Bonus Chapter 2

Printing with Visual Basic 2010

Evangelos Petroutsos

The topic of printing with Visual Basic is a not trivial, and many developers use third-party tools to add print capabilities to their applications. As you already know, there’s no control with built-in printing capabilities. It would be nice if certain controls, such as the TextBox or the ListView control, would print their contents, but this is not the case. Even to print a few text paragraphs entered by the user on a TextBox control, you must provide your own code.

Printing with VB isn’t complicated, but it requires a lot of code — most of it calling graphics methods. You must carefully calculate the coordinates of each graphic element placed on the paper, take into consideration the settings of the printer and the current page, and start a new page when the current one is filled. It’s like generating graphics for the monitor, so you need a basic understanding of the graphics methods, even if you’re only going to develop business applications. If you need to generate elaborate printouts, I suggest that you look into third-party controls with built-in printing capabilities, because the controls that come with Visual Studio have no built-in printing capabilities.

The examples of this tutorial will address many of your day-to-day needs, and I’m including examples that will serve as your starting point for some of the most typical printing needs, from printing tabular data to bitmaps.

The Printing Components

We’ll start our exploration of Visual Basic’s printing capabilities with an overview of the printing process, which is the same no matter what you print. In the following section, you’ll find a quick overview of the printing controls (you’ll find more information on them, as well as examples, in the following sections). You don’t need to use all these components in your project. Only the PrintDocument component is required, and you will have to master the members of this control.

The PrintDocument Control

This object represents your printer, and you must add a PrintDocument control to any project that generates printouts. In effect, everything you draw on the PrintDocument object is sent to the printer. The PrintDocument object represents the printing device, and it exposes a Graphics object that represents the printing surface, just like the Graphics property of all Windows controls. You can program against the Graphics object by using all the graphics methods. If you can create drawings on a form, you can just as easily print them on your printer. To print text, for example, you must call the DrawString method. You can also print frames around the text with the DrawLine or DrawRectangle method. In general, you can use all the methods of the Graphics object to prepare the printout.
The PrintDocument control is invisible at runtime, and its icon will appear in the Components tray at design time. When you're ready to print, call the PrintDocument object's Print method. This method doesn't produce any output, but it does raise the control's BeginPrint and PrintPage events. The BeginPrint event is fired as soon as you call the Print method, and this is where you insert the printout's initialization code. The PrintPage event is fired once for every page of the printout, and this is where you must insert the code that generates output for the printer. Finally, the EndPrint event is fired when the printout ends, and this is where you insert the code to reset any global variables.

The following statement initiates the printing:

```vbnet
PrintDocument1.Print
```

This statement is usually placed in a button's or a menu item's Click event handler. To experiment with simple printouts, create a new project, place a button on the form, add an instance of the PrintDocument object to the form, and enter the preceding statement in the button's Click event handler.

After the execution of this statement, the PrintDocument1_PrintPage event handler takes over. This event is fired for each page, so you insert the code to print the first page in this event's handler. The PrintPage event exposes the `e` argument, which gives you access to the Graphics property of the current Printer object. This is the same object you use to generate all kinds of graphics on a PictureBox control or a Form. The printer has its own Graphics object, which represents the page you print on. If you need to print additional pages, you set the `e.HasMorePages` property to True just before you exit the event handler. This will fire another PrintPage event. The same process will repeat until you've printed everything. After you finish, you set the `e.HasMorePages` property to False, and no more PrintPage events will be fired. Instead, the EndPrint event will be fired and the printing process will come to an end.

**Figure 2.1**

All printing takes place in the PrintPage event handler of the PrintDocument object.

---

The code in Listing 2.1 shows the structure of a typical PrintPage event handler. The PrintPage event handler prints three pages with the same text but a different page number on each page.
Listing 2.1: A Simple PrintPage Event Handler

Private Sub PrintDocument1_PrintPage( _
    ByVal sender As Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
Handles PrintDocument1.PrintPage
Static pageNum As Integer
Dim prFont As New Font("Verdana", 24, GraphicsUnit.Point)
e.Graphics.DrawString( _
    "PAGE " & pageNum + 1, prFont, _
    Brushes.Black, 700, 1050)
e.Graphics.DrawRectangle(Pens.Blue, 0, 0, 300, 100)
e.Graphics.DrawString( _
    "Printing with VB 2005", prFont, _
    Brushes.Black, 10, 10)
' Add more printing statements here
' Following is the logic that determines whether we're done printing
pageNum = pageNum + 1
If pageNum <= 3 Then
    e.HasMorePages = True
Else
    e.HasMorePages = False
    pageNum = 0
End If
End Sub

Notice that the page number is printed at the bottom of the page, but the corresponding statement is the first one in the subroutine. I assume that you’re using a letter-size page, so I hard-coded the coordinates of the various elements in the code. You’ll see shortly how to take into consideration not only the dimensions of the physical page, but also its orientation.

The `pageNum` variable is declared as Static, so it retains its value between invocations of the event handler and isn’t reset automatically. The last statement resets the `pageNum` variable in anticipation of another printout. Without this statement, the first page of the second printout (if you clicked the button again) would be page 4, and so on. Moreover, the printout would never come to an end because the `pageNum` variable would never become less than 3. Every time you repeat a printout, you must reset the global and static variables. This is a common task in printing with the PrintDocument control, and is a common source of many bugs.

Initialization of Static Variables

You can also declare variables such as the `pageNum` variable at the form’s level, so that they’ll retain their value between successive invocations of the PrintPage event handler. These variables can be reset in the PrintDocument’s BeginPrint event handler, which is fired every time you start a new printout by calling the PrintDocument.Print method, or at the end of the printing process.
The code of Listing 2.1 uses the drawing methods of the `e.Graphics` object to generate the printout. After printing something and incrementing the page number, the code sets the `e.HasMorePages` property to True, to fire the `PrintPage` event again, this time to print the next page. As long as there are more pages to be printed, the program sets the `e.HasMorePages` property to True. After printing the last page, it sets the same argument to False to prevent further invocations of the `PrintPage` event. If you want to print a single page, you can ignore everything in this listing, except for the drawing methods that produce the output.

The entire printout is generated by the same subroutine, one page at a time. Because pages are not totally independent of one another, we need to keep some information in variables that are not initialized every time the `PrintPage` event handler is executed. The page number, for example, must be stored in a variable that will maintain its value between successive invocations of the `PrintPage` event handler, and it must be increased every time a new page is printed. If you’re printing a text file, you must keep track of the current text line, so that each page will pick up where the previous one ended, not from the beginning of the document. You can use static variables or declare variables on the form’s level, whatever suits you best. This is a recurring theme in programming the `PrintPage` event, and you’ll see many more examples of this technique in the following sections. I can’t stress enough the importance of resetting these variables at the end of a printout (or initializing them at the beginning of the printout).

The PrintDialog Control

The PrintDialog control displays the standard Print dialog box, shown in Figure 2.2, which allows users to select a printer and set its properties. If you don’t display this dialog box, the output will be sent automatically to the default printer and will use the default settings of the printer.

**Figure 2.2**
The Print dialog box

To display the Print dialog box, call the PrintDialog control’s `ShowDialog` method. However, you must set the control’s `PrinterSettings` property first; if not, a runtime exception will be thrown. We usually display the Print dialog box via the following statements:

```vbnet
PrintDialog1.PrinterSettings = PrintDocument1.PrinterSettings
If PrintDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
    PrintDocument1.PrinterSettings = PrintDialog1.PrinterSettings
End If
```
Among other settings, the Print dialog box allows you to specify the range of pages to be printed. Before allowing users to select a range, be sure that you have a way to skip any number of pages. If the user specifies pages 10 through 19, your code must calculate the section of the document that would normally be printed on the first nine pages, skip it, and start printing after that. If the printout is a report with a fixed number of rows per page, skipping pages is trivial. If the printout contains formatted text, you must execute all the calculations to generate the first nine pages and ignore them (skip the statements that actually print the graphics). Starting a printout at a page other than the first one can be a challenge, so make sure that your code will work before enabling the Print Range zone in the Print dialog box.

