drawing.Text;
```

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

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

`margin` will be 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 cases doesn't do things the way you ought to in a real application. For example, I've just hard-coded in a guessed value for the document size of (200,1500) and set the `AutoScrollMinSize` property to this value using the Visual Studio .NET Properties window. Normally you would have to examine the text to be displayed to work out the document size. We will do that in the next section.

Here is the `OnPaint()` method:

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

In this code we start off by using an `InstalledFontCollection` object to obtain an array that contains details of all the available font families. For each family, we instantiate a 10 point `Font`. We use a simple constructor for `Font` – there are many more that allow additional options to be specified. The constructor we've picked takes two parameters, the name of the family and the size of the font:

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

This constructor constructs a font that has the regular style. To be on the safe side, however, we 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, and 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 may be combined with the bitwise `OR` operator. The possible flags are `Bold`, `Italic`, `Regular`, `Strikeout`, and `Underline`.

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

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

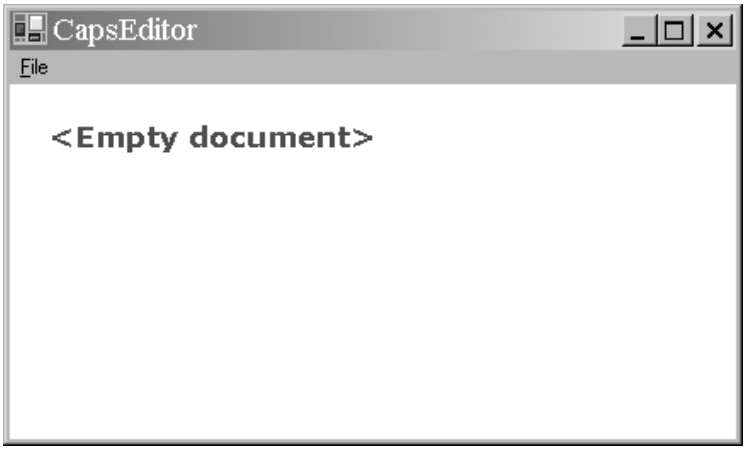
Again, to keep things simple, our version of `OnPaint()` reveals some bad programming practices. For a start, we haven't bothered to check what area of the document actually needs drawing – we just try to display everything. Also, instantiating a `Font` is, as remarked earlier, a computationally intensive process, so we really ought to save the fonts rather than instantiating new copies every time `OnPaint()` is called. As a result of the way the code has been designed, you may notice that this example actually takes a noticeable time to paint itself. In order to try to conserve memory and help the garbage collector out we do, however, call `Dispose()` on each font instance after we have finished with it. If we didn't, then after 10 or 20 paint operations, there'd be a lot of wasted memory storing fonts that are no longer needed.

Editing a Text Document: The CapsEditor Sample

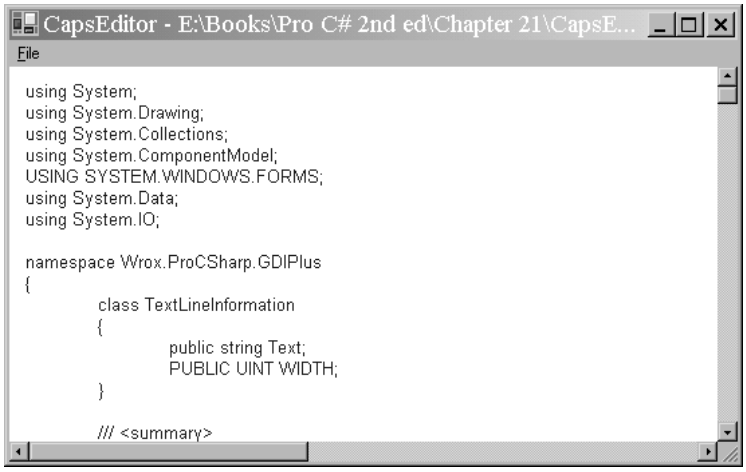
We now come to the extended example in this chapter. The `CapsEditor` example is designed to illustrate how the principles of drawing that we've learned up till now need to be applied in a more realistic example. The example won't require any new material, apart from responding to user input via the mouse, but it will show 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 is functionally quite simple. It 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 on any line, that line will be changed to all uppercase. That's literally all the sample does. Even with this limited set of features, we'll find that the work involved in making sure everything gets displayed in the right place while considering performance issues is quite complex. In particular, we have a new element here, that the contents of the document can change – either when the user selects the menu option to read a new file, or when they double-click to capitalize a line. In the first case we need to update the document size so the scrollbars still work correctly, and we have to redisplay everything. In the second case, we need to check carefully whether the document size is changed, and what text needs to be redisplayed.

We'll start by reviewing the appearance of `CapsEditor`. When the application is first run, it has no document loaded, and displays this:



The File menu has two options: Open and Exit. Exit exits the application, while Open brings up the standard `OpenFileDialog` and reads in whatever file the user selects. The next screenshot shows `CapsEditor` being used to view its own source file, `Form1.cs`. I've also randomly double-clicked on a couple of lines to convert them to uppercase:



The sizes of the horizontal and vertical scrollbars are, by the way, correct. The client area will scroll just enough to view the entire document. `CapsEditor` doesn't try to wrap lines of text – the example is already complicated enough without doing that. 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 we are assuming it is a text file and doesn't contain any non-printable characters.

We'll start off by adding a `using` command:

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

This is because we'll be using the `StreamReader` class which is in `System.IO`. Next we'll add in 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
        private ArrayList documentLines = new ArrayList();   // the 'document'
```

