Printing in C# Made Easy

Platform Support: .NET 2.0, 1.1, 1.0
Language: C#

Authors:
C# Corner Authors Team
Published on: May 15, 2007

Printing Introduction

This article covers the following topics -

- Basic understanding of printing process in .NET
- How to get and set printer and page settings
- How to get and set paper size
- How to get and set paper tray
- How to print text and text files
- How to print drawings and graphic shapes
- How to print images
- How to use print dialogs
- How to write your own custom printing and page setup dialogs
- How to print multi page documents
- How to print a Form and its controls
- How to print invoices
- How to print a ruler
- How to print a bar code
- How to print a DataGridView control

Understanding the Printing Process

The following steps are required in a printing process.

Step 1: Specify a printer

First step to specify what printer you are going to use for printing. In code, we create a `PrintDocument` object and specify the printer by setting its PrinterName property.

Step 2: Set the printer and page properties.

This is an optional step. This step includes setting printer and page settings. If we don't set these properties, the default settings of the printer will be used.

Step 3: Set the print-page event handler.

The print-page event handler is responsible for printing. We create a print-page event handler by setting the `PrintDocument.PrintPage` member.

Step 4: Print the document.
Finally, we call the `PrintDocument.Print` method, which sends printing objects to the printer.

NOTE

All printing related functionality in .NET Framework is defintied in the System.Drawing.Printing namespace. Before you use any printing related classes or objects, you must import the System.Drawing.Printing namespace using the following code:

```
using System.Drawing.Printing;
```

NOTE

Before you use any printer-related classes in your application, a printer must be installed on your machine.

Hello, Printer!

Now, let's create our first printing application. In this application, we will print a text, "Hello, Printer!" to the printer from our application.

Create a Windows Forms application using Visual Studio and import the System.Drawing.Printing namespace using the following code:

```
using System.Drawing.Printing;
```

Now, we add a label, a combo box, and a button controls to the form and change their names and text accordingly. We change combo box's Name property to to PrintersList and button's Name property to PrintButton. The final form should look like Figure 1.

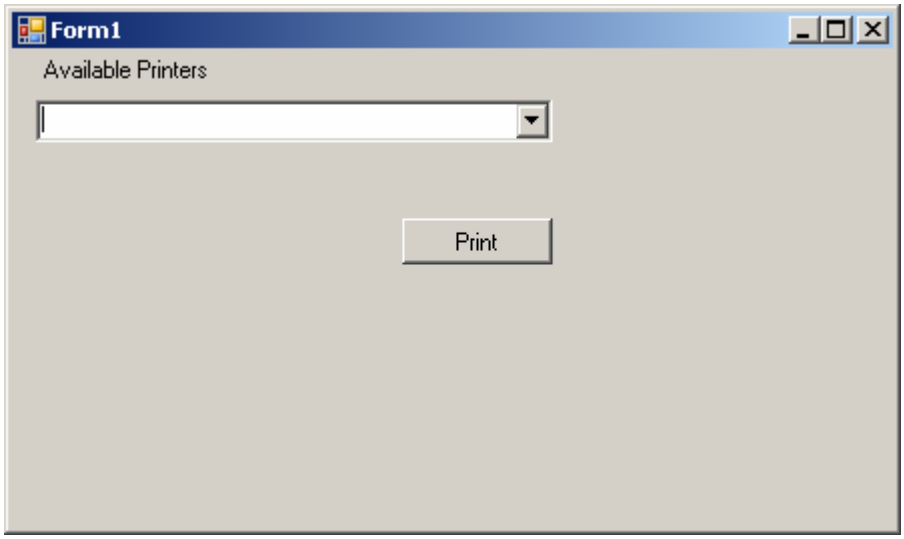

Figure 1: Hello, Printer! Application

On let's load all installed printers in the PrintersList combo box by adding Listing 1 code on the Form's Load event handler.

Listing 1: Getting all installed printers

```
private void Form1_Load(object sender, EventArgs e)
{
    foreach (String printer in PrinterSettings.InstalledPrinters)
    {
        printersList.Items.Add(printer.ToString());
    }
}
```

The `PrinterSetting.InstalledPrinter` in Listing 1 returns the installed printers on a machine.

Now, let's add code to the Print button click event handler by double clicking on the Print bytton. See Listing 2.

In Listing 2, we create a `PrintDocument` object and set the `PrintDocument.PrinterSettings.PrinterName` property to the printer selected from the printer list combo box. Then we add print-page event handler and call the `PrintDocument.Print` method, which actually prints the document.

Listing 2: The Print button click event handler

```
private void PrintButton_Click(object sender, EventArgs e)
{
      //Create a PrintDocument object
      PrintDocument PrintDoc = new PrintDocument();
      //Set PrinterName as the selected printer in the printers list
      PrintDoc.PrinterSettings.PrinterName =
PrintersList.SelectedItem.ToString();
      //Add PrintPage event handler
      PrintDoc.PrintPage += new PrintPageEventHandler(PrintPageMethod);
      //Print the document
      PrintDoc.Print();
}
```

The last step is to add the print-page event handler code. See code Listing 3. This code is responsible for creating a `Graphics` object for the printer. It calls the `DrawString` method, which is responsible for drawing text. First we create a `Graphics` object from `PrintPageEventArgs.Graphics.` Then we create `Font` and `SolidBrush` objects and call `DrawString` to draw some text on the printer. The `DrawString` method takes a string that represents the text to be drawn; the font; a brush; and a layout rectangle that represents the starting point, width, and height of a rectangle for the text.

Listing 3: The print-page event handler

```
public void PrintPageMethod(object sender, PrintPageEventArgs ppev)
{
//Get the Graphics object
Graphics g = ppev.Graphics;
//Create a font verdana with size 14
Font font = new Font("Verdana", 20);
//Create a solid brush with red color
SolidBrush brush = new SolidBrush(Color.Red);
// Create a rectangle
Rectangle rect = new Rectangle(50, 50, 200, 200);
//Draw text
g.DrawString("Hello, Printer! ", font, brush, rect);
}
```

Now, build and run the application, select a printer from the printers list, and click the **Print** button. You should see “Hello, Printer!” on you printed page on the printer.

Working with Printer and Page Properties

There are times when you need to control printer and page settings programmatically such as number of copies, paper size, paper kind and print quality. All this may be controlled through the PrinterSettings property of PrintDocument and PageSettings classes, which is represented by the PrinterSettings class.

Let’s take a quick look at the PrinterSettings class and its members. After that, we will write an application that allows us to get and set printer and page settings programmatically.

The `InstalledPrinters` static property returns the names of all available printers on a machine, including printers available on the network. The following code snippet iterates through all the available printers on a machine.

```
foreach (String printer in
PrinterSettings.InstalledPrinters)
{
    string str = printer.ToString();
}
```

The `PaperSizes` property returns the paper sizes supported by a printer. It returns all the paper sizes in a `PrinterSettngs.PaperSizeCollection` object. The following code snippet iterates through all the available paper sizes.

```
PrinterSettings prs = new PrinterSettings();
foreach (PaperSize ps in prs.PaperSizes)
{
    string str = ps.ToString();
}
```


The `PrinterResolutions` property returns all the resolutions supported by a printer. It returns all the printer resolutions in a `PrinterSettings.PrinterResolutionsCollection` object that contains `PrinterResolution` object. The following code snippet reads the printer resolution and adds them to a `ListBox` control. Here `YourPrinterName` is the name of the printer you want to use. If you do not set a printer name, the default printer will be used.

```
PrinterSettings ps = new PrinterSettings();
//Set the printer name
ps.PrinterName = YourPrinterName;
foreach (PrinterResolution pr in ps.PrinterResolutions)
{
    string str = pr.ToString();
}
```

The `PrinterResolution` class, which represents the resolution of a printer, is used by the `PrinterResolutions` and `PrinterResolution` properties of `PrinterSettings` to get and set printer resolutions. Using these two properties, we can get all the printer resolutions available on a printer. We can also use it to set the printing resolution for a page.

The `PrinterResolution` class has three properties: `Kind`, `X`, and `Y`. The `Kind` property is used to determine whether the printer resolution is the `PrinterResolutionKind` enumeration type or `Custom`. If it's `Custom`, the `X` and `Y` properties are used to determine the printer resolution in the horizontal and vertical directions, respectively in dots per inch. If the `Kind` property is not `Custom`, the value of `X` and `Y` each is –1.

The `CanDuplex` property is used to determine whether a printer can print on both sides of a page. If so, we can set the `Duplex` property to `true` to print on both sides of page. The following code snippet determines whether your printer can print on both sides of a page.

```
PrinterSettings ps = new PrinterSettings();
string str = ps.CanDuplex.ToString();
```

The `Duplex` enumeration specifies the printer's duplex settings, which are used by `PrinterSettings`. The members of the `Duplex` enumeration are described in Table 1.

The `Collate` property (both get and set) is used only if we choose to print more than one copy of a document. If the value of `Collate` is `true`, and entire copy of the document will be printed before the next copy is printed. If the value is `false`, all copies of page 1 will be printed, then all copies of page 2, and so on. The following code snipped sets the `Collate` property of `PrinterSettings` to `true`:

```
PrinterSettings ps = new PrinterSettings();
ps.Collate = true;
```


The `Copies` property (both get and set) allows us to enter the number of copies of a document that we want to print. Not all printers support this feature (in which case this setting will be ignored).