When users select a printer in this dialog box, it automatically becomes the active printer. Any printout generated after the printer selection will be sent to that printer; you don’t have to insert any code to switch printers. The actual printer to which you will send the output of your application is almost transparent to the printing code. The same commands will generate the same output on any printer. It is also possible to set the printer from within your code by using a statement like the following, where printer is the name of one of the installed printers:

```
PrintDocument1.PrinterSettings.PrinterName = printer
```

For more information on selecting a printer from within your code, see the section “Retrieving the Printer Names,” later in this tutorial. There are times when you want to set a printer from within your code and not give users a chance to change it. An application that prints invoices and reports, for example, will most likely use a different printer for each type of printout.

**The PageSetupDialog Control**

The PageSetupDialog control displays the Page Setup dialog box, which allows users to set up the page (its orientation and margins). The dialog box, shown in Figure 2.3, returns the current page settings in a PageSettings object, which exposes the user-specified settings as properties. These settings don’t take effect on their own; you simply read their values and take them into consideration as you prepare the output for the printer from within your code. As you can see, there aren’t many parameters to set in this dialog box, but you should display it and take into account the settings specified by the user.

**Figure 2.3**
The Page Setup dialog box
To use this dialog box in your application, drop the PageSetupDialog control on the form and call its ShowDialog method from within the application’s code. The single property of this control that you’ll be using exclusively in your projects is the PageSettings property. PageSettings is an object that exposes a number of properties reflecting the current settings of the page (margins and orientation). These settings apply to the entire document. The PrintDocument object has an analogous property: the DefaultPageSettings property. After the user closes the Page Setup dialog box, we assign its PageSettings property to the DefaultPageSettings property of the PrintDocument object to make the user-specified settings available to our code. Here’s how we usually display the Page Setup dialog box from within our application and retrieve its PageSettings property:

```vbnet
If PageSetupDialog1.ShowDialog() = DialogResult.OK Then
End If
```

Notice that the first line that initializes the dialog box is mandatory. If you attempt to display the dialog box without initializing its PageSettings property, an exception will be thrown. We’ll explore the properties of the PageSettings object and we’ll use it in most of the examples of this tutorial. You can also create a new PageSettings object, set its properties, and then use it to initialize the Page Setup dialog box.

The statements that manipulate the printing objects can get fairly lengthy. It’s common to use the With structure to make the statements shorter. The preceding code segment can also be coded as follows:

```vbnet
With PageSetupDialog1
    If .ShowDialog() = DialogResult.OK Then
    End If
End With
```

To change the default margins in the Page Setup dialog box before displaying it, you can create a new PageSettings object and set its Margins property as shown in the following code segment. The margins are specified in the default coordinate system, and they correspond to 1.25, 1.75, 1, and 2 inches because the default coordinate system of the page is 1/100 of an inch.

```vbnet
Dim PS As New System.Drawing.Printing.PageSettings
PS.Margins.Left = 125
PS.Margins.Right = 175
PS.Margins.Top = 100
PS.Margins.Bottom = 200
PageSetupDialog1.PageSettings = PS
If PageSetupDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
    PrintDocument1.Print()
End If
```
**Different Locales Use Different Units**

If the application is running on a computer with a European locale, the margins will be converted to tenths of a millimeter (or hundredths of a centimeter). The values of the previous example will be mapped to 12.5, 17.5, 10, and 20 millimeters. The default coordinates of the page, however, are always expressed in hundredths of an inch. If you request the values of the `Margins.Left` and `Margins.Right` properties of the `PrintDocument1.DefaultPageSettings` object, you’ll get back the values 49 and 69. 49/100 of an inch corresponds (practically) to half an inch, which is the same as 12.5 millimeters (there are 25.4 millimeters in an inch). The value 125 corresponds to one and a quarter inches if the target computer uses the American locale, but only half an inch if the computer is using a European locale. The `PageSetupDialog` control, however, will display the appropriate units in the Margins section (inches or millimeters).

**The PrintPreviewDialog Control**

Print Preview is another dialog box that displays a preview of the printed document. It exposes a lot of functionality and allows users to examine the output and, optionally, to send it to the printer. The Print Preview dialog box, shown in Figure 2.4, is made up of a preview pane, where you can display one or more pages at the same time at various magnifications, and a toolbar. The buttons on the toolbar allow users to select the magnification, set the number of pages that will be displayed on the preview pane, move to any page of a multipage printout, and send the preview document to the printer.

**Figure 2.4**
The Print Preview dialog box
After you write the code to generate the printout, you can direct it to the PrintPreviewDialog control. You don’t have to write any additional code; just place an instance of the control on the form and set its Document property to the PrintDocument control on the form. Then show the control instead of calling the Print method of the PrintDocument object:

```vbnet
PrintPreviewDialog1.Document = PrintDocument1
PrintPreviewDialog1.ShowDialog
```

After the execution of these two lines, the PrintDocument control takes over. It fires the PrintPage event as usual, but it sends its output to the Print Preview dialog box, not to the printer. The dialog box contains a Print button, which the user can click to send the document being previewed to the printer. The exact same code that generated the preview document will print the document on the printer.

The PrintPreviewDialog control can save you a lot of paper and toner when you test your printing code, because you don’t have to print every page to see what it looks like. Because the same code generates both the preview and the actual printed document, and the Print Preview option adds a professional touch to your application, there’s no reason why you shouldn’t add this feature to your projects.

### You Can’t Use the PrintPreviewDialog Control without a Printer

The PrintPreviewDialog control generates output that would normally be printed by the default printer (or the printer selected in the Print dialog box). If this printer is a networked printer that your computer can’t access at the time, the PrintPreview dialog box will not be displayed. Instead, an exception will be thrown, which you must catch from within your code.

Of course, this control is no substitute for actual printing tests. You should also try to generate physical printouts (on several types of printers, if possible) to uncover any problems with your printing code before your customers do. For example, most printers can’t print near their page edges, but this isn’t a problem for the PrintPreviewDialog control. If you print near the edges, the printout will appear fine on the preview pane, but some unexpected cropping might occur on the hard copy. Some black-and-white printers might translate colors to gray shades poorly, and what appears light gray on the monitor during a preview might show as black on a printout.

I mentioned earlier that the PageSettings class exposes the Margins property, which returns the margins specified by the user on the PageSetupDialog control. The PageSettings class also exposes the HardMarginX and HardMarginY properties, which return the width and height of the unprintable area of the page, respectively. For my ink-jet printer, the two values are 25 and 11 (in hundredths of an inch). Use these two properties in your code to make sure that the margins specified by the user are at least equal to the page’s hard margins.

The trivial sample code presented so far prints three simple pages to the printer. To redirect the output of the program to the PrintPreview control, add an instance of the PrintPreview control to the form and replace the statement that calls the PrintDocument1.Print method in the button’s Click event handler with the following statements:

```vbnet
PrintPreviewDialog1.Document = PrintDocument1
PrintPreviewDialog1.ShowDialog
```

Run the project, and this time you preview the document on your monitor. If you’re satisfied with its appearance, you can click the Print button to send the document to the printer.
To avoid runtime errors, you can use the following exception handler, whether you print
directly to the printer or you’re displaying a printout preview:

Try
    PreviewDialog1.Document = PrintDocument1
    PreviewDialog1.ShowDialog
Catch exc As Exception
    MsgBox &quot;The printing operation failed” & vbCrLf & exc.Message
End Try

Printer and Page Properties
Before you can generate a printout, you must retrieve the settings of the current printer and
page, and this is a good place to present the members of these two objects because we’ll use
them extensively in the examples of the following sections. The properties of these two items
are reported to your application through the PrinterSettings and the PageSettings objects.
The PageSettings object is a property of the PrintPageEventArgs class, and you can access it
through the e argument of the PrintPage event handler. The DefaultPageSettings property
of the PrintDocument component exposes the current page’s settings.
The PrinterSettings object is a property of the PrintDocument object, as well as a prop-
erty of the PageSetupDialog and PrintDialog controls. Finally, one of the properties exposed by
the PageSettings object is the PrinterSettings object. These two objects provide all the infor-
mation you might need about the selected printer and the current page through the properties
listed in Tables 2.1 and 2.2.