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

Most of these fields should be self-explanatory. The `documentLines` field is an `ArrayList` that contains the actual text of the file that has been read in. In a real sense, 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. Since it's an `ArrayList`, rather than a plain array, we can dynamically add elements to it as we read in a file. You'll notice I've also liberally used `#region` preprocessor directives to block up bits of the program to make it easier to edit.

I said each `documentLines` element contains information about a line of text. This information is actually an instance of another class I've defined, `TextLineInformation`:

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

`TextLineInformation` looks like a classic case where you'd normally use a struct rather than a class since it's just there to group together a couple of fields. However, its instances are always accessed as elements of an `ArrayList`, which expects its elements to be stored as reference types, so declaring `TextLineInformation` as a class makes things more efficient by saving a lot of boxing and unboxing operations.

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'd 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 only change when other parts of the document are modified in some way). In this case we've only stored the `Width` of the item. The reason is because the height in this case is just the height of whatever our selected font is. It's the same for all lines of text so there's no point storing it separately for each one; we 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 as:

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

If we'd 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 I wanted to keep it simple here, which is why we're treating each line of text as one single item.

Let's deal with the main menu now. This part of the application is more the realm of Windows Forms – the subject of Chapter 7, than of GDI+. I added the menu options using the design view in Visual Studio .NET, but renamed them as `menuFile`, `menuFileOpen`, and `menuFileExit`. I then added event handlers for the File Open and File Exit menu options using the Visual Studio .NET Properties window. The event handlers have their VS .NET-generated names of `menuFileOpen_Click()`and `menuFileExit_Click()`.

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

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

   CreateFonts();
   fileOpenDialog.FileOk += new
      System.ComponentModel.CancelEventHandler(
      this.OpenFileDialog_FileOk);
      fileOpenDialog.Filter =
         "Text files (*.txt)|*.txt|C# source files (*.cs)|*.cs";

}
```

The event handler added here is for when the user clicks OK on the File Open dialog. We have also set the filter for the open file dialog so that we can only load up text files – we've opted for `.txt` files as well as C# source files, so we can use the application to examine the sourcecode for our samples.

`CreateFonts()` is a helper method that sorts out the fonts we 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 stuff:

```
      protected void OpenFileDialog_FileOk(object Sender, CancelEventArgs e)
      {
         this.LoadFile(fileOpenDialog.FileName);
      }