The `IsPlotter` property tells us if the printer we're using is actually a plotter that can accept plotter commands. The following code snipped indicates whether the printer is a plotter:

```
PrinterSettings ps = new PrinterSettings();
string str = ps.IsPlotter.ToString();
```

If we print without setting the `PrinterName` property, our printout will be sent to the default printer. The `PrinterName` property allows us to specify a printer to use. The `IsValid` property tells us whether the `PrinterName` value we have selected represents a valid printer on our system. The following code snippets checks if a printer is a vaid printer or not.

```
PrinterSettings ps = new PrinterSettings();
string str = ps.IsValid.ToString();
```

The `MaximumCopies` property determines how many copies the printer can print. Some printers do not allow us to print more than one copy at a time. The following code snippet reads the maximum number of copies that a printer can print.

```
PrinterSettings ps = new PrinterSettings();
int maxcopies = ps.MaximumCopies;
```

The `SupportsColor` property tells us whether the current printer supports printing in color. It will return `true` if the printer supports color printing and `false` otherwise. The following code snippet reads the value of the `SupportsColors` property to find out whether a printer supports colors.

```
PrinterSettings ps = new PrinterSettings();
string str = ps.SupportsColor.ToString();
```

Getting and Setting Printer Paper Size

The `PaperSize` class represents the size of paper used in printing. This class is used by `PrinterSettings` through its `PaperSizes` property to get and set the paper sizes for the printer.

The `PaperSize` class has four properties: `Height, Width, Kind, and PaperName.` The `Height` and `Width` properties are used to get and set the paper's height and width, respectively, in hundredths of an inch. The `PaperName` property is used to get and set the name of the type of paper, but it can be used only when the `Kind` property is set to `Custom`. The `Kind` property returns the type of paper.

The code in Listing 4 reads the `PaperSize` properties.

Listing 4: Reading PaperSize properties

```
PrinterSettings ps = new PrinterSettings();
Console.WriteLine("Paper Sizes");
foreach (PaperSize psize in ps.PaperSizes)
{
    string str1 = psize.Kind.ToString();
    string str2 = psize.PaperName.ToString();
    string str3 = psize.Height.ToString();
    string str4 = psize.Width.ToString();
}
```

Getting and Setting Paper Tray

The `PaperSource` class specifies the paper tray from which the printer retrieves the paper for the current printing task. This class is used by `PritnerSettings` through its `PaperSoruces` property to get and set the paper source trays that a re available on the printer. The `PaperSize` class has two properties: `Kind` and `SourceName`. The `Kind` property returns an enumerated value for the paper source, and `SoruceName` returns the name of the paper source as a string.

The code Listing 5 reads all the paper sources and displays them in a message box.

Listing 5: Reading paper sources

```
PrinterSettings ps = new PrinterSettings();
foreach (PaperSource p in ps.PaperSources)
{
    MessageBox.Show(p.SourceName);
}
```

Printer and Page Properties Application

The basis of the preceding discussion of printer settings, and of printer related classes and their members, lets' write an application using these classes. In this application we will display available printers, the resolutions they support, available paper sizes, and other printer properties. This application will also allow us to set printer properties.

First we create a Windows application and add a combo box, two list boxes, three buttons, six check boxes, and two text boxes to the form. The final form looks like Figure 8. Then we add a reference to the `System.Drawing.Printing` namespace.

Next we write code. The **Available Printers** combo box displays all available installed printers on the machine in the `ListBox` control. We load all installed printers on the form's load event. As Listing 6 shows, we use the `InsalledPrinters` static property of `PrinterSettings`, which returns all installed printer names. We check if the installed printers count is more than 0 and add the installed printers to the combo box.

Listing 6: Reading all available printers

```
private void Form1_Load (object sender, System.EventArgs e)
{
//See if any printers are installed
if (PrinterSettings.InstalledPrinters.Count <= 0)
{
    MessageBox.Show("Printer not found");
    return;
}
//Get all the available printers and add them to the combo box
foreach (String printer in PrinterSettings.InstalledPrinters)
{
    PrintersList.Items.Add(printer.ToString());
}
}
```

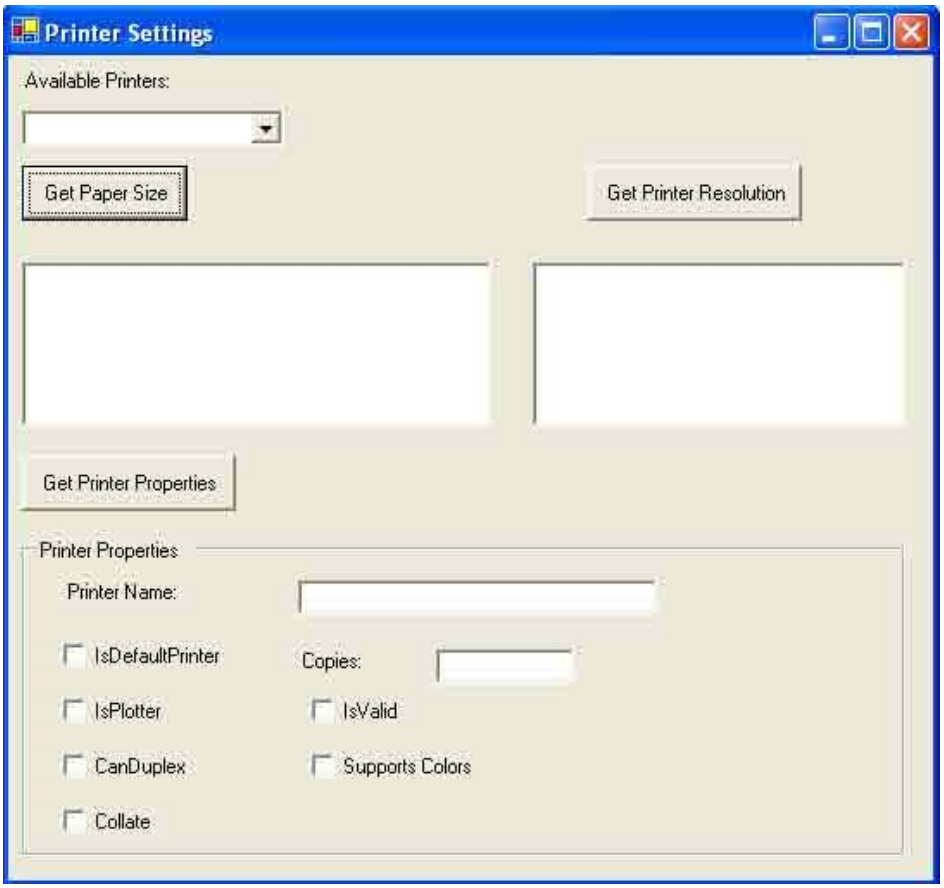

Figure 2: The printer settings form

The **Get Printer Resolution** button returns resolutions supported by a printer selected in `ListBox1`. The `PrinterResolutions` property of `PrinterSettings` returns the printer resolutions supported by the printer Listing 7 reads all available resolutions for the selected printer in `ListBox1` and adds them to `ListBox2`.

Listing 7: Reading printer resolutions

```
private void button2_Click(object sender, System.EventArgs e)
{
    //If no printer is selected
    if (PrintersList.Text == string.Empty)
    {
         MessageBox.Show("Select a printer from the list");
         return;
    }
    //Get the current selected printer form the list of printers
    string str = PrintersList.SelectedItem.ToString();
    //Create a PrinterSettings objects
    PrinterSettings PrSetting = new PrinterSettings();
    //Set the current printer
    PrSetting.PrinterName = str;
    //Read all printer resolutions and add them to the list box
    foreach (PrinterResolution pr
    in PrSetting.PrinterResolutions)
    {
         ResolutionList.Items.Add(pr.ToString());
    }
}
```

The Get Paper Size button returns the available paper sizes. Again we use the `PaperSizes` property of `PrinterSettings`, which returns all available paper sizes. Listing 8 reads all available paper sizes and adds them to the list box.

Listing 8: Reading paper sizes

```
private void button3_Click (object sender, System.EventArgs e)
{
    //If no printer is selected
    if (PrintersList.Text == string.Empty)
    {
         MessageBox.Show("Select a printer from the list");
         return;
    }
    //Create a printer settings
    PrinterSettings prs = new PrinterSettings();
    //Get the current selected printer from the list of printers
    string str = PrintersList.SelectedItem.ToString();
    prs.PrinterName = str;
    //Read paper sizes and add them to the list box
    foreach (PaperSize PrSetting in prs.PaperSizes)
    {
         listBox1.Items.Add(PrSetting.ToString());

    }
}
```

The **Get Printer Properties** buttons gets the printer properties and sets the check boxes and text box controls according to the values returned. The **Get Printer Properties** button click event handler code is given in Listing 9. We read many printer properties that were discussed earlier in this article.

Listing 9: Reading printer properties

```
private void GetProperties_Click (object sender, System.EventArgs e)
{
     //If no printer is selected
     //If no printer is selected
     if (PrintersList.Text == string.Empty)
     {
         MessageBox.Show("Select a printer from the list");
         return;
     }
     PrinterSettings PrSetting = new PrinterSettings();
     string str = PrintersList.SelectedItem.ToString();
     PrSetting.PrinterName = str;
     //Check if the printer is valid
     if (!PrSetting.IsValid)
     {
         MessageBox.Show("Not a valid printer");
         return;
     }
     //Set printer name and copies
     textBox1.Text = PrSetting.PrinterName.ToString();
     textBox2.Text = PrSetting.Copies.ToString();
     //If printer is the default printer
     if (PrSetting.IsDefaultPrinter == true)
     IsDefPrinterChkBox.Checked = true;
     else
     IsDefPrinterChkBox.Checked = false;
     //If printer is a plotter
     if (PrSetting.IsPlotter)
     IsPlotterChkBox.Checked = true;
     else
     IsPlotterChkBox.Checked = false;
     //Duplex printing possible?
     if (PrSetting.CanDuplex)
     CanDuplexChkBox.Checked = true;
     else
     CanDuplexChkBox.Checked = false;
     //Collate?

     if (PrSetting.Collate)
     CollateChkBox.Checked = true;
     else
     CollateChkBox.Checked = false;
     //Printer valid?
     if (PrSetting.IsValid)
     IsValidChkBox.Checked = true;
     else
     IsValidChkBox.Checked = false;
     //Color printer?
     if (PrSetting.SupportsColor)
     SuppColorsChkBox.Checked = true;
     else
     SuppColorsChkBox.Checked = false;
}
```


Now let's run the application. By default, the **Available Printers** combo box displays all available printers. Select a printer from the list, and click the **Get Printer Resolution** button, which display the printer resolutions supported by the selected printer. Also click on the **Get Paper Size** and **Get Printer Properties** buttons. The final output of the application is shown in Figure 3.