<table>
<thead>
<tr>
<th>PROPERTY</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounds</td>
<td>Returns the bounds of the page (Bounds.Width and Bounds.Height). If the current orientation is landscape, the width is larger than the height.</td>
</tr>
<tr>
<td>Color</td>
<td>Returns, or sets, a True/False value that indicates whether the current page should be printed in color. On a monochrome printer, this property is always False.</td>
</tr>
<tr>
<td>Landscape</td>
<td>A True/False value that indicates whether the page is printed in landscape or portrait orientation.</td>
</tr>
<tr>
<td>PaperSource</td>
<td>The page’s paper tray.</td>
</tr>
<tr>
<td>PrinterResolution</td>
<td>The printer’s resolution for the current page.</td>
</tr>
<tr>
<td>PrinterSettings</td>
<td>This property returns, or sets, the printer settings associated with the page. For more information on the PrinterSettings object and the properties it exposes, see Table 2.2.</td>
</tr>
</tbody>
</table>
### Table 2.2: The Members of the PrinterSettings Object

<table>
<thead>
<tr>
<th>MEMBER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstalledPrinters</td>
<td>This method retrieves the names of all printers installed on the computer. The same printer names also appear in the Print dialog box, in which the user can select any one of them.</td>
</tr>
<tr>
<td>CanDuplex</td>
<td>A read-only property that returns a True/False value indicating whether the printer supports double-sided printing.</td>
</tr>
<tr>
<td>Collate</td>
<td>Another read-only property that returns a True/False value indicating whether the printout should be collated.</td>
</tr>
<tr>
<td>Copies</td>
<td>This property returns the requested number of copies of the printout.</td>
</tr>
<tr>
<td>DefaultPageSettings</td>
<td>This property is the PageSettings object that returns, or sets, the default page settings for the current printer.</td>
</tr>
<tr>
<td>Duplex</td>
<td>This property returns, or sets, the current setting for double-sided printing.</td>
</tr>
<tr>
<td>FromPage, ToPage</td>
<td>The printout's starting and ending pages, as specified in the Print dialog box by the user.</td>
</tr>
<tr>
<td>IsDefaultPrinter</td>
<td>Returns a True/False value that indicates whether the selected printer (the one identified by the PrinterName property) is the default printer. Note that selecting a printer other than the default one in the Print dialog box doesn't change the default printer.</td>
</tr>
<tr>
<td>IsPlotter</td>
<td>Returns a True/False value that indicates whether the printer is a plotter.</td>
</tr>
<tr>
<td>IsValid</td>
<td>Returns a True/False value that indicates whether the PrinterName corresponds to a valid printer.</td>
</tr>
<tr>
<td>LandscapeAngle</td>
<td>Returns an angle, in degrees, by which the portrait orientation must be rotated to produce the landscape orientation.</td>
</tr>
<tr>
<td>MaximumCopies</td>
<td>Returns the maximum number of copies that the printer allows you to print at a time.</td>
</tr>
<tr>
<td>MaximumPage</td>
<td>Returns, or sets, the largest value that the FromPage and ToPage properties can have.</td>
</tr>
<tr>
<td>MinimumPage</td>
<td>Returns, or sets, the smallest value that the FromPage and ToPage properties can have.</td>
</tr>
<tr>
<td>PaperSizes</td>
<td>Returns all the paper sizes that are supported by this printer.</td>
</tr>
</tbody>
</table>
Table 2.3: The Members of the PrinterSettings Object (CONTINUED)

<table>
<thead>
<tr>
<th>MEMBER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaperSources</td>
<td>Returns all the paper source trays on the selected printer</td>
</tr>
<tr>
<td>PrinterName</td>
<td>Returns, or sets, the name of the printer to use</td>
</tr>
<tr>
<td>PrinterResolutions</td>
<td>Returns all the resolutions that are supported by this printer</td>
</tr>
<tr>
<td>PrintRange</td>
<td>Returns, or sets, the numbers of the pages to be printed, as</td>
</tr>
<tr>
<td></td>
<td>specified by the user. When you set this property, the value</td>
</tr>
<tr>
<td></td>
<td>becomes the default setting when the Print dialog box is opened.</td>
</tr>
<tr>
<td>SupportsColor</td>
<td>Returns a True/False value that indicates whether this printer</td>
</tr>
<tr>
<td></td>
<td>supports color printing.</td>
</tr>
</tbody>
</table>

Retrieving the Printer Names

To retrieve the names of the installed printers, use the InstalledPrinters collection of the PrinterSettings object. This collection contains the names of the printers as strings, and you can access them with the following loop:

```vbnet
Dim i As Integer
With PrintDocument1.PrinterSettings.InstalledPrinters
    For i = 0 To .Count - 1
        Debug.WriteLine(.Item(i))
    Next
End With
```

These statements will produce output such as the following when executed:

Fax
HPLaser
\TOOLKIT\XEROX

The first two printers are local (Fax isn’t even a printer; it’s a driver for the fax and it’s installed by Windows). The last printer’s name is XEROX, and it’s a network printer connected to the TOOLKIT workstation.

You can also change the current printer by setting the PrinterName property of the PrinterSettings property with either of the following statements:

```vbnet
PrintDocument1.PrinterSettings.PrinterName = "HPLaser"
```
Another property that needs additional explanation is the PrinterResolution property. The PrinterResolution property is an object that exposes provides the Kind property, which returns, or sets, the current resolution of the printer, and its value is one of the PrinterResolutionKind enumeration’s members: Custom, Draft, High, Low, and Medium. To find out the exact horizontal and vertical resolutions, read the X and Y properties of the PrinterResolution property. When you set the PrinterResolutionKind property to Custom, you must specify the X and Y properties.

Page Geometry

Printing on a page is similar to generating graphics onscreen. Like the drawing surface on the monitor (the client area), the page on which you’re printing has a fixed size and resolution. The most challenging aspect of printing is the calculation of the coordinates and dimensions of each graphic element on the page. In business applications, the most common elements are strings (rendered in various fonts, styles, and sizes), lines, and rectangles, which are used as borders for tabular data.

Although you can print anywhere on the page, we usually print one element at a time, calculate the space it takes on the page, and then print the next element next to or below it. Printing code makes heavy use of the MeasureString method, and nearly all the examples presented in this tutorial use this method.

The printable area is determined by the size of the paper you’re using, and in most cases it’s 8.5 × 11 inches (keep in mind that most printers can’t print near the edge of the page). Printed pages have a margin on all four sides, and users can set a different margin on each side through the Page Setup dialog box. Your program should confine its printing within the specified margins.

To access the current page’s margins, use the Margins property of the PrintDocument1.DefaultPageSettings object. This object exposes the Left, Right, Top, and Bottom properties, which are the values of the four margins. The margins, as well as the page coordinates, are expressed in hundredths of an inch. The width of a standard letter-sized page, for example, is 8,500 units, and its height is 11,000 units. Of course, you can use non-integer values for even greater granularity, but you won’t see two straight lines printed at less than one-hundredth of an inch apart. You can use other units, which are all members of the PageUnit enumeration. In the examples of this tutorial, I’m using the default units (1/100 of an inch).

Another property exposed by the DefaultSettings object is the PageSize property, which represents the dimensions of the page. The width and height of the page are given by the following expressions:

\[
\]

The top of the page is at coordinates (0, 0), which correspond to the top-left corner of the page. We never actually print at this corner. The coordinates of the top-left corner of the printable area of the page are given by the following expressions:

\[
\text{PrintDocument1.DefaultPageSettings.Margins.Top} \\
\text{PrintDocument1.DefaultPageSettings.Margins.Left}
\]

Now that you have seen how to use the printing components, their basic properties, and the page’s geometry, you can look at some examples that demonstrate how to generate practical printouts.
VB 2010 at Work: The SimplePrintout Project

Let’s put the information of the preceding paragraphs together to build a simple application that prints a string at the top-left corner of the page (the origin of the page) and a rectangle that delimits the page’s printable area. To print something, start by dropping the PrintDocument object on your form. Then place a button on the form and enter the following statement in its Click event handler:

```vbnet
PrintDocument1.Print()
```

This statement tells the PrintDocument object that you’re ready to print. The PrintDocument object will fire the BeginPrint event, in which you can place any initialization code (reset the variables that must maintain their value between consecutive invocations of the PrintPage event handler, for example). Then, it will fire the PrintPage event, whose definition is the following:

```vbnet
Private Sub PrintDocument1_PrintPage( _
    ByVal sender As Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
Handles PrintDocument1.PrintPage
End Sub
```

As implied by its name, the PrintPage event is fired once for each page. You must place the VB code required to produce the desired output in this event’s handler. To access the page in the printer from within the PrintPage event’s handler, use the `e.Graphics` property, which is a Graphics object. Anything you draw on this object is printed on paper.