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


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

We'll examine the `LoadFile()` method now. It's the method that handles the opening and reading in of a file (as well as ensuring a `Paint` event gets 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();

           this.Text = standardTitle + " - " + FileName;
           this.Invalidate();
        }
```

Most of this function is just standard file-reading stuff, as covered in Chapter 12. Notice how as the file is read in, we progressively add lines to the `documentLines` `ArrayList`, so this array ends up containing information for each of the lines in order. After we've read in the file, we set the `documentHasData` flag, which indicates whether there is actually anything to display. Our next task is to work out where everything is to be displayed, and, having done that, how much client area we need to display the file – the document size that will be used to set the scrollbars. Finally, we set the title bar text and call `Invalidate()`. `Invalidate()` is an important method supplied by Microsoft, so we'll break for a couple of pages to explain its use, before we examine the code for the `CalculateLineWidths()` and `CalculateDocumentSize()` methods.

The Invalidate() Method

`Invalidate()` is a member of `System.Windows.Forms.Form` that we've not met before. 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. There are a couple of overrides to `Invalidate()`: you can pass it a rectangle that specifies (in page coordinates) precisely which area of the window needs repainting, or if you don't pass any parameters it'll just mark the entire client area as invalid.

You may wonder why we are doing it this way. If we know that something needs painting, why don't we 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, in general you should call `Invalidate()`. Here's 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 our example, if we'd directly called a method to do the drawing from the `LoadFile()` method, then the `LoadFile()` method wouldn't return until that drawing task was complete. During that time, our application can't respond to any other events. On the other hand, by calling `Invalidate()` we are simply getting Windows to raise a `Paint` event before immediately returning from `LoadFile()`. Windows is then free to examine the events that are waiting 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, it picks one and calls the corresponding event handler. Although the `Paint` event may be the only one sitting in the queue (so `OnPaint()` gets called immediately anyway), in a more complex application there may be other events that ought to get priority over our `Paint` event. In particular, if the user has decided to quit the application, this will be marked by a message known as `WM_QUIT`.
- Related to the first reason, if you have a more complicated, multithreaded, application, you'll 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 (whatever thread is responsible for the message queue – this will be the thread that called `Application.Run()`) does all the drawing, no matter what other thread requested the drawing operation.
- There's an additional performance-related reason. Suppose at about the same time a couple of different requests to draw part of the screen come in. 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.
- Finally, 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 guys 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's easier to do if you don't 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've drawn, then you may 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 may 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` *that 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

We'll return to the `CapsEditor` example now and examine the `CalculateLineWidths()` and `CalculateDocumentSize()` methods that are called from `LoadFile()`:

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

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. We store the value, because `MeasureString()` is computationally intensive. If we hadn't made the `CapsEditor` sample so simple that we can easily work out the height and location of each item, this method would almost certainly have needed to be implemented in such a way as to compute all those quantities too.

Now we know how big each item on the screen is, and we can calculate where each item goes, we are in a position to work out the actual document size. The height is basically 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, we will also want to make an allowance for a small margin around the displayed document, to make the application look more attractive.

Here's 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 + 2*margin;
         if (tempLineLength > maxLineLength)
            maxLineLength = tempLineLength;
      }
      documentSize.Width = (int)maxLineLength;
   }
   this.AutoScrollMinSize = documentSize;
}
```

This method first checks whether there is any data to be displayed. If there isn't we cheat a bit and use a hard-coded document size, which I happen to know is big enough to display the big red <Empty Document> warning. If we'd wanted to really do it properly, we'd have used `MeasureString()` to check how big that warning actually is.

Once we've worked out the document size, we tell the `Form` instance what the size is by setting the `Form.AutoScrollMinSize` property. When we 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 do I say that's interesting? It illustrates perfectly what I was saying earlier about using the `Form.Invalidate()` method. You see, if you look back at the code for `LoadFile()` you'll realize that our call to `Invalidate()` in that method is actually redundant. The client area will be invalidated anyway when we set the document size. I left the explicit call to `Invalidate()` in the `LoadFile()` implementation to illustrate how in general you should normally do things. In fact in this case, all calling `Invalidate()` again will do is needlessly request a duplicate `Paint` event. However, this in turn illustrates what I was saying about how `Invalidate()` gives Windows the chance to optimize performance. The second `Paint` event won't in fact get raised: Windows will see that there's 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'll be an negligible amount of time compared to how long it takes to actually do some painting.