We will be using many `PrinterSettings` class members throughout this article.

Understanding PrintDocument and Print Events

So far we have seen how to print simple text and how to read and set printer settings. In the previous sections we saw that in a printing application, we create a `PrintDocument` object, set its printer name, set the printer page event handler, and then call the `Print` method. `PrintDocument` offers more than this. In this section we will cover `PrintDocument` members and print events.

The `PrintDocument` class is used to tell the printing system how printing will take place. Table 1 describes the properties of the `PrintDocument` class.

Besides the properties described in Table 1, `PrintDocument` also provides printing-related methods that invoke print events. These methods are described in Table 2.

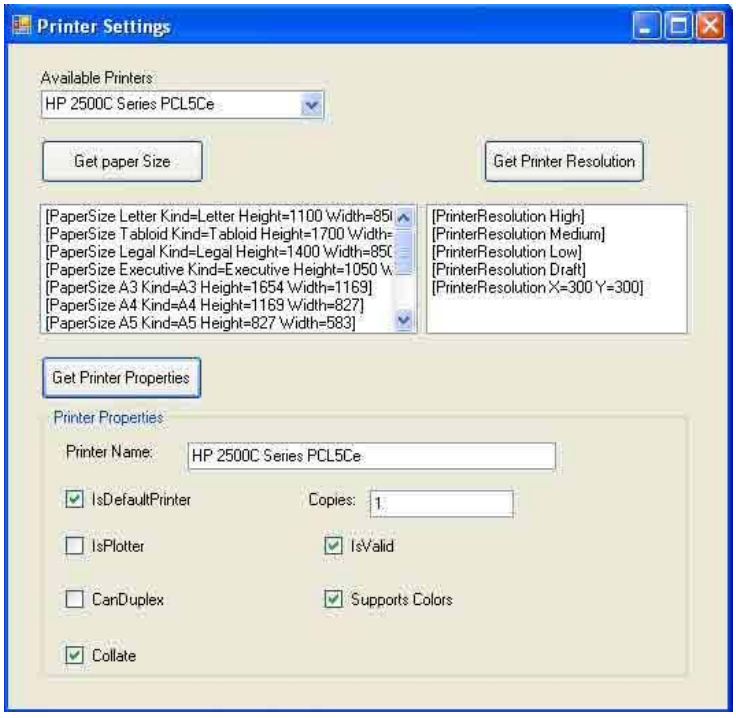

Figure 3: Reading printer properties

Table 1: PrintDocument properties

Property	Description
`DefaultPageSettings`	Represents the page settings using a `PageSettings` object.
`DocumentName`	Returns the name of the document to be displayed in a print status dialog box or printer queue while printing the document.
`PrintController`	Returns the print controller that guides the printing process.
`PrinterSettings`	Returns the printer settings represented by a `PrinterSettings` object.

Table 2: PrintDocument methods

Method	Description
`OnBeginPrint`	Raise the `BeginPrint` event, which is called after the `Print` method and before the first page of the document is printed.
`OnEndPrint`	Raises the `EndPrint` event, which is called when the last page of the document has been printed.
`OnPrintPage`	Raises the `PrintPage` event, which is called before a page prints.
`OnQueryPageSettings`	Raises the `QueryPageSettings` event, which is called immediately before each `PrintPage` event.
`Print`	Starts the document's printing process.

All of these methods allow derived classes to handle the event without attaching a delegate. This is the preferred technique for handling the event in a derived class. We will discuss these methods and their events, and how to handle them, in our examples.

Understanding Print Events

During the printing process, the printing system fires events according to the stage of a printing process. The three common events are `BeginPrint, PrintPage,` and `EndPrint`. As their names indicate, the `BeginPrint` event occurs when the `Print` method is called, and the `EndPrint` event occurs when the last page of the document has been printed. The `PrintPage` event occurs for each page being printed (as in Figure 4) when the `Prin`t method is called and after the `BeginPrint` event has occurred.

Figure 4 shows a flowchart for the print events during a printing process. The `BeginPrint` event is raised after the `Print` method is called. Then the printing process checks if there are any pages. If there are, the `PrintPage` event occurs, which is responsible for the actual printing, and the control goes back to check if there are more pages to print. When all pages are done printing, the `EndPage` event is fired.

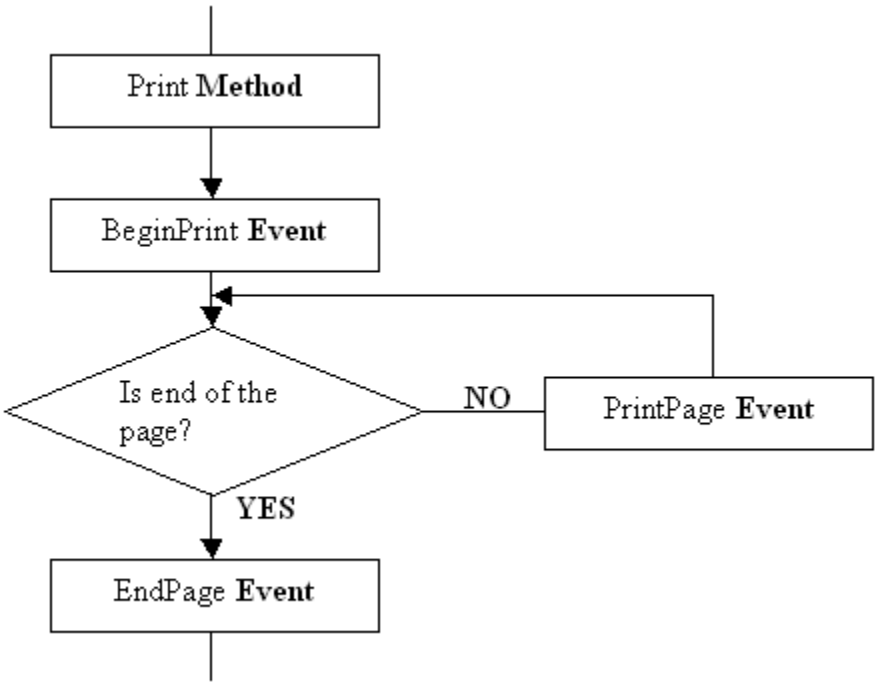

Figure 4: Print events

The `PrintEventArgs` class provides data for `BeginPrint` and `EndPrint` events. This class is inherited from `CancelEventArgs`, which implements a single property called `Cancel`, that indicates if an event should be canceled (in the current .NET Framework release, `PrintEventArgs` is reserved for future use).

The `BeginPrint` event occurs when the `Print` method is called and before the first page prints. `BeginPrint` takes a `PrintEventArgs` object as an argument. This event is the best place to initialize resources. The `PrintEventHandler` method, which is used to handle the event code, is called whenever the `BeginPrint` event occurs.

The `PrintPage` event occurs when the `Print` method is called and before a page prints. When we create a `PrintPageEventHandler` delegate, we identify a method that handles the `PrintPage` event. The event handler is called whenever the `PrintPage` event occurs.

The code snipped that follows creates a `PrintPageEventHandler` delegate, where `pd_PrintPage` is an event handler:

```
PrintDocument pd = new PrintDocument();
pd.PrintPage +=
new PrintPageEventHandler(pd_PrintPage);
```

`PrintPageEventHandler` takes a `PrintPageEventArgs` object as its second argument, which has the six properties described in Table 3.

The following code snippet shows how to get the `Graphics` object from `PrintPageEventArgs`:

```
public void pd_PrintPage (object sender, PrintPageEventArgs ev)
{
//Get the Graphics object attached to PrintPageEventArgs
Graphics g = ev.Graphics;
//Use g now
}
```

The `EndPrint` event occurs when the last page of the document has been printed. It takes a `PrintEventArgs` object as an argument. This is the best place to free your resources. The `PrintEventHandler` method is called whenever the `EndPrint` event occurs and is used to handle the event code.

Now let's write an application that shows how to use these events. We create a Windows application and add a combo box and a button to the form. We set `ComboBox.Name` to `printersList` and the text to the button to **PrintEvent Start**. The final form looks like Figure 5.

Table 3: PrintPageEventArgs properties

Property	Description
`Cancel`	Indicates whether the print jobs should be canceled. Both get and set.
`Graphics`	Returns the `Graphics` object.
`HasMorePages`	Indicates whether an additional page should be printed. Used in multipage document before the `Print` methods is called. Both get and set.
`MarginBounds`	Returns the portion of the page inside the margins.
`PageBounds`	Returns the total area of the page
`PageSettings`	Returns page setting for the current page.