To print a string at the page’s top-left corner, call the Graphics object’s `DrawString` method, as shown here:

```vbnet
Dim pFont As Font
pFont = New Font("Comic Sans MS", 20)
e.Graphics.DrawString("ORIGIN", pFont, Brushes.Black, 0, 0)
```

The last two arguments of the `DrawString` method are the coordinates of a point where the string will be printed. The string is printed right below the origin, so that it’s visible. If you attempt to print a string at the bottom-right corner of the page, the entire string will fall just outside the page, and no visible output will be produced. The coordinates passed to the `DrawString` method are the coordinates of the upper-left corner of a box that encloses the specified string.

No matter what your default printer is, it’s highly unlikely that it’s been set to no margins. The page’s margins aren’t enforced by the PrintDocument object; you must respect them from within your code because it is possible to print anywhere on the page. To take into consideration the page’s margins, change the coordinates from (0, 0) to the left and top margins.

You can also use the other members of the Graphics object to generate graphics. The following statement will render the text on the page by using an anti-alias technique (anti-aliased text looks much smoother than text rendered with the default method):

```vbnet
```

Next, we’ll print a rectangle around the area of the page in which we’re allowed to print — a rectangle delimited by the margins of the page. To draw this rectangle, we need to know the size of all four margins and the size of the page (obviously). To read (or set) the page’s margins, use the `PrintDocument1.DefaultPageSettings.Margin` object, which provides...
the Left, Right, Top, and Bottom properties. We’re also going to need the dimensions of the page, which we can read through the Width and Height properties of the PrintDocument1.DefaultPageSettings.PaperSize object. The four margins are calculated and stored in four variables via the following statements:

```vba
Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
    Lmargin = .Left
    Rmargin = .Right
    Tmargin = .Top
    Bmargin = .Bottom
End With
```

The rectangle we want to draw should start at the point (Lmargin, Tmargin) and extend PrintWidth units to the right and PrintHeight units down. These two variables are the width and height of the page minus the respective margins, and they’re calculated with the following statements:

```vba
Dim PrintWidth, PrintHeight As Integer
    PrintWidth = .Width - Lmargin - Rmargin
    PrintHeight = .Height - Tmargin - Bmargin
End With
```

Then insert the following statements in the PrintPage event handler to draw the rectangle:

```vba
Dim R As Rectangle
R = New Rectangle(Lmargin, Tmargin, PrintWidth, PrintHeight)
e.Graphics.DrawRectangle(Pens.Black, R)
```

The printing takes place from within the PrintPage event handler, which is shown in Listing 2.2. The event handler contains all the statements presented in the previous paragraphs and a few comments.

---

**LISTING 2.2: Generating a Simple Printout**

```vba
    ' Turn on antialias for text
    ' Print a string at the origin
    Dim pFont As Font
    pFont = New Font('Comic Sans MS', 20)
e.Graphics.DrawString('ORIGIN', pFont, Brushes.Black, 0, 0)
    ' Read margins into local variables
    Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
```

---

The printing takes place from within the PrintPage event handler, which is shown in Listing 2.2. The event handler contains all the statements presented in the previous paragraphs and a few comments.

---

**LISTING 2.2: Generating a Simple Printout**

```vba
    ' Turn on antialias for text
    ' Print a string at the origin
    Dim pFont As Font
    pFont = New Font('Comic Sans MS', 20)
e.Graphics.DrawString('ORIGIN', pFont, Brushes.Black, 0, 0)
    ' Read margins into local variables
    Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
```

---

The printing takes place from within the PrintPage event handler, which is shown in Listing 2.2. The event handler contains all the statements presented in the previous paragraphs and a few comments.

---

**LISTING 2.2: Generating a Simple Printout**

```vba
    ' Turn on antialias for text
    ' Print a string at the origin
    Dim pFont As Font
    pFont = New Font('Comic Sans MS', 20)
e.Graphics.DrawString('ORIGIN', pFont, Brushes.Black, 0, 0)
    ' Read margins into local variables
    Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
```
Lmargin = .Left
Rmargin = .Right
Tmargin = .Top
Bmargin = .Bottom
End With
' Calculate the dimensions of the printable area
Dim PrintWidth, PrintHeight As Integer
    PrintWidth = .Width - Lmargin - Rmargin
    PrintHeight = .Height - Tmargin - Bmargin
End With
' Now print the rectangle
Dim R As Rectangle
    R = New Rectangle(Lmargin, Tmargin, PrintWidth, PrintHeight)
e.Graphics.DrawRectangle(Pens.Black, R)
End Sub

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

In this section, we’ll write a more elaborate application to print a rectangle bounded by the margins of the page as before. In addition to printing the rectangle, the application also prints four strings, one in each margin, with different orientations (as seen in Figure 2.5). The project that generated the output is called PageSettings, and it also demonstrates how to display the Page Setup dialog box from within your code and then generate a printout according to the settings on this dialog box.

**Figure 2.5**
The output of the Page-Settings project
You saw the statements that print a rectangle enclosing the printable area of the page. Printing the labels is a bit involved. Because the four strings appear in all four orientations, some rotation transformation is involved. We'll discuss the code for printing the captions later. For now, let's examine the PageSetupDialog control and how you take into consideration the settings in this dialog box from within your code.

**Setting Up the Page**

To display the Page Setup dialog box, first place an instance of the PageSetupDialog control on your form. Then set its PageSettings property to a PageSettings object that contains the default settings for the printer. We usually set this property to the DefaultPageSettings property of the PrintDocument object, although you can create a new PageSettings object and set its properties from within your code. Finally, display the dialog box by calling its ShowDialog method:

```vbnet
If PageSetupDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
End If
```

Upon return, we assign the PageSettings property of the control to the DefaultPageSettings property of the `PrintDocument1` control. Now, we must take into consideration the settings specified in the dialog box from within the PrintPage event's code. The area on the page in which we must restrict our output is a rectangle with its top-left corner at the left and top margins, and its dimensions being the width and height of the page (less the corresponding margins). The following statements set up a few variables to hold the page's dimensions:

```vbnet
Dim PrintWidth, PrintHeight As Single
Dim PageWidth, PageHeight As Single
    PrintWidth = .Width - LMargin - RMargin
    PrintHeight = .Height - TMargin - BMargin
    PageWidth = .Width
    PageHeight = .Height
End With
```

A few additional statements are required if the user changes the orientation of the page. When you're printing in landscape mode, the size of the paper doesn't change. If you examine the Width and Height properties of the PaperSize object, you'll realize that the page is always taller than it is wide. This means that we must swap the width and height from within our code. The margins, however, remain the same. Notice that as you change the orientation of the page in the Page Setup dialog box, the margins are swapped automatically (the left and right margins become top and bottom, respectively).

To find out whether the user has changed the page's orientation, examine the Landscape property of the DefaultPageSettings object. If this property is True, it means that the user wants to print in landscape mode, and you must swap the page's width and height. The following statements calculate the dimensions of the page area within the margins when the orientation is set to landscape:

```vbnet
If PrintDocument1.DefaultPageSettings.Landscape Then
        PrintWidth = .Height - TMargin - BMargin
        PrintHeight = .Width - LMargin - RMargin
    End With
```
PRINTING THE LABELS

Now we can focus on the code that prints the captions in the space of the four margins, which is considerably more elaborate. The top margin's caption isn't rotated; it's printed at the default orientation. The caption in the right margin is rotated by 90 degrees, and the caption in the bottom margin is rotated by 180 degrees. The caption in the left margin is rotated by –90 degrees. These rotations take place around the origin, so the labels must also be moved to their places with a translation transformation. Let's look at the code that prints the Right Margin String caption, shown in Listing 2.3.