OnPaint()

Now we've 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 = this.AutoScrollPosition.X;
   int scrollPositionY = this.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 >= this.documentLines.Count ||
      maxLineInClipRegion == -1)
   maxLineInClipRegion = this.documentLines.Count-1;

   TextLineInformation nextLine;
   for (int i=minLineInClipRegion; i<=maxLineInClipRegion ; i++)
   {
```

```
            nextLine = (TextLineInformation)documentLines[i];
            dc.DrawString(nextLine.Text, mainFont, mainBrush,
                          this.LineIndexToWorldCoordinates(i));
         }
      }
```

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 to do with optimizing the painting – the usual stuff about figuring out what exactly needs painting instead of rushing in and telling the graphics instance to redraw everything.

We start off by checking if there is any data in the document. If there isn't, we draw a quick message saying so, call the base class's `OnPaint()` implementation, and exit. If there is data, then we start looking at the clipping rectangle. The way we do this is by calling another method that we've written, `WorldYCoordinateToLineIndex()`. We'll examine this method 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 we call the `WorldYCoordinateToLineIndex()` method, we 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`, we'll play safe and assume we need to start at the beginning of the document (this is the case if the top of the clipping region is in the top margin).

Once we've done all that, we essentially repeat the same process for the bottom of the clipping rectangle, in order 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 we can just run a `for` loop between these values to do our painting. Inside the painting loop, we actually need to do roughly the reverse transformation to the one performed by `WorldYCoordinateToLineIndex()`. We are given the index of a line of text, and we need to check where it should be drawn. This calculation is actually quite simple, but we've 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's fine, because we have already called `TranslateTransform()` on the `Graphics` object so that we need to pass it world, rather than page, coordinates when asking it to display items.

Coordinate Transforms

In this section, we'll examine the implementation of the helper methods that we've written in the `CapsEditor` sample to help us with coordinate transforms. These are the `WorldYCoordinateToLineIndex()` and `LineIndexToWorldCoordinates()` methods that we referred to in the last 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;
      }
```

We also used a method that roughly does the reverse transform in `OnPaint()`. `WorldYCoordinateToLineIndex()` works out the line index, but it only takes into account a vertical world coordinate. This is 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, we 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 isn't 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 occasions we 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, we add the `AutoScrollPosition`, which is negative.

Although these methods by themselves don't look particularly interesting, they do illustrate a general technique that you'll probably often need to use. With GDI+, we'll often find ourselves in a situation where we have been given some coordinates (for example the coordinates of where the user has clicked the mouse) and we'll need to figure out what item is being displayed at that point. Or it could happen the other way round – given a particular display item, whereabouts should it be displayed? Hence, if you are writing a GDI+ application, you'll 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 we've 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. We're now going to add that facility 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, and indeed we've already covered how handle user input in Chapter 7. Essentially, you override methods from the `Form` class that get called from the relevant event handler – in much the same way that `OnPaint()` is called when a `Paint` event is raised.

For the case of detecting when the user clicks or moves the mouse the methods you may wish to override include:

Method	Called when:
`OnClick(EventArgs e)`	mouse is clicked
`OnDoubleClick(EventArgs e)`	mouse is double-clicked
`OnMouseDown(MouseEventArgs e)`	left mouse button pressed
`OnMouseHover(MouseEventArgs e)`	mouse stays still somewhere after moving
`OnMouseMove(MouseEventArgs e)`	mouse is moved
`OnMouseUp(MouseEventArgs e)`	left mouse button is released

If you want to detect when the user types in any text, then you'll probably want to override these methods:

Method	Called when:
`OnKeyDown(KeyEventArgs e)`	a key is depressed
`OnKeyPress(KeyPressEventArgs e)`	a key is pressed and released
`OnKeyUp(KeyEventArgs e)`	a pressed key is released

Notice that some of these events overlap. For example, if the user presses a mouse button this will raise the `MouseDown` event. If the button is immediately released again, this will raise the `MouseUp` event and the `Click` event. Also, 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 `KeyEventArgs` and `KeyPressEventArgs` have properties that indicate which key or keys the event concerns.