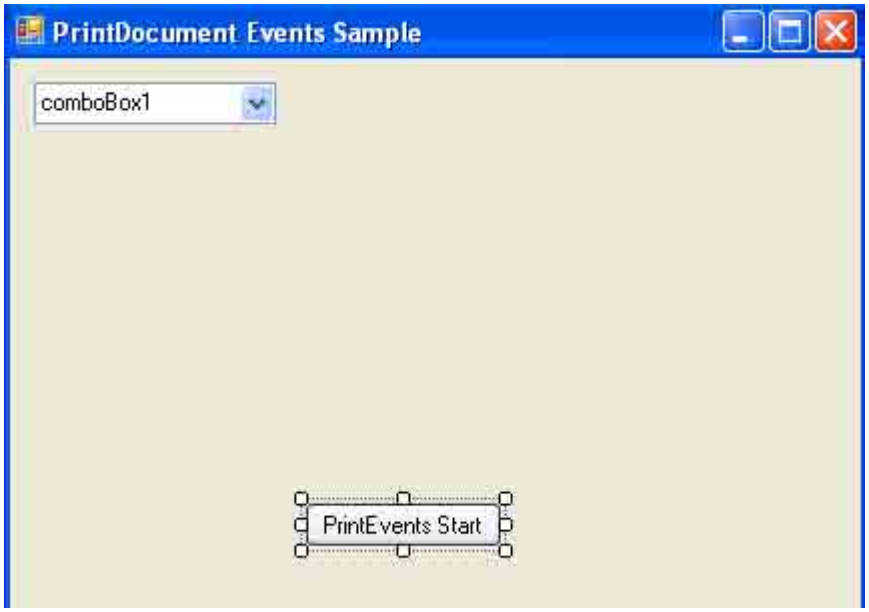

Figure 5: The print events application

Next we add a reference to the `System.Drawing.Printing` namespace as follows:

```
using System.Drawing.Printing;
```

Then we add code on the form's load event handler that adds all installed printers to the combo box (see Listing 10).

Listing 10: Loading all installed printers

```
private void Form1_Load (object sender, System.EventArgs e)
{
    //See if any printers are installed
    if (PrinterSettings.InstalledPrinters.Count <= 0)
    {
        MessageBox.Show("Printer not found! ");
        return;
    }
    //Get all available printers and add them to the combo box
    foreach (String printer in PrinterSettings.InstalledPrinters)
    {
         printersList.Items.Add(printer.ToString());
    }
}
```

Now we write code for the button click event handler. Listing 11 create all three print event handlers, attaches them to a `PrintDocument` object, and calls `PrintDocument's` print methods.

Listing 11: Attaching BeginPrint, EndPrint, and PagePrinteventhandlers

```
private void PrintEvents_Click (object sender, System.EventArgs e)
{
    //Get the selected printer
    string printerName = printersList.SelectedItem.ToString();
    //Create a PrintDocument object and set the current printer
    PrintDocument printDc= new PrintDocument();
    printDc.PrinterSettings.PrinterName = printerName;
    //BeginPrint event
    printDc.BeginPrint += new PrintEventHandler(BgnPrntEventHandler);
    //PrintPage event
    printDc.PrintPage += new PrintPageEventHandler(PrntPgEventHandler);
    //EndPrint event
    printDc.EndPrint +=new PrintEventHandler(EndPrntEventHandler);
    //Print the document
    printDc.Print();
}
```

As state earlier, the `BeginPrint` event handler can be used to initialize resources before printing starts, and the `EndPrint` event handler can be used to free allocated resources. Listing 12 shows all three print event handlers. The `PrintPage` event handler uses the properties for `PrintPageEventArgs` can calls `DrawRectangle` and `FillRectangle` to print the rectangles. This example simply shows how to call these events. You can use the `PrintPage` event handler to draw anything you want to print, as we have seen in previous examples.

Listing 12: The BeginPrint, EndPrint, and PagePrint event handlers

```
public void BgnPrntEventHandler (object sender, PrintEventArgs peaArgs)
{
    //Create a brush and a pen
    redBrush = new SolidBrush (Color.Red);
    bluePen = new Pen (Color.Blue, 3);
}
public void EndPrntEventHandler (object sender, PrintEventArgs peaArgs)
{
    //Release brush and pen objects
    redBrush.Dispose();
    bluePen.Dispose();
}

public void PrntPgEventHandler (object snder, PrintPageEventArgs
ppeArgs)
{
    //Create PrinterSettings object
    PrinterSettings ps = new PrinterSettings();
    //Get Graphics object
    Graphics g = ppeArgs.Graphics;
    //Create PageSettings object
    PageSettings pgSetting = new PageSettings(ps);
    //Set page margins
    ppeArgs.PageSettings.Margins.Left = 50;
    ppeArgs.PageSettings.Margins.Right = 100;
```



```
    ppeArgs.PageSettings.Margins.Top = 50;
    ppeArgs.PageSettings.Margins.Bottom = 100;
    //Create two rectangles0
    Rectangle rect1 = new Rectangle(20, 20, 50, 50);
    Rectangle rect2 = new Rectangle(100, 100, 50, 100);
    //Draw and fill rectangles
    g.DrawRectangle(bluePen, rect1);
    g.FillRectangle (redBrush, rect2);
}
```

As this discussion has shown, the print event can be handy when you need to initialize or free resources.

Printing Text and Text Files

Now let's see how to print text and text files.

Step 1: Create a Windows Forms application
Step 2: Add a reference to the `System.Drawing.Printing` namespace.
Step 3: Add a text box and four buttons to the form. We also change the `Name` and `Text` properties of the buttons controls. The final form looks like Figure 6. As you might guess, the **Browse Text File** button allows us to browse for text files.

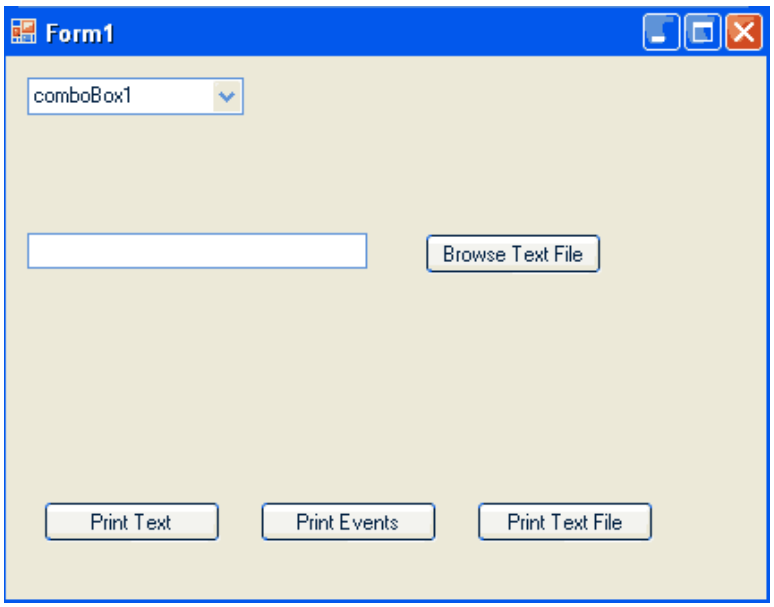

Figure 6: The form with text file printing options

The code for the **Browse Text File** button is given in Listing 13. This button allows you to browse a file and adds the selected file name to the text box. Clicking the **Print Text File** button prints the selected text file. We use an `OpenFileDialog` object to open a text file and set `textBox1.Text` as the selected file name. The functionality of the **Print Text** and **Print Events** buttons is obvious.

Listing 13: The Browse Text File button click event handler

```
private void BrowseBtn_Click (object sender, System.EventArgs e)
{
    //Create an OpenFileDialog object
    OpenFileDialog FileDlg = new OpenFileDialog();
    //Set its properties
    FileDlg.Title = "C# Corner Open File Dialog";
    FileDlg.InitialDirectory = @"C:\ ";
    FileDlg.Filter = "Text files (*.txt | .txt | All files (*.*) |
*.*";
    FileDlg.FilterIndex = 2;
    FileDlg.RestoreDirectory = true;
    //Show dialog and set the selected file name
    //as the text of the text box
    if (FileDlg.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = FileDlg.FileName;
    }
}
```

Now let's add code for the **Print Text File** button click. First we add tow private variable to the application as follows:

```
private Font verdana10Font;
private StreamReader reader;
```

Then we proceed as shown in Listing 14. The code is pretty simple. First we make sure that the user has selected a file name. Then we create a `StreamReader` object and read the file by passing the file name as the only argument. Next we create a font with font family Verdana and size 10. After that we create a `PrintDocument` object, and a `PrintPage` event handler, and call the `Print` method. The rest is done by the `PrintPage` event handler.

NOTE

The `StreamReader` class is defined in the `System.IO` namespace.

Listing 14: The Print Text File button click event handler

```
private void PrintTextFile_Click (object sender, System.EventArgs e)
{
    //Get the file name
    string filename = textBox1.Text.ToString();
    //check if it's not empty
    if (filename.Equals(string.Empty))
    {
        MessageBox.Show("Enter a valid file name");
        textBox1.Focus();
        return;
    }
    //Create a StreamReader object
```



```
    reader = new StreamReader(filename);
    //Create a Verdana font with size 10
    verdana10Font = new Font("Verdana", 10);
    //Create a PrintDocument object
    PrintDocument PrintDoc= new PrintDocument();
    //Add PrintPage event handler
    PrintDoc.PrintPage += new PrintPageEventHandler
    (this.PrintTextFileHandler);
    //Call Print Method
    PrintDoc.Print();
    //Close the reader
    if (reader != null)
    reader.Close();
}
```

The code for the `PrintPage` event handler `PrintTextFileHandler` is given in Listing 15. Here we read one line at a time from the text file, using the `StreamReader.ReadLine` method, and call `DrawString`, which prints each line until we reach the end of the file. To give the text a defined size, we use the `verdana10Font.GetHegiht` method.