**Listing 2.3:** Printing a Caption in the Right Margin

```csharp
strWidth = e.Graphics.MeasureString(RMarginCaption, pFont).Width
strHeight = e.Graphics.MeasureString(RMarginCaption, pFont).Height
X = PageWidth - (Rmargin - strHeight) / 2
Y = TMargin + (PrintHeight - strWidth) / 2
e.Graphics.ResetTransform()
e.Graphics.TranslateTransform(X, Y)
e.Graphics.RotateTransform(90)
e.Graphics.DrawString(RMarginCaption, pFont, Brushes.Black, 0, 0)
```

First, we calculate the string's width and height by using the MeasureString method and store them in the `strWidth` and `strHeight` variables. The string will be rotated by 90 degrees before being printed. The rotation alone would place the string just outside the left margin, so we must translate it to the right. The amount of the translation is the page's width minus half the difference between the string's height and the right margin. Translating the caption by the width of the page would bring it to the very right edge of the paper. To center it in the right margin, we must split the difference of the string's height from the right margin on either side of the string. We're using the string's height in calculating the x-coordinate and the string's width in calculating the y-coordinate because after the string is rotated by 90 degrees, the width and height will be swapped. X and Y are the amounts by which the string must be moved along the horizontal and vertical axes. The rotation of the string will be performed by a rotation transformation. Because transformations are cumulative, the code resets any existing transformations and applies two new ones.

Then, the DrawString method is called to print the string. The DrawString method draws the string at the point (0, 0), but the two transformations will place it at the proper location. This is the simplest method for printing transformed strings (or any other graphic element): Set up the appropriate transformation(s) and then draw the string at the origin.

The code for placing the other three captions is quite analogous. It uses the proper translation and rotation transformations, and the only complication is the calculation of the
coordinates of the translation transformation. The listing of the PrintPage event handler of the PageSettings project is fairly lengthy. Listing 2.4 shows the code that prints the caption in the right margin.

**Listing 2.4:** Printing the Rectangle and the Margin Captions

```vbnet
Private Sub PrintDocument1_PrintPage(...) Handles PrintDocument1.PrintPage
    Dim R As Rectangle
    Dim strWidth, strHeight As Integer
    Dim pFont As Font
    pFont = New Font("Comic Sans MS", 20)
    e.Graphics.DrawString("ORIGIN", pFont, Brushes.Black, 0, 0)
    pFont = New Font("Comic Sans MS", 40)
    Dim X, Y As Integer
    Dim TMarginCaption As String = "Top Margin String"
    Dim LMarginCaption As String = "Left Margin String"
    Dim RMarginCaption As String = "Right Margin String"
    Dim BMarginCaption As String = "Bottom Margin String"
    Dim LMargin, RMargin, TMargin, BMargin As Integer
        LMargin = .Left
        RMargin = .Right
        TMargin = .Top
        BMargin = .Bottom
    End With
    Dim PrintWidth, PrintHeight, PageWidth, PageHeight As Integer
        PrintWidth = .Width - LMargin - RMargin
        PrintHeight = .Height - TMargin - BMargin
        PageWidth = .Width
        PageHeight = .Height
    End With
    If PrintDocument1.DefaultPageSettings.Landscape Then
            PrintWidth = .Height - TMargin - BMargin
            PrintHeight = .Width - RMargin - LMargin
            PageWidth = .Height
            PageHeight = .Width
        End With
    End If
    ' Draw rectangle
    R = New Rectangle(LMargin, TMargin, PageWidth - LMargin - RMargin, _
        PageHeight - BMargin - TMargin)
    e.Graphics.DrawRectangle(Pens.Black, R)
    strWidth = e.Graphics.MeasureString(RMarginCaption, pFont).Width
    strHeight = e.Graphics.MeasureString(RMarginCaption, pFont).Height
    X = PageWidth - (RMargin - strHeight) / 2
```
As always, you must call the PrintDocument object’s Print method for this event handler to be activated. You can use the Print method of the PrintDocument object, but the sample project uses the PrintPreviewDocument object to display a preview of the printout. Listing 2.5 shows the code behind the button on the form.

Listing 2.5: The Print Button

```
Private Sub Button1_Click(...) Handles Button1.Click
    Try
        PrintPreviewDialog1.Document = PrintDocument1
        If PageSetupDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
            PrintPreviewDialog1.ShowDialog()
        End If
    Catch exc As Exception
        MsgBox("Printing Operation Failed" & vbCrLf & exc.Message)
    End Try
End Sub
```

The code uses an exception handler to prevent the program from crashing with a runtime exception if there’s a problem with the printer. The application should work if there’s a default printer; it will fail to generate a preview only if the default printer is a network printer and you have no access to it at the time.

The first statement sets up the PrintPreview control by setting its Document property to the PrintDocument object. The second statement assigns the default page settings to the PageSetupDialog control, and the following statement displays the Page Setup dialog box. After the user has specified the desired settings and closed the dialog box, the new settings are assigned to the PrintDocument object’s DefaultPageSettings property. The last statement displays the Print Preview dialog box. This statement initiates the printing process, which sends its output to the preview pane instead of the printer. That’s all it takes to add a preview feature to your application.

If you feel uncomfortable with the transformations, especially the rotation transformation, Figure 2.6 shows what happens to a string when it’s rotated in all four directions around the origin. The origin — the point with coordinates (0, 0) — is where the two axes meet.
The statements in the PrintPage event handler rotate the string \textit{GDI + Graphics} around the origin by 90, 180, and 270 degrees. The numbers in parentheses indicate the angle of rotation for each string. Of course, I couldn’t print to the left of the origin or above the origin, so I had to rotate the translated string by 50 percent of the page’s width to the right and 50 percent of the page’s height down, to appear at the middle of the page. The two axes were also translated by the same amounts in the two directions. This illustration’s purpose is to help you visualize how the string is rotated around the origin. Besides the string itself, the enclosing rectangle is also printed. This is the rectangle returned by the \texttt{MeasureString} method, subject to the same transformations as the string it encloses. To examine the code that produced Figure 2.6, open the RotatedStrings sample project in Visual Studio; the printing code is well documented in the code, and you should be able to understand and experiment with it.

**Practical Printing Examples**

In principle, using the Framework’s printing components is straightforward. Depending on the type of printout you want to generate, however, the code of the PrintPage event handler can get quite complicated. Because there are no techniques that you can apply to all situations, I included a few typical examples to demonstrate how to use the same objects to perform very different tasks. The first example demonstrates how to print tabular reports, which is the most common report type for business applications. A tabular report has the form of a grid, with columns of different widths and rows of different heights.

The second example is the printing of text, and even if this is the least exciting type of printout, you should be able to send text to the printer. As it turns out, it’s not a trivial operation. The last example prints bitmaps, probably the simplest type of printout. The only challenge
with printing bitmaps is that you might have to reduce the size of the bitmap to make it fit in the width or the height of the page, or a rectangular area within the page.

**Printing Tabular Data**

The printing operation you’ll be using most often in typical business applications that require custom printing is that of tabular data. Figure 2.7 shows an example of a printout with tabular data. This printout was generated by the PrintTable project.

**FIGURE 2.7**

Using the PrintTable application to print data in a tabular arrangement

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0393049513</td>
<td>The Dream of Reason: A History of Philosophy from the Greeks to the Renaissance</td>
<td>Anthony Gottlieb</td>
</tr>
<tr>
<td>0156495085</td>
<td>In Search of the Inconclusive / Fragments of an Unknown Teaching</td>
<td>P. O. Oquistok</td>
</tr>
<tr>
<td>0472065211</td>
<td>Simulations and Simulation</td>
<td>Stan Birkland, Paul Farrell-Kreid</td>
</tr>
<tr>
<td>0065057633</td>
<td>Logic in 50 Minutes</td>
<td>Ralph W. Aitken</td>
</tr>
<tr>
<td>0850970745</td>
<td>paradoxes of Time and Being (2007 to the Task of Thinking)</td>
<td>Paul Wartman, Ralph W. Aitken</td>
</tr>
<tr>
<td>1890518649</td>
<td>The Inflation of His Follies</td>
<td>Ralph W. Aitken</td>
</tr>
<tr>
<td>1861022790</td>
<td>Professional Arduino 2.5 Programming (Arduino Professional Guides)</td>
<td>Delia F. Zilker</td>
</tr>
<tr>
<td>0782138103</td>
<td>Networking Complete</td>
<td>Delia F. Zilker</td>
</tr>
<tr>
<td>0156495085</td>
<td>In Search of the Inconclusive / Fragments of an Unknown Teaching</td>
<td>P. O. Oquistok</td>
</tr>
<tr>
<td>0472065211</td>
<td>Simulations and Simulation</td>
<td>Stan Birkland, Paul Farrell-Kreid</td>
</tr>
<tr>
<td>0065057633</td>
<td>Logic in 50 Minutes</td>
<td>Ralph W. Aitken</td>
</tr>
<tr>
<td>1890518649</td>
<td>The Inflation of His Follies</td>
<td>Ralph W. Aitken</td>
</tr>
<tr>
<td>0850970745</td>
<td>paradoxes of Time and Being (2007 to the Task of Thinking)</td>
<td>Paul Wartman, Ralph W. Aitken</td>
</tr>
</tbody>
</table>

The ISBN column contains a 10-character string, and it’s quite simple to handle. All you have to do is make sure that the ISBN will fit in the corresponding column. If you allow the user to select the font at runtime and you can’t set a fixed width for this column, you should print only as many characters as will fit in the reserved width. In this example, we won’t do anything special with the ISBN column. You can also retrieve the width of a 10-character string in the specific font and use this value (plus a small margin) as the column’s width.