That's all there is to it. It's then up to you to think about the logic of precisely what you want to do. The only point to note is that you'll probably find yourself doing a bit more logic work with a GDI+ application than you would have with a `Windows.Forms` application. That's because in a `Windows.Forms` application you are typically responding to quite high-level events (`TextChanged` for a textbox, for example). By contrast with GDI+, the events tend to be more basic – user clicks the mouse, or hits the key *h*. The action your application takes is likely to depend on a sequence of events rather than a single event. For example, say your application works like Word for Windows, where in order to select some text the user clicks the left mouse button, then moves the mouse, then releases the left mouse button. Your application will receive the `MouseDown` event, but there's 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'll want 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. 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 really 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, though the coding you do will be quite fiddly, as you may need to take into account a lot of 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 sooner or later some user is going to try it!

For the `CapsEditor` example, we are keeping things very simple, so we don't really have any combinations to think about. The only thing we are going to respond to is when the user double-clicks – in which case we capitalize whatever line of text the mouse pointer is hovering over.

This should be a fairly simple task, but there is one snag. We need to trap the `DoubleClick` event, but the table above shows that this event takes an `EventArgs` parameter, not a `MouseEventArgs` parameter. The trouble is that we'll need to know where the mouse is when the user double-clicks, if we 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 that is implemented by the `Form1` object, `Control.MousePosition`, to find out the mouse position, like so:

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

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 doesn't get called until perhaps half a second or so *after* the user has double-clicked. You don't really want delays like that, because they usually annoy users intensely, but even so, occasionally it does happen, and sometimes for reasons beyond the control of your app (a slow computer for instance). Trouble is, half a second is easily enough time for the mouse to get moved halfway across the screen – in which case your call to `Control.MousePosition` will return completely the wrong location!

A better way 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 then we know that `OnMouseDown()` has also just been called, with the mouse at the same location. We can use the `OnMouseDown()` override to record the position of the mouse, ready for `OnDoubleClick()`. This is the approach we take in `CapsEditor`:

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

Now let's look at our `OnDoubleClick()` override. There's quite a bit more work to do here:

```
protected override void OnDoubleClick(EventArgs e)
{
   int i = PageCoordinatesToLineIndex(this.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)
      {
         this.documentSize.Width = (int)newWidth;
         this.AutoScrollMinSize = this.documentSize;
      }
      Rectangle changedRectangle = new Rectangle(
                                       LineIndexToPageCoordinates(i),
                                       new Size((int)newWidth,
                                       (int)this.lineHeight));
      this.Invalidate(changedRectangle);
   }
   base.OnDoubleClick(e);
}
```

We 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 we weren't over any text, so there's nothing to do; except, of course, call the base class version of `OnDoubleClick()` to let Windows do any default processing.

Assuming we've identified a line of text, we 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 we kept the sample so simplistic, there aren't too many combinations. We can assume for a start, 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. We also know that since we are not wrapping lines, our line of text won't overflow to the next line and push out other text below. Our action of converting the line to uppercase won't, therefore, actually change the locations of any of the other items being displayed. That's 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 our line the longest line, and cause the width of the entire document to increase. If that's the case then we'll 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, we need to get the screen redrawn, by calling `Invalidate()`. Only one line has changed; therefore, we don't want to have the entire document repainted. Rather, we need to work out the bounds of a rectangle that contains just the modified line, so that we can pass this rectangle to `Invalidate()`, ensuring that just that line of text will be repainted. That's precisely what the above code does. Our call to `Invalidate()` will result in `OnPaint()` being called, when the mouse event handler finally returns. Bearing in mind our comments earlier in the chapter 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, you'll find that the `PaintEventArgs` parameter to `OnPaint()` does indeed contain a clipping region that matches the specified rectangle. And since we've overloaded `OnPaint()` to take careful account of the clipping region, only the one required line of text will be repainted.

Printing

In this chapter we've focused so far entirely 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 too. That's the topic of this section. We're going to extend the `CapsEditor` sample so that it is able to print preview and print the document that is being edited.