Listing 15: Adding a print event handler

```
private void PrintTextFileHandler (object sender, PrintPageEventArgs
ppeArgs)
{
    //Get the Graphics object
    Graphics g = ppeArgs.Graphics;
    float linesPerPage = 0;
    float yPos = 0;
    int count = 0;
    //Read margins from PrintPageEventArgs
    float leftMargin = ppeArgs.MarginBounds.Left;
    float topMargin = ppeArgs.MarginBounds.Top;
    string line = null;
    //Calculate the lines per page on the basis of the height of the
page and the height of the font
    linesPerPage = ppeArgs.MarginBounds.Height /
    verdana10Font.GetHeight(g);
    //Now read lines one by one, using StreamReader
    while (count < linesPerPage &&
    ((line = reader.ReadLine()) != null))
    {
        //Calculate the starting position
        yPos = topMargin + (count *
        verdana10Font.GetHeight(g));
        //Draw text
        g.DrawString(line, verdana10Font, Brushes.Black,
        leftMargin, yPos, new StringFormat());
        //Move to next line
        count++;
    }
    //If PrintPageEventArgs has more pages to print
    if (line != null)
```



```
    ppeArgs.HasMorePages = true;
    else
    ppeArgs.HasMorePages = false;
}
```

You should be able to add code for the **Print Text** and **Print Events** buttons yourself. Their functionality should be obvious.

Now run the application, browse a text file, and hit the **Print Text File** button, and you should be all set.

Printing Graphics Shapes and Images

Printing graphics shapes such as rectangles, ellipses, lines, paths, and polygons is as simple as drawing them. In this section, we will create an application that shows how to print simple graphics shapes such as lines, curves, rectangles, and images.

We create a Windows application and add a main menu to the form. We add four menu items to the main menu. The final form looks like Figure 7. As you might guess, the **Draw Shapes** and **View Image** menu items will draw graphics objects and show an image, respectively. The **Print Image** and **Print Shapes** menu items will print the image and the graphics items, respectively.

The next step is to add a reference to the `System.Drawing.Printing` namespace.

Printing Graphics Shapes

Now let's add code on Draw and Print menu items. On **Draw Shapes** menu item, we add code listed in Listing 16. This menu item draws two lines, a rectangle, and an ellipse. First we create a `Graphics` object using the `Form.CreateGraphics` method and call the `DrawLine, DrawRrctangle,` and `FillEllipse` methods.

Listing 16: Drawing graphics items

```
private void DrawItems_Click (object sender, System.EventArgs e)
{
    //Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    //Draw graphics items
    g.DrawLine(Pens.Blue, 10, 30, 10, 100);
    g.DrawLine(Pens.Blue, 10, 30, 100, 30);
    g.DrawRectangle(Pens.Red, 30, 50, 200, 180);
    g.FillRectangle(Brushes.BlueViolet, 70, 90, 120, 100);
    //Dispose of object
    g.Dispose();
}
```


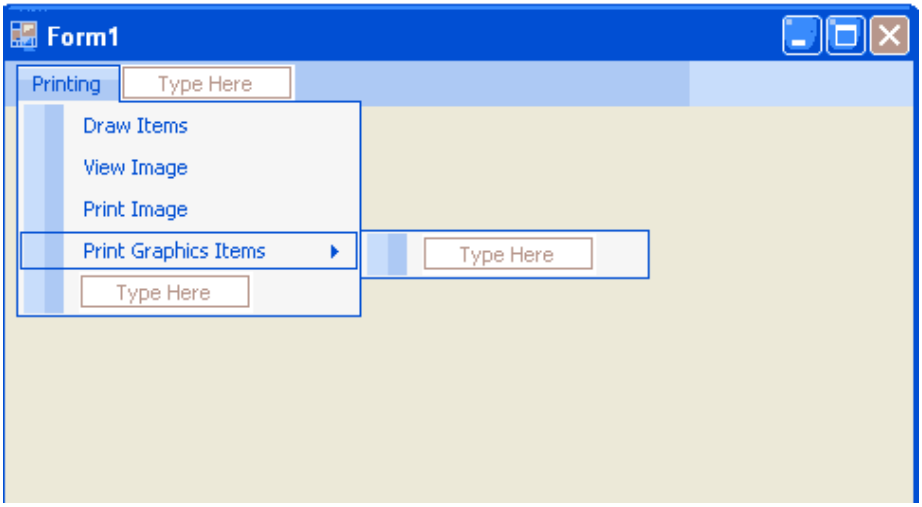

Figure 7: A graphics-printing application

Figure 8 shows the output from Listing 16.

Now let's write code for **Print Shapes**. We want to print the output shown in Figure 8. We create a `PrintDocument` object, and add a `PrintPage` event handler, and call the `Print` method. The `PrintPage` event handler draws the graphics items.

Listing 17 contains two methods. The `PrintGraphicsItems_Click` method is a menu click event handler that creates a `PrintDocument` object, sets its `PrintPage` event, and calls the `Print` method. The second method `PrintGraphicsItemsHandler`, simply calls the draw and fill methods of `PrintPageEventArgs.Graphics`.

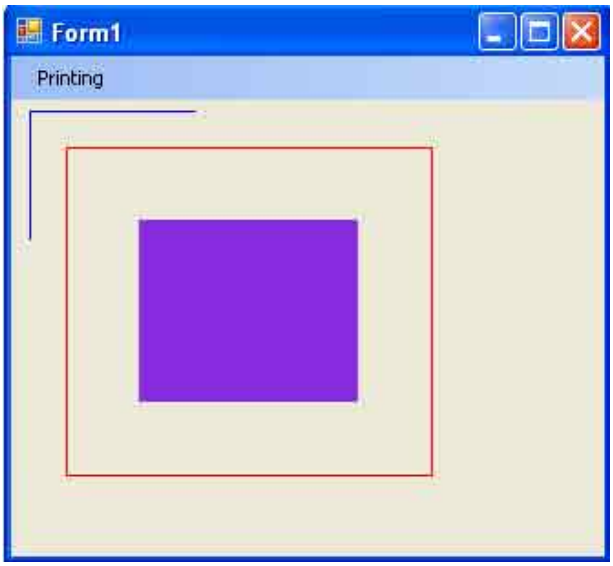

Figure 8 Drawing graphics shapes

Listing 17: Printing shapes

```
private void PrintGraphicsItems_Click (object sender, System.EventArgs
e)
{
    //Create a PrintDocument object
    PrintDocument printDc = new PrintDocument();
    //Add PrintPage event handler
    printDc.PrintPage += new PrintPageEventHandler
    (this.PrintGraphicsItemsHandler);
    //Print
    printDc.Print();
}
private void PrintGraphicsItemsHandler(object sender,
PrintPageEventArgs ppeArgs)
{
    //Create a printer Graphics object
    Graphics g = ppeArgs.Graphics;
    //Draw graphics items
    g.DrawLine(Pens.Blue, 10, 10, 10, 100);
    g.DrawLine(Pens.Blue, 10, 10, 100, 10);
    g.DrawRectangle(Pens.Yellow, 20, 20, 200, 200);
    g.FillEllipse(Brushes.Gray, 40, 40, 100, 100);
}
```

Printing Images

Similarly, the `DrawImage` method of `PrintPageEventArgs.Graphics` prints an image to the printer, which then prints that image onto paper.

Before we add code for the **View Image** menu item, we need to add two application scope variables as follows:

```
private Image curImage = null;
private string curFileName = null;
```

View Image lets us browse for an image and then draws it on the form. As Listing 18 shows, we create a `Graphics` object using `Form.CreateGraphics`. Then we use `OpenFileDialog` to browse files on the system. Once a file has been selected, we create the Image object by using `Image.FromFile`, which takes the file name as its only parameter. Finally, we use `DrawImage` to draw the image.

Listing 18: Viewing an image

```
private void ViewImage_Click (object sender, System.EventArgs e)
{
    //Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    //Call OpenFileDialog, which allows to browse images
    OpenFileDialog OpnDlg = new OpenFileDialog();
    OpnDlg.Filter = "All Image files | *.bmp; *.gif; *.jpg; *.ico; " +
```



```
"*.emf, .wmf | Bitmap files (.bmp; *.gif; *.jpg; " +
"*.ico) | *.bmp; *.gif; *.jpg; *.ico |" +
"Meta Files (*.emf; *.wmf) | *.emf; *.wmf";
    string filter = OpnDlg.Filter;
    //Set InitialDirectory, Title, and ShowHelp properties
    OpnDlg.InitialDirectory =
    Environment.CurrentDirectory;
    OpnDlg.Title = "Open Image File";
    OpnDlg.ShowHelp = true;
    //If OpenFileDialog is OK
    if (OpnDlg.ShowDialog() == DialogResult.OK)
    {
        //Get the file name
        CurrentFName = OpnDlg.FileName;
        //Create an Image object from file name
CurrentImg = Image.FromFile(CurrentFName);
    }
    if (CurrentImg != null)
    {
        //Draw image using the DrawImage method
        g.DrawImage(CurrentImg, AutoScrollPosition.X,
AutoScrollPosition.Y,CurrentImg.Width, CurrentImg.Height);
    }
    //Dispose of object
    g.Dispose();
}
```

Now we run the application and select an image. Figure 9 shows the output.

Now let's write a **Print Image** item click handler. This option prints an image that we're currently viewing on the form. As in the previous example, we create a `PrintDocument`, add a `PrintPage` event handler, and call the `Print` method. This time, however, instead of using the `DrawRectangle` and `DrawLine` methods, we us the `DrawImage` method, which draws the image.