The Title column has a variable length, and you might have to break long titles into two or more printed lines — this is the real challenge of the application. The DrawString method can print a string in a rectangle you pass as an argument. The width of this rectangle must be the same as the width of the Title column. The height of the rectangle should be enough for the entire text to fit in it. In our code, we’ll use a rectangle with the appropriate width and adequate height to make sure that the entire title will be printed. Alternatively, you can trim the title if it’s too long, but there’s no point in trimming substantial information.

The last intricacy of this application is the Author(s) column. Each book might have no authors, one author, or more, and we’ll print each author on a separate line. The total height
of each row depends on the height of the Title or Author(s) cell: Note in Figure 2.7 that the height of some lines is determined by the height of the Title cell, while the height of others is determined by the height of the Author(s) cell. We must keep track of the height of these two cells and move down accordingly before printing the following row. Where the height of the Author(s) cells is determined by the number of authors (we’ll print each author on a single line and assume that the name does not exceed the width of the page), we must provide the code to break the title into multiple lines. If an author’s name does not fit in a single line, it will be truncated and an ellipsis will appear in the place of the missing characters.

So, where does the data come from? It could come from a text file, an XML document, or a database. It doesn’t really make a difference, as long as you can access one row at a time and extract its fields. For the purposes of this example, and because we haven’t discussed databases yet, I’m using a ListView control to store the data. The ListView control is populated with the Load Data button on the form of Figure 2.8 (the project’s main form). Each book is a different item in the ListView, and the various fields are subitems. Each item’s Text property is the book’s ISBN, and the remaining fields are stored as subitems. I took sample data from an online bookstore and, in some cases, edited their titles to make them long or added fictitious authors. Here are the statements that populate the ListView control with the first two items:

```vbscript
Dim BookItem As New ListViewItem
BookItem.Text = "0393049515"
BookItem.SubItems.Add("
   'The Dream of Reason:
      A History of Philosophy from
      the Greeks to the Renaissance"
)
BookItem.SubItems.Add("Anthony Gottlieb")
ListView1.Items.Add(BookItem)

BookItem = New ListViewItem
BookItem.Text = "0156445085"
BookItem.SubItems.Add("In Search of the Miraculous :
   Fragments of an Unknown Teaching ")
BookItem.SubItems.Add("P. D. Ouspensky")
ListView1.Items.Add(BookItem)
```

**Figure 2.8**
The PrintTable project’s main form
Notice that the ListView control has four columns (one for the ISBN, one for the title, and two for author names), but you can add as many authors to each title as you wish. Subitems beyond the fourth one are invisible on the ListView control, but they're there. After the list has been populated, you can click the Preview & Print Data button to generate the preview and print the report. The main form of the PrintTable project, populated with the data shown in the sample printout, is shown in Figure 2.8.

**FORMATTING THE CELLS**

The report is generated one row at a time. The vertical coordinate of the current row is stored in the variable \( y \), which is incremented accordingly for each new row. This coordinate applies to all the cells of the current row, and if a cell contains multiple lines, the \( y \)-coordinate is adjusted accordingly for the following row. The \( x \)-coordinate of each column is the same for all rows. These coordinates are calculated at the beginning and don't change from row to row.

Breaking a string into multiple lines isn't trivial. You should include as many words as you can on each line without exceeding the available width. Fortunately, the Graphics object's `MeasureString` method can break a string into the required number of lines to fit the string into a rectangle and report the number of lines. This form of the `MeasureString` method is as follows:

```csharp
Graphics.MeasureString(string, font, size, format, cols, lines)
```

The first argument is the string to be printed, and it will be rendered in the font specified by the second argument. The `size` argument is the width and height of the rectangle in which the string must fit. In our case, the width is that of the cell in which the string must fit. The `format` argument is a `StringFormat` object that lets you specify various options for printing text (its orientation, for example). You will find more information about this argument in the following section. For the purposes of this example, we'll use the default `FormatString` object. The last two arguments are the number of characters that will fit across the rectangle and the number of lines the string must be broken into, and they're set by the `MeasureString` method. Even if we don't know the height of the rectangle in advance, we can use an absurdly large value. The `MeasureString` method will tell you how many text lines it needs, and you'll use this value to calculate the height of the rectangle. To calculate the height of the cell in which the title will fit, the program uses the following statements:

```csharp
Dim cols, lines As Integer
e.Graphics.MeasureString(strTitle, tableFont, _
    New SizeF(W2, 100), New StringFormat(), _
    lines, cols)
```

`strTitle` is a string variable that holds the title, and `tableFont` is the font in which the string will be rendered. \( W2 \) is the width of the second column of the grid, in which the title appears. This is a fixed value, calculated ahead of time. The initial height of the rectangle is 100 pixels, but this value is totally arbitrary. It is possible for a given cell's text to be so long that it will take a page and a half to print. The PrintTable project can't handle similar extreme situations. You will have to provide additional code to handle the overflow of a cell to the following page.

The `lines` and `cols` variables are passed by reference, so they can be set by the `MeasureString` method to the number of lines and number of characters that will fit in the specified
rectangle. After we have the number of lines it takes for the title to be printed in the specified
width, we can advance the vertical coordinate by the following amount:

\[
\text{lines} \times \text{tableFont.GetHeight(e.Graphics)}
\]

where \text{tableFont} is the font we use to print the table. Its \text{GetHeight} method returns the height
of the font when rendered on the \text{Graphics} object passed as an argument. The few statements
shown here will take care of breaking long titles into multiple lines, which is the most challeng-
ing aspect of the code. The last cell in each row contains a line for each author. The following
loop goes through all the authors and prints them, each one on a separate line:

\[
\text{For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1}
\text{str = ListView1.Items(itm).SubItems(subitm).Text}
\text{e.Graphics.DrawString(str, tableFont, Brushes.Black, X3, Yc)}
\text{Yc = Yc + tableFont.Height + 2}
\text{Next}
\]

The y-coordinate of the last author is stored in the variable \text{Yc}. To calculate the y-coordinate
of the next row of the table, we compare the \text{Y} and \text{Yc} variables and keep the larger value. This
value, plus a small displacement, is used as the y-coordinate for the following line. Listing 2.6
is the complete listing of the \text{PrintPage} event handler of the PrintTable project.