Unfortunately, we don't have enough space to go into too much detail about printing here, so the printing functionality we will implement will be very basic. Usually, if you are implementing the ability for an application to print data, you will add three items to the application's main File menu:

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

In our case, to keep things simple, we won't implement a Page Setup menu option. Printing will only be possible using default settings. We will note, however, that, if you do want to implement Page Setup, then Microsoft has already written a page setup dialog class for you to use. It is the class `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 that we need to use are `System.Drawing.Printing.PrintDocument` and `System.Drawing.Printing.PrintPreviewDialog`. These two classes handle the process of making sure that drawing instructions passed to a device context get appropriately handled for printing, leaving you to think about the logic of what to print where.

There are some important differences between printing/print previewing on the one hand, and displaying to the screen on the other hand. Printers cannot scroll – instead they have pages. So you'll need to make sure 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'll need, and which page each part of the document needs to be written to.

Despite the above 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 will internally cause an event, `PrintPage`, to be raised to signal the printing of 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`, `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 will modally display 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 once again by repeatedly raising the `PrintPage` event until the `HasMorePages` property is `false`. There's no need to write a separate event handler for this; you can use the same event handler as used for printing each page since 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 we've gone over the broad steps to be taken, let's see how this works in code terms. The code is downloadable as the `PrintingCapsEdit` project, and consists of the `CapsEditor` project, with the changes highlighted below made.

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

```
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();

      this.Text = standardTitle + " - " + FileName;
      this.Invalidate();
   }
```

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

```
      public class Form1 : System.Windows.Forms.Form
      {
         private int pagesPrinted = 0;
```

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

Next, the event handlers for when the user selects 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 += new PrintPageEventHandler
         (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();
   }
```

We've already explained the broad procedure involved in printing, and we can see that these event handlers are simply implementing that procedure. In both cases we are instantiating a `PrintDocument` object and attaching an event handler to its `PrintPage` event. For the case of printing, we call `PrintDocument.Print()`, while for print previewing, we attach the `PrintDocument` object to a `PrintPreviewDialog`, and call the preview dialog object's `ShowDialog()` method. The real work is going to be done in that event handler to the `PrintPage` event – and this is what that handler looks like:

```
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 = this.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++;
}
```

After declaring a couple of local variables, the first thing we 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 we recall from the `CapsEditor` sample 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 we arguably could have cached the value the first time we calculated it. However, the calculation isn't too hard, and in a more sophisticated application the value might change, so it's not bad practice to recalculate it every time we print a page.

We 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.

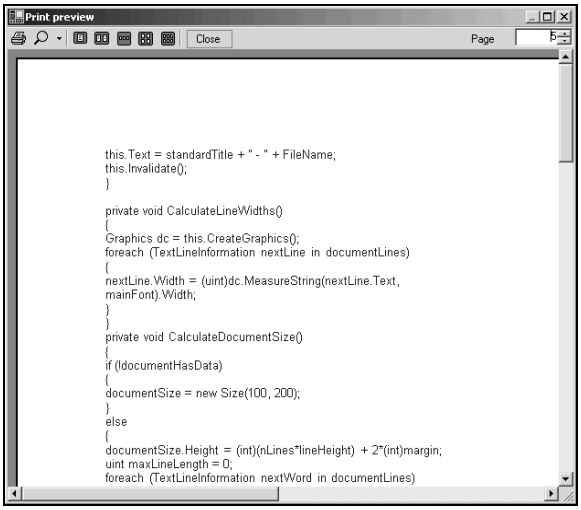
Next we run through a loop, printing each line. This loop will terminate either when we find that we have printed all the lines of text in the document, or when we find that we have printed all the lines that will fit on this page – whichever condition occurs first. Finally, we check whether there is any more of the document to be printed, and set the `HasMorePages` property of our `PrintPageEventArgs` accordingly, and also increment the `pagesPrinted` field, so that we 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 we do not worry about where the drawing commands are being sent. We 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 we are printing, the `Graphics` object will have been hooked up to the printer, while if we are print previewing then the `Graphics` object will have been hooked up to the print preview form on the screen.

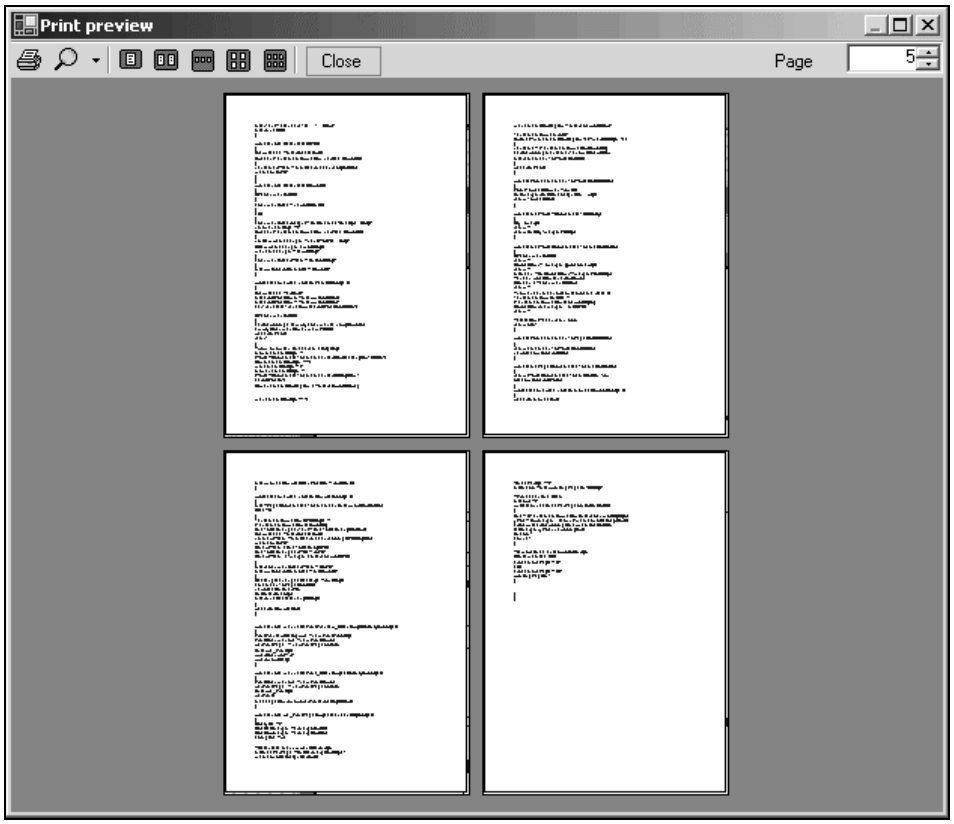
Finally, we need to ensure the `System.Drawing.Printing` namespace is searched for type definitions:

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

All that remains is to compile the project and check that the code works. We can't really show screenshots of a printed document(!) but this is what happens if you run `CapsEdit`, load a text document (as before, we've picked the C# source file for the project), and select Print Preview:



In the screenshot, we have scrolled through to page 5 of the document, and set the preview to display normal size. The `PrintPreviewDialog` has supplied quite a lot of features for us, as can be seen from the toolbar at the top of the form. The options available include actually printing the document, zooming in or out, and displaying two, three, four or six pages together. These options are all fully functional, without our needing to do any work. For example, if we change the zoom to auto and click to display four pages (third toolbar button from the right), we get this.



Summary

In this chapter, we've covered the area of drawing to a display device, where the drawing is done by your code rather than by some predefined control or dialog – the realm of GDI+. GDI+ is a powerful tool, and there are many .NET base classes available to help you draw to a device. We've 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 doesn't 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 in order 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.

There are many more .NET classes concerned with drawing than we've had space to cover in this chapter, but if you've worked through it and understood the principles involved in drawing, you'll be in an excellent position to explore them, by looking at their lists of methods in the documentation and instantiating instances of them to see what they do. In the end, drawing, like almost any other aspect of programming, requires logic, careful thought, and clear algorithms. Apply that and you'll be able to write sophisticated user interfaces that don't depend on the standard controls. Your software will benefit hugely in both user-friendliness and visual appearance: There are many applications out there that rely entirely on controls for their user interface. While this can be effective, such applications very quickly end up looking just like 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 your sales!