As Listing 19 shows, our code creates a `PrintDocument` object, sets the `PrintPage` event of `PrintDocument` and the `PrintPage` event handler, and calls `PrintDocument.Print`. The `PrintPage` event handler calls `DrawImage`.

Figure 9: Viewing an image

Listing 19: Printing an image

```
private void PrintImage_Click (object sender, System.EventArgs e)
{
    //Create a PrintDocument object
    PrintDocument printDc = new PrintDocument();
    //Add the PrintPage event handler
    printDc.PrintPage += new PrintPageEventHandler
    (this.PrintImageHandler);
    //Print
    printDc.Print();
}

private void PrintImageHandler(object sender, PrintPageEventArgs
ppeArgs)
{
    //Get the Graphics object from PrintPageEventArgs
    Graphics g = ppeArgs.Graphics;
    //If Graphics object exists
    if (CurrentImg != null)
    {
        //Draw image using the DrawImage method
        g.DrawImage(CurrentImg, 0, 0,
CurrentImg.Width, CurrentImg.Height);
    }
}
```

If we run the application, open and view a file, and click the **Print Image** menu item, we get a printout that looks like Figure 9.

Understanding Print Dialogs

Print Dialog controls are defined in the `System.Windows.Forms` namespace. Before using Print dialog controls, you must import this namespace. In this section, we will talk about following print dialog controls:

- `PrintDialog`
- `PrintPriviewDialog`
- `PrintPriviewControl`
- `PageSetupDialog`

The PrintDialog Control

The `PrintDialog` class represents the `PrintDialog` control in the .NET Framework library. This class represents a standard Windows printer dialog, which allows the user to select a printer and choose which portions of the document to print. Table 4 describes the `PrintDialog` class properties. By default, all of these properties are `false` when a `PrintDialog` object is created, and all the properties have both get and set options.

Beside the properties defined in Table 4, `PrintDialog` has on method called `Reset`. This method resets all options, the last selected printer, and the page settings to their default values.

Listing 20 creates a `PrintDialog` object, sets it properties, calls `ShowDialog` and prints the document.

Listing 20: Create and using the PrintDialog control

```
PrintDialog printDlg = new PrintDialog();
PrintDocument printDoc = new PrintDocument();
printDoc.DocumentName = "Print Document";
printDlg.Document = printDoc;
printDlg.AllowSelection = true;
printDlg.AllowSomePages = true;
//Call ShowDialog
if (printDlg.ShowDialog() == DialogResult.OK)
printDoc.Print();
```

Table 4: PrintDialog properties

Property	Description
`AllowSelection`	Indicates whether the **From... To.., Page** option button is enabled.
`AllowSomePages`	Indicates whether the **Pages** option button is enabled.
`Document`	Identifies the `PrintDocument` object used to obtain printer settings.

Property	Description
PrinterSettings	Identifies the printer settings that the dialog box modifies.
PrintToFile	Indicates whether the **Print** to file check box is checked.
ShowHelp	Indicates whether the **Help** button is displayed.
ShowNetwork	Indicates whether the **Network** button is displayed.

The PageSetupDialog Control

The PageSetupDialog class represents the PageSetupDialog control in the .NET Framework library. This class represents a standard Windows page setup dialog that allows users to manipulate page settings, including margins and paper orientation. Users can also set a PageSettings object through PageSetupDialog's PageSettings property. Table 5 describes the properties of the PageSetupDialog class. All of these properties have both get the set options.

As with PrintDialog, the PageSetupdialog class has a Reset method that resets all the default values for the dialog.

Listing 21 creates a PageSetupDialog object, sets its properties, calls ShowDialog, and prints the document.

Table 5: PageSetupDialog properties

Property	Description
AllowMargins	Indicates whether the margins section of the dialog box is enabled. By default, true when a PageSetupDialog object is created.
AllowOrientation	Indicates whether the orientation section of the dialog box (landscape versus portrait) is enabled. By default, true when a PageSetupDialog object is created.
AllowPaper	Indicates whether the paper section of the dialog box (paper size and paper source) is enabled. By default, true when a PageSetupDialog object is created.
AllowPrinter	Indicates whether the **Printer** button is enabled. By default true when a PageSetupDialog object is created.
Document	Identifies the PrintDocument object from which to get page settings. By default, null when a PageSetupDialog object is created.
MinMargins	Indicates the minimum margins the user is allowed to select, in hundredths of an inch. By default, null when a PageSetupDialog object is created.
PageSettings	Identifies the page settings to modify. By default, null when a PageSetupDialog object is created.

Property	Description
PrinterSettings	Identifies the printer settings that the dialog box will modify when the user clicks the **Printer** button. By default null when a PageSetupDialog object is created.
ShowHelp	Indicates whether the **Help** button is visible. By default, false when a PageSetupDialog object is created.
ShowNetwork	Indicates whether the **Network** button is visible. By default, true when a PageSetupDialog object is created.

Listing 21: Creating and using the PageSetupDialog control

```
setupDlg = new PageSetupDialog ();
printDlg = new PrintDialog ();
printDoc = new PrintDocument ();
printDoc.DocumentName = "Print Document";
//PageSetupDialog settings
setupDlg.Document = printDoc;
setupDlg.AllowMargins = false;
setupDlg.AllowOrientation = false;
setupDlg.AllowPaper = false;
setupDlg.AllowPrinter = false;
setupDlg.Reset();

if (setupDlg.ShowDialog() == DialogResult.OK)
{
    printDoc.DefaultPageSettings =
    setupDlg.PageSettings;
    printDoc.PrinterSettings =
    setupDlg.PrinterSettings;
}
```

The PrintPreviewDialog Control

The PrintePreviewDialog class represents the PrintPreviewDialog control in the .NET Framework library. This class represents a standard Windows print preview dialog, which allows users to preview capabilities before printing. The PrintPreviewDialog class is inherited from the Form class, which means that this dialog contains all the functionality defined in Form, Control, and other base classes.

In addition to the properties provided by the base classes, this class has its own properties. Many of these properties are very common and are provided by many controls. Table 6 describers a few important PrintPreviewDialog class properties. All of these properties have both get and set options.

Listing 22 creates a PrintPreviewDialog object, sets its properties calls ShowDialog, and prints the document.

Listing 22: Create and using the PrintPreviewDialog control

```
//Create a PrintPreviewDialog object
PrintPreviewDialog previewDlg = new PrintPreviewDialog();
//Create a PrintDocument object
PrintDocument printDoc = new PrintDocument();
//Set Document property
previewDlg.Document = printDoc;
previewDlg.WindowState = FormWindowState.Normal;
//show Dialog
previewDlg.ShowDialog();
```

Table 6: Some PrintPreviewDialog properties

Property	Description
`Document`	Identifies the document shown in preview.
`HelpButton`	Indicates whether a help button should be displayed in the caption box of the form. The default value is `false`.
`KeyPreview`	Indicates whether the form will receive key events before the event is passed to the control that has focus. The default value is `false`.
`ShowInTaskbar`	Indicates whether the form is displayed in the Windows taskbar. The default value is `true`.
`TransparencyKey`	Identifies the color that will represent transparent areas of the form.
`UseAntiAlias`	Indicates whether printing uses the anti-aliasing features of the operating system.
`WindowState`	Identifies the form's window state.

Putting Print Dialogs to Work

Now we are going to create a Windows application that will these print dialog controls.

We create a Windows application and add a `MainMenu` control to the form. We also add four menu items and a separator to the `MainMenu` control. The final form looks like Figure 10.

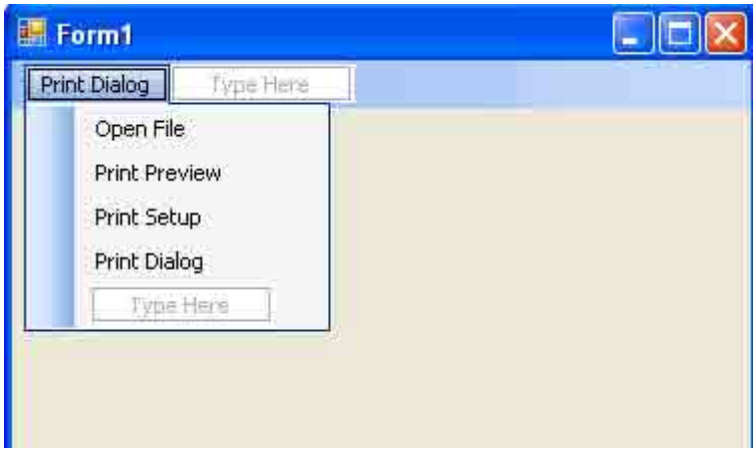

Figure 10: The print dialog application

As usual, our first step is to add some private variable to the project, as follows:

```
//Variable
private Image curImage = null;
private string curFileName = null;
private PrintPreviewDialog previewDlg = null;
private PageSetupDialog setupDlg = null;
private PrintDocument printDoc = null;
private PrintDialog printDlg = null;
//We also add the following namespace to the project:
using System.Drawing.Printing;
using System.Drawing.Imaging;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
```

On our forms' load event, we initialize these dialogs. We also create a `PrintPage` event handler and add it to the `PrintDocument` object, as shown in Listing 23.