**Listing 2.6:** The PrintPage Event Handler of the PrintTable Project

```vbnet
Private Sub PrintDocument1_PrintPage( _
    ByVal sender As Object, ByVal e As _
Handles PrintDocument1.PrintPage
    e.Graphics.DrawString("Title", TitleFont, Brushes.Black, X2, Y)
    e.Graphics.DrawString("Author(s)", TitleFont, Brushes.Black, X3, Y)
    Y = Y + 30
    While itm < ListView1.Items.Count
        str = ListView1.Items(itm).Text
        e.Graphics.DrawString(str, tableFont, Brushes.Black, X1, Y)
        str = ListView1.Items(itm).SubItems(1).Text
        Dim R As New RectangleF(X2, Y, W2, 80)
        e.Graphics.DrawString(str, tableFont, Brushes.Black, R)
        Dim lines, cols As Integer
        e.Graphics.MeasureString(str, tableFont, _
            New SizeF(W2, 50), New StringFormat(), _
            cols, lines)
        Dim subitm As Integer, Yc As Integer
        Yc = Y
        For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1
            str = ListView1.Items(itm).SubItems(subitm).Text
            e.Graphics.DrawString(str, tableFont, Brushes.Black, X3, Yc)
            Yc = Yc + tableFont.Height + 2
        Next
    End While
End Sub
```
The code uses a few variables that are declared on the form level with the following statements:

```vbnet
Dim tableFont, titleFont As Font
Dim X1, X2, X3 As Integer
Dim W1, W2, W3 As Integer
Dim Y As Integer
Dim itm As Integer
```

**Setting the Column Widths**

Before we can print, we must specify the widths of the columns. Because we know the information we’re going to display in each column, we can make a good estimate of the column widths. The first column, in which the ISBN is displayed, starts at the left margin of the page and extends 120 units to the right, which is an adequate width for printing 13 characters. The default unit is 1/100 of an inch, so the ISBN column’s width is 1.2 inches. The Title column should take up most of the page’s width. In the PrintTable example, I gave 50 percent of the available page width to this column. The remaining space goes to the Author(s) column. You can’t use fixed widths for all columns, because you don’t know the paper size or the page’s orientation. That’s why I’m mixing percentages and allow the last column to fill the space to the right edge of the page. The variables $X1$, $X2$, and $X3$ are the x-coordinates of the left edge of each column, whereas the variables $W1$, $W2$, and $W3$ are the widths of the columns. These variables are set in the Print button’s Click event handler. Then, the subroutine displays the Print Preview dialog box with the document’s preview. Listing 2.7 shows the Print button’s Click event handler. I’m also using different fonts for the headers and the table’s cells.
LISTING 2.7: Setting Up the Columns and Printing the Table

Private Sub Button2_Click(...) Handles Button2.Click
    If PageSetupDialog1.ShowDialog() Then
    End If
    tableFont = New Font("Arial", 8)
    titleFont = New Font("Arial", 12, FontStyle.Bold)
    Dim pageWidth As Integer
    With PrintDocument1.DefaultPageSettings
    End With
    X2 = X1 + 100
    X3 = X2 + pageWidth * 0.5
    W1 = X2 - X1
    W2 = X3 - X2
    W3 = pageWidth - X3
    PrintPreviewDialog1.Document = PrintDocument1
    PrintPreviewDialog1.ShowDialog()
    itm = 0
End Sub

After setting the coordinates and widths of the columns, you can call the ShowDialog method of the PrintPreviewDialog control to preview the document. This method fires the PrintPage event, where we start printing the report by printing the header of the table via the following statements:

end Point, titleFont, Brushes.Black, X2, Y)
e.Graphics.DrawString("Author(s)", titleFont, Brushes.Black, X3, Y)
Y = Y + 30

titleFont is a Font object that represents the font we use for the table header and is declared on the form level. The rest of the program uses the tableFont object, which represents the font in which the table's cells will be rendered.

Then we set up two nested loops. The outer loop goes through all the items on the ListView control, and the inner loop goes through the subitems of the current item. The structure of the two loops is the following:

While itm < ListView1.Items.Count
    print current item
    For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1
        print all subitems
    Next
End While
The PrintTable project is based on the assumption that the author names will fit in the specified width. If not, part of the author name will be truncated. Alternatively, you can print the report in landscape mode — you will have to adjust the widths of the Title and Author(s) columns.

The PrintTable project is the starting point for tabular reports, and it demonstrates the core of an application that prints tables. You will have to add a title to each page, a header and a footer for each page (with page numbers and dates), and quite possibly a grid to enclose the cells. Experiment with the PrintTable project by adding more features to it. You can become as creative as you want with this application. I should also bring to your attention the fact that the PrintTable application ends the page when the report’s height exceeds 95 percent of the page’s printable area. This test takes place after printing each item. If the last title printed on a page has a dozen different authors, it will run over the bottom of the page. You can change this value depending on the type of report and the font you’re using. I’m assuming that no row can fit comfortably in 5 percent of the available printable area, so I end the current page when this point it reached. Note that there’s no mechanism to prevent the last row from overflowing the bottom margin (not by a whole lot, of course). If your report’s cells may contain from 1 to 10 lines of text, you’ll have to come up with a more elaborate test for the end-of-page condition. The application’s code is adequately commented, and you’ll be able to tweak it to your needs.

Because you’re printing the contents of a ListView control, you base the widths of the printout’s columns on the widths of the columns of the ListView control. A column that takes up 20 percent of the control’s width should also take up 20 percent of the width of the form’s printable area. This way, you won’t have to come up with any arbitrary rules for the column widths.

Using Static Variables

The PrintPage event handler produces all the pages, one after the other. These pages, however, are not independent of one another. When you print a long text file, for example, you must keep track of the pages printed so far or the current line. When printing a tabular report, you might have to keep track of the current row. If you set up a variable that keeps track of the current line, you shouldn’t reset this variable every time the PrintPage event handler is executed. One way to maintain the value of a variable between consecutive calls of the same procedure is to declare it with the Static keyword. Static variables maintain their values between calls, unlike the private variables.

In the PrintTable project, I used the \texttt{itm} variable to keep track of the item being printed. By making the variable \texttt{itm} static, we’re sure that it won’t be reset every time the PrintPage event handler is entered. After the completion of the printout, however, we must reset the static variables in anticipation of a new printout. If you neglect to reset the \texttt{itm} variable, the next you time you click the Preview & Print Data button, the code will attempt to print rows past the last one on the ListView control.

Printing Plain Text

In this section, we’ll examine a less-exciting operation: the printing of a text file. It should be a trivial task after the program that prints the tabular reports, but it’s not nearly as trivial as you might think. But why bother with a simple operation such as printing plain text? The reason is that no control has built-in printing capabilities, and text files are still quite common. Printing formatted text is even more complicated, so we’ll start with plain-text files.
Plain text means that all characters are printed in the same font, size, and style — just like the text you enter in a TextBox control. Your task is to start a new page when the current one fills and to break the lines at or before the right margin. Because the text is totally uniform, you know in advance the height of each line and you can easily calculate the number of lines per page ahead of time.

VB 2010 at Work: The PrintText Project

To demonstrate the process of printing text we’ll build the PrintText application. The main form of this application contains a TextBox control and a button that prints the text on the control. The program displays a preview of the text in a Print Preview dialog box, and you can print the text by clicking the Print button in the dialog box. Figure 2.9 shows a section of text previewed with the PrintText application.

The idea is to instantiate a Rectangle object that represents the printable area of the page. Then call the MeasureString method to find out how many characters will fit into the rectangle, and print that many characters with the DrawString method. Just two method calls, and the first page is ready. Repeat the same process for the following pages, starting with the character following the last character printed on the previous page.

The text to be printed is stored in the textToPrint variable, which is declared at the form’s level. To make the application more flexible, I added a Page Setup dialog box, in which users can specify the margins and the orientation of the printout. The application displays the Page Setup dialog box by calling the ShowDialog method of the PageSetupDialog control. Then it initiates printing on an instance of the PrintPreviewDialog control by calling its ShowDialog method. Listing 2.8 shows the code behind the Preview Printout button on the form, which initiates the printing, and the PrintPreview() subroutine.
LISTING 2.8: Initiating the Printing of Plain Text

Private Sub btnPreview_Click(...) Handles btnPreview.Click
    PrintPreview()
End Sub

Public Sub PrintPreview()
    PD = New Printing.PrintDocument
    PSetup.PageSettings = PD.DefaultPageSettings
    If PSetup.ShowDialog() = DialogResult.OK Then
        PPView.Document = PD
        PPView.ShowDialog()
    End If
End Sub

The ShowDialog method of the PrintPreviewDialog control is equivalent to calling the Print method of the PrintDocument control. After that, a series of PrintPage events will follow. Listing 2.9 shows the code in the PrintPage event's handler.