Listing 23: Initializing print dialogs

```
private void Form1_Load (object sender, System.EventArgs e)
{
//Create print preview dialog and other dialogs
     pPrViewDlg = new PrintPreviewDialog();
     setupDlg = new PageSetupDialog();
     printDoc = new PrintDocument();
     printDlg = new PrintDialog();
     //Set document name
     printDoc.DocumentName = "Print Document";
     //PrintPreviewDialog settings
     pPrViewDlg.Document = printDoc;
     //PageSetupDialog settings
     setupDlg.Document = printDoc;
     //PrintDialog settings
     printDlg.Document = printDoc;
     printDlg.AllowSelection = true;
```



```
        printDlg.AllowSomePages = true;

        //Create a PrintPage event handler
        printDoc.PrintPage += new PrintPageEventHandler(this.pd_Print);
}
```

Now we add the `PrintPage` event handler, which calls `DrawGraphicsItems` as shown in Listing 24. We pass `PrintPageEventArgs.Graphics` as the only parameter to `DrawGraphicsItems.`

Listing 24: The PrintPage event handler

```
private void pd_Print (object sender, PrintPageEventArgs ppeArgs)
{
    DrawGraphicsItems(ppeArgs.Graphics);
}
```

The `DrawGraphics` Items method draws an image and text on the printer or the form, depending on the `Graphics` object. If we pass `Form.Graphics`, the `DrawGraphicsItems` method will draw graphics objects on the form, but if we pass `PrintPageEventArgs.Graphics`, this method will send drawings to the printer.

The code for the `DrawGraphicsItems` method is given in Listing 25. This method also sets the smoothing mode and text qualities via the `SmoothingMode` and `TextRenderingHint` properties. After that it calls `DrawImage` and `DrawText`.

Listing 25: The DrawGraphicsItems method

```
private void DrawGraphicsItems (Graphics gobj)
{
    //Set text and image quality
    gobj.SmoothingMode =
SmoothingMode.AntiAlias;
    gobj.TextRenderingHint = TextRenderingHint.AntiAlias;
    if (image1!= null)
    {
        //Draw image using the DrawImage method
        gobj.DrawImage(image1, AutoScrollPosition.X,
AutoScrollPosition.Y, image1.Width, image1.Height);
    }
    //Draw a string
    gobj.DrawString("Printing Dialogs Text", new Font("Verdana", 25),
new SolidBrush(Color.Violet), 40, 40);
}
```

There's just one more thing to do before we write the menu item event handlers. We call `DrawGraphicsItems` from the form's paint event handler, as Listing 26 shows. Adding this code will display the drawing on the form.

Listing 26: The form's paint event handler

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    DrawGraphicsItems (e.Graphics);
}
```

Now we can write code for the menu items. The **Open File** menu item just let use browse images and creates an `Image` object by calling the `Image.FromFile` method, as Listing 27 shows.

Listing 27: The Open File menu handler

```
private void OpenFile_Click (object sender, System.EventArgs e)
{
    //Create a Graphics object
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    //Create open file dialog
    OpenFileDialog openDlg = new OpenFileDialog();
    //Set filter as image
    openDlg.Filter =
    "All Image files | *.bmp; *.gif; *.jpg; *.ico; " +
    "*.emf, .wmf | Bitmap Files (*.bmp; *.gif; *.jpg; " +
    "*.ico) | *.bmp; *.gif; *.jpg; *.ico|" +
    "Meta Files (*.emf; *.wmf) | *.emf; *.wmf";
    string filter = openDlg.Filter;
    //Set title and initial directory
    openDlg.InitialDirectory =
    Environment.CurrentDirectory;
    openDlg.Title = "Open Image File";
    openDlg.ShowHelp = true;
    //Show dialog
    if (openDlg.ShowDialog() == DialogResult.OK)
    {
        //Get the file name and create Image object from file
        FileName = openDlg.FileName;
        image1= Image.FromFile(FileName);
    }
    if (image1 != null)
    {
        //Draw image using the DrawImage method
        g.DrawImage(image1, AutoScrollPosition.X,
        AutoScrollPosition.Y,
        image1.Width, image1.Height);
    }
    //Dispose of object
    g.Dispose();
}
```

The code for `PrintPreviewDialog, PageSetupDialog`, and `PrintDialog` is given in Listing 28. We show `PrintDialog` and call its `PrintDocument.Print` method if the user selects **OK** on the print dialog. We set

`PageSetupDialog` page and printer settings when the user selects **OK** on the page setup dialog. For the print preview dialog, we set the `UseAntiAlias` property and call `ShowDialog`.

Listing 28: Print dialogs

```
private void PrintDialog_Click (object sender, System.EventArgs e)
{
    if (printDlg.ShowDialog() == DialogResult.OK)
    printDoc.Print();
}
private void PageSetupDialog_Click (object sender, System.EventArgs e)
{
    if (setupDlg.ShowDialog() == DialogResult.OK)
    {
        printDoc.DefaultPageSettings = setupDlg.PageSettings;
        printDoc.PrinterSettings = setupDlg.PrinterSettings;
    }
}
private void PrintPreview_Click (object sender, System.EventArgs e)
{
    pPrViewDlg.UseAntiAlias = true;
    pPrViewDlg.WindowState = FormWindowState.Normal;
    pPrViewDlg.ShowDialog();
}
```

Now when we run the application and browse an image using the **Open File** menu item, the form looks like Figure 11.

If we click on **Print Preview**, our program will display the print preview dialog, as shown in Figure 12.

As stated earlier, the page setup dialog allows us to set the page properties, including size, sources, orientation, and margins. Clicking on **Print Setup** on the dialog menu brings up the page setup dialog, which is shown in Figure 13.

We can use these dialogs as we would in any other Windows applications.

Figure 11: Viewing an image and text

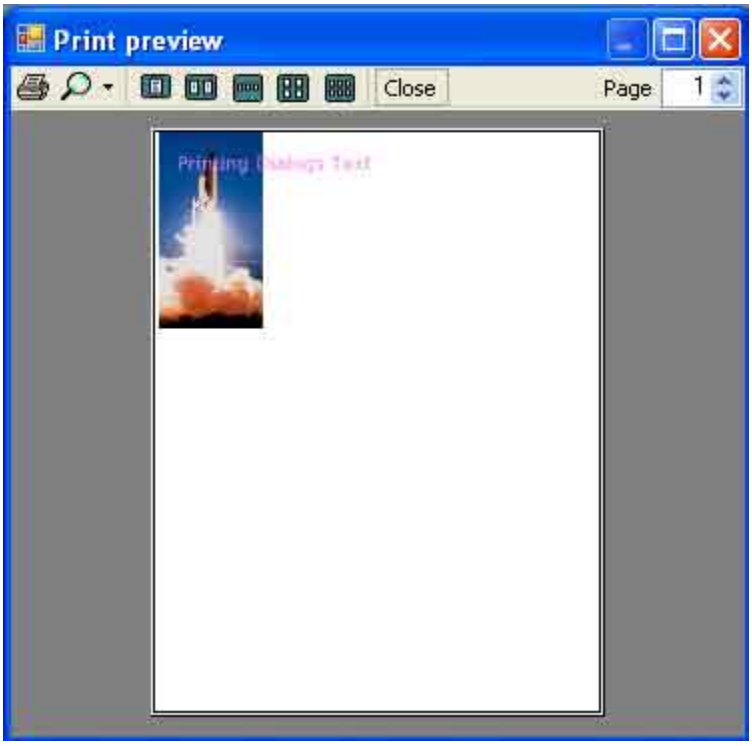

Figure 12: The print preview dialog

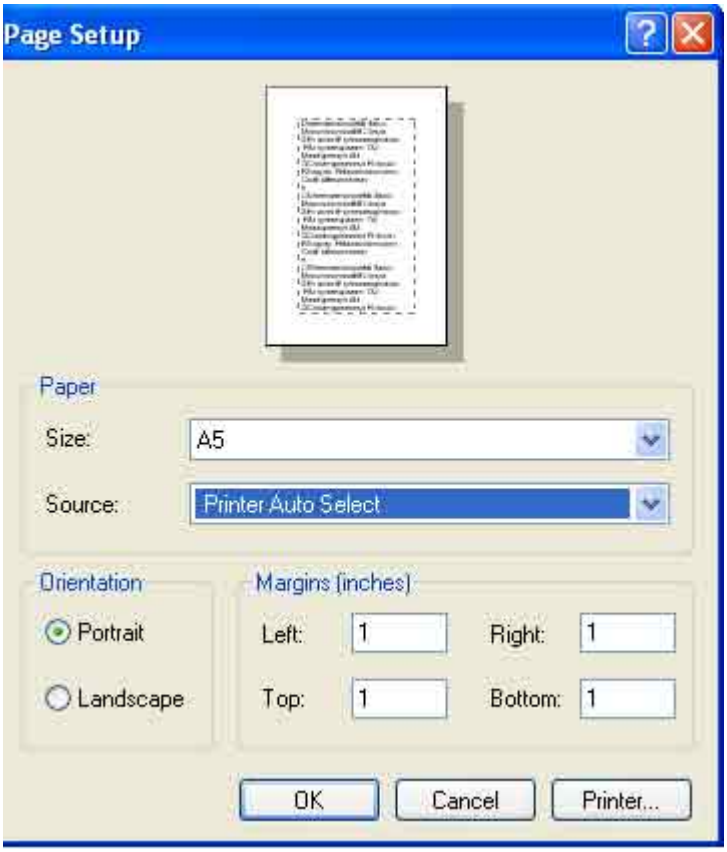

Figure 13: The page setup dialog

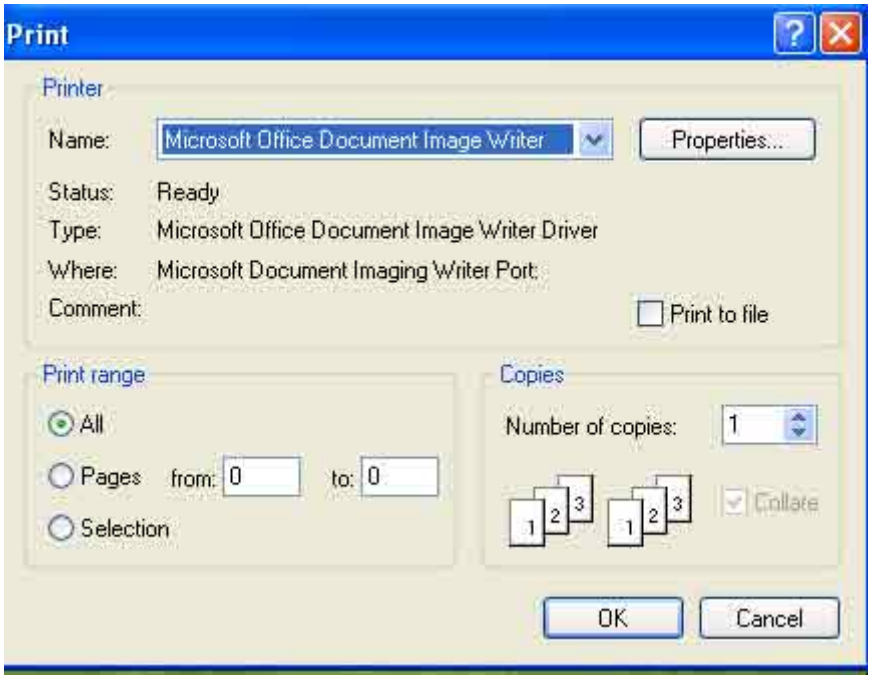

Figure 14: The print dialog

Customizing Page Settings

There are times when we may need custom page settings with custom dialogs. For example, suppose we want to change the text of the dialog or don't want the user to have page selection or anything else that is not available on the default Windows dialogs.

The `System.Drawing.Printing` namespace also defines functionality to manage page settings programmatically.

The PageSettings Class

Page settings are the properties of a page that are being used when a page is printer, including colors, page margins, paper size, page bounds, and page resolution.

The `PageSettings` class represents page settings in the .NET Framework library. This class provides members to specify page settings. It is used by the `Printdocument.DefultPageSettings` property to specify the page settings of a `PrintDocument` object. Table 7 describes the properties of the `PageSettings` class.

Besides the properties described in Table 7, the `PageSettings` class provides three methods: `Clone, CopyToHdevmode,` and `SetHdevmode`. The Clone method simply creates a copy of the `PageSettings` object. `CopyToHdevmode` copies relevant information from the `PageSettings` object from the specified `DEVMODE` structure, and `SetHdevmode` copies relevant information to the `PageSettings` object from the specified `DEVMODE` structure. The `DEVMODE` structure is used by WIN32 programmers.

Page Margins

The `Margins` class represents a page margin in the .NET Framework library. It allows you to get the current page margin settings and set new margin settings. This class has four properties – `Left, Right, Top,` and `Bottom` – which represent the left, right, top, and bottom margins, respectively, in hundredths of an inch. This class is used by the `Margins` property of the `PageSettings` class. We will use this class and its members in our examples.

Table 7: PageSettings properties

Property	Description
`Bounds`	Returns the size of the page.
`Color`	Indicates whether the page should be printed in color. Both get and set. The default is determined by the printer.
`Landscape`	Indicates whether the page is printed in landscape or portrait orientation. Both get and set. The default is determined by the printer.

Property	Description
Margins	Identifies the page margins. Both get and set.
PaperSize	Identifies the paper size. Both get and set.
PaperSource	Identifies the paper source (a printer tray). Both get and set.
PrinterResolution	Identifies the printer resolution for the page. Both get and set.
PrinterSettings	Identifies the printer settings associated with the page. Both get and set.

Creating a Custom Paper Size

As mentioned earlier, the PaperSize class specifies the size and type of paper. You can create you own custom paper sizes. For example, Listing 29 creates a custom paper size with a height of 200 and width of 100.

Listing 29: Creating a custom paper size