LISTING 2.9: Printing Plain Text

Private Sub PD_PrintPage(ByVal sender As Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
    Handles PD.PrintPage
    Static currentChar As Integer
    Static currentLine As Integer
    Dim txtFont As Font = TextBox1.Font
    Dim txtH, txtW As Integer
    Dim LMargin, TMargin As Integer
    ' Calculate the dimensions of the printable area of the page
    With PD.DefaultPageSettings
        LMargin = PD.DefaultPageSettings.Margins.Left
        TMargin = PD.DefaultPageSettings.Margins.Top
    End With
    e.Graphics.DrawRectangle(Pens.Blue, _
        New Rectangle(LMargin, TMargin, txtW, txtH))
    ' If the text is printed sideways, swap the printable area's width and height
    If PD.DefaultPageSettings.Landscape Then
        Dim tmp As Integer
tmp = txtH
txtH = txtW
txtW = tmp

End If

' Calculate the number of lines per page
Dim linesperpage As Integer = CInt(Math.Round(txtH / txtFont.Height))

' R is the rectangle in which the text should fit
Dim R As New RectangleF(LMargin, TMargin, txtW, txtH)

Dim fmt As StringFormat
If Not TextBox1.WordWrap Then
    fmt = New StringFormat(StringFormatFlags.NoWrap)
    fmt.Trimming = StringTrimming.EllipsisWord
    Dim i As Integer
    For i = currentLine To Math.Min(currentLine + linesperpage, TextBox1.Lines.Length - 1)
    Next
    currentLine += linesperpage
    If currentLine >= TextBox1.Lines.Length Then
        e.HasMorePages = False
        currentLine = 0
    Else
        e.HasMorePages = True
    End If
End If

End If

fmt = New StringFormat(StringFormatFlags.LineLimit)
Dim lines, chars As Integer

e.Graphics.MeasureString(Mid(TextBox1.Text, currentChar + 1), txtFont, New SizeF(txtW, txtH), fmt, chars, lines)
If currentChar + chars < TextBox1.Text.Length Then
    If TextBox1.Text.Substring(currentChar + chars, 1) <> vbLf Then
        While chars > 0
            chars -= 1
        End While
        chars = 1
    End If
End If
The PrintPage event handler is quite lengthy, but if you open the PrintText project, you will find a lot of comments that will help you understand how it works. The core of the printing code is concentrated in the following three statements:

```vbnet
Dim fmt As New StringFormat(StringFormatFlags.LineLimit)
e.Graphics.MeasureString(textToPrint.Substring(currentChar + 1), _
    txtFont, New SizeF(txtW, txtH), fmt, chars, lines)
e.Graphics.DrawString(textToPrint.Substring(currentChar + 1), _
    txtFont, Brushes.Black, R, fmt)
```

The first statement determines the number of characters that will fit in a rectangle with dimensions `txtW` and `txtH` when rendered on the page in the specified font. The `fmt` argument is crucial for the proper operation of the application, and I will explain it momentarily. The `MeasureString` method calculates the number of characters that will fit in the specified rectangle, because all characters will be rendered in the same font and carriage returns are normal characters (in a way, they’re printed and move to the next line).

The second statement prints the segment of the text that will fit in this rectangle. Notice that the code is using the `Substring` method to pass not the entire text, but a segment starting at the location following the last character on the previous page. The location of the first character on the page is given by the `currentChar` variable, which is increased by the number of characters printed on the current page. The number of characters printed on the current page is retrieved by the `MeasureString` method and stored in the `chars` variable.

And the trick that makes this code work is how the `fmt` `StringFormat` object is declared. The height of the printable area of the page might not (and usually does not) accommodate an integer number of lines. The `MeasureString` method will attempt to fit as many text lines in the specified rectangle as possible, even if the last line fits only partially. To force the `MeasureString` and `DrawString` methods to work with an integer number of lines, create a `FormatString` object passing the constant `StringFormatFlags.LineLimit` as an argument:

```vbnet
Dim fmt As New StringFormat(StringFormatFlags.LineLimit)
```

If you pass the `fmt` object as argument to both the `MeasureString` and `DrawString` methods, they ignore partial lines, and the rest of the printing code works as expected.
If the user changes the orientation of the page, the code switches the page’s width and height (see the If PD.DefaultPageSettings.Landscape block in the listing). This is all it takes to print text in landscape orientation. The page’s margins are also accounted for.

The program also takes into consideration the control’s WordWrap property. If word wrapping has been turned off, the program prints only the section of the line that will fit across the page. You can adjust the code to handle program listings. Program listings are plain-text files, like the ones you can print with this application, but you must mark long code lines that are broken to fit on the page. You can insert a special symbol either at the end of a code line that continues on the following line on the page, or in front of the continued line. This symbol is usually a bent arrow that resembles the Enter key. You can also number the lines while printing them.

**Printing Bitmaps**

If you have a color printer, you probably want to print images, too. Actually, most black-and-white printers print images in grayscale too, so you can experiment with the material of this section even if you have only a black-and-white laser printer. As you have probably guessed, you call the DrawImage method to send the bitmap to the printer. As a reminder, the simplest form of the DrawImage method of the Graphics object accepts two arguments, the bitmap to be drawn (an Image object) and a rectangle in which the image will be drawn:

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

The method will stretch the bitmap specified by the `image` argument to fill the rectangle specified by the `rectangle` argument. It’s imperative that you carefully calculate the dimensions of the rectangle, so that they will retain their original aspect ratio. If not, the image will be distorted in the process. Most applications will let the user specify a zoom factor, and then apply it to both dimensions. If the image fits on the page in actual size, you can make the rectangle equal to the dimensions of the image and not worry about distortions.

Because the reduced image will, most likely, be smaller than the dimensions of the paper on which it will be printed, you must also center the image on the paper. To do so, you can subtract the image’s width from the paper’s width and split the difference on the two sides of the image (you will do the same for the vertical margins). These operations will be demonstrated with the code of the PrintBitmap application, whose main form is shown in Figure 2.10. The application allows you to load an image and zoom in or out. The Zoom > Auto command resizes the image to fit the current size of the form as best as possible, while the Zoom > Normal command displays the image in actual size, regardless of whether it fits on the form or not. If not, the appropriate scroll bars will be attached automatically, because the form’s AutoSize property is set to True.

If you specify a rectangle the same size as the image, the image will be printed at its actual size. A common image resolution is 72 dots per inch. If the bitmap is 1,024 pixels wide, it will take approximately 14 inches across the page — this means that part of the image won’t be printed.

If the bitmap is too large for a letter-size page, you must reduce its size. The following statements, which must appear in the PrintDocument event, print the image centered on the page. If the image doesn’t fit on the page, its top-left corner is printed at the origin, and the rightmost and bottommost parts of the image will be cropped. Notice also that the image isn’t printed in actual size; instead, it’s printed at the current magnification. Listing 2.10 provides the code of the PrintPage event handler.
**Figure 2.10**
The PrintBitmap application resizes and rotates bitmaps to best fit the width of the page and prints them.

**Listing 2.10:** Scaling and Printing a Bitmap

```vbnet
    Dim R As Rectangle
    Dim PictWidth, PictHeight, PictLeft, PictTop As Integer
    PictWidth = PictureBox1.Width
    PictHeight = PictureBox1.Height
        If PictWidth < .Width Then
            PictLeft = (.Width - PWidth) / 2
        Else
            PictLeft = 0
        End If
        If PictHeight < .Height Then
            PictTop = (.Height - PHeight) / 2
        Else
            PictTop = 0
        End If
    End With
    R = New Rectangle(PictLeft, PictTop, PictWidth, PictHeight)
    e.Graphics.DrawImage(PictureBox1.Image, R)
End Sub
```
The `PictWidth` and `PictHeight` variables hold the dimensions of the scaled image, whereas `PictLeft` and `PictTop` are the coordinates of the image’s top-left corner on the page. To initiate the printing process, you must call the PrintDocument object’s `Print` method, or you can display the Print Preview dialog box, which is what the following code does:

```vbnet
Private Sub btnPrint_Click(...) Handles btnPrint.Click
    PrintPreviewDialog1.Document = PrintDocument1
    PrintPreviewDialog1.ShowDialog()
End Sub
```

The PrintBitmap application allows the user to resize and rotate the image before printing it. These rotation commands can be found in the main form’s Process menu; the Zoom menu has four options: Auto, Normal, Zoom In, and Zoom Out (Figure 2.11). The last two commands zoom in and out, respectively, by 25 percent at a time. These commands change the size of the PictureBox control that holds the image, and the `PrintPage` event handler uses the dimensions of this control to determine the dimensions of the printed image. The Normal command resets the image to its actual size, and the Auto command resizes the image proportionally so that its height is 400 pixels.

**Figure 2.11**
The PrintBitmap application’s main form