```
//Create a custom paper size and add it to the list
PaperSize customPaperSize = new PaperSize();
customPaperSize.PaperName = "Custom Size";
customPaperSize.Height = 200;
customPaperSize.Width = 100;
```

The PaperKind Enumeration

The PaperKind enumeration, as we saw earlier, is used by the Kind property to specify standard paper sizes. This enumeration has over 100 members. Among them are A2, A3, A3Extra, A3ExtraTransverse, A3Rotated, A3Transverse, A4, A5, A6, Custom, DCEnvelope, Executive, InviteEnvelope, ItalyEnvelope, JapanesePostCard, Ledger, Legal, LegalExtra, Letter, LetterExtra, LetterSmall, Standard10x11 (10x14, 10x17, 12x11, 15x11, 9x11), Statement, and Tabloid.

The PaperSourceKind Enumeration

The PaperSourceKind enumeration represents standard paper sources. Table 8 describes the members of the PaperSourceKind enumeration.

Table 8: PaperSoruceKind members

Member	Description
AutomaticFeed	Automatically fed paper
Cassette	A paper cassette
Custom	A printer-specific paper source
Envelope	An Envelope

Member	Description
`FormSource`	The printer's default input bin
`LargeCapacity`	The printer's large-capacity bin
`LargeFormat`	Large-format paper
`Lower`	The lower bin of a printer
`Manual`	Manually fed paper
`ManualFeed`	Manually fed envelope
`Middle`	The middle bin of a printer
`SmallFormat`	Small-format paper
`TractorFeed`	A tractor feed
`Upper`	The upper bin of a printer

Page Settings Example

Now let's see how we can create an application that will allow us to get and set page settings. In this application we will create a custom dialog.

We start by creating a new Windows application in VS .NET. We add some controls to the form, with the result shown in Figure 15. The **Available Printers** combo box displays all available printers. The **Size** and **Source** combo boxes display paper sizes and sources, respectively. The **Paper Orientation** section indicates whether paper is oriented in landscape mode or portrait mode. The **Paper Margins** text boxes obviously represent left, right, top and bottom margins. The `Bounds` property is represented by the **Bounds (Rectangle**) text box. The **Color Printing** check box indicates whether the printer supports color printing. The **Set Properties** button allows us to enter new values in the controls.

The form's load event (see Listing 30) loads all the required `PageSettings`-related settings using the `LoadPrinters, LoadPaperSizes, LoadPaperSources,` and `ReadOtherSettings` methods.

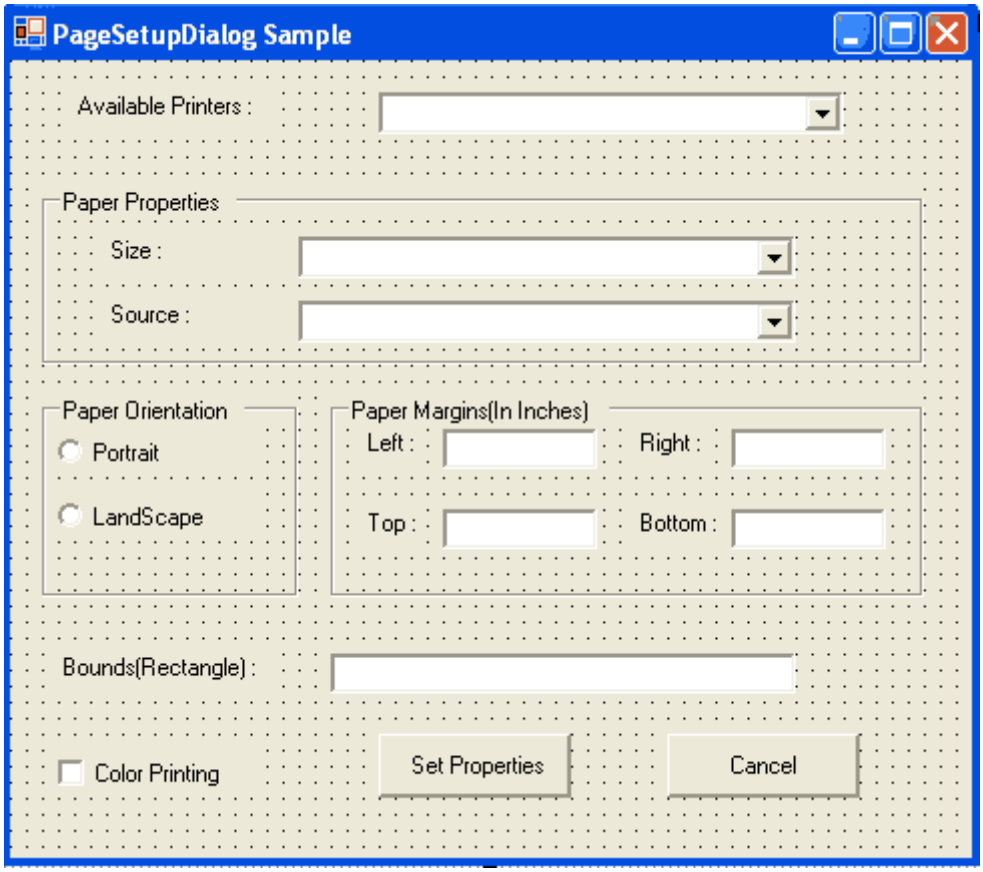

Figure 15: The custom page settings dialog

Listing 30: The form's load event handler

```
private void Form1_Load (object sender, System.EventArgs e)
{
//Load all available printers
LoadPrinters();
//Load paper sizes
LoadPaperSizes();
//Load paper sources
LoadPaperSources();
//Load other settings
ReadOtherSettings();
}
```

The `LoadPrinters, LoadPaperSizes, LoadPaperSources,` and `ReadOtherSettings` methods are used to load printers, paper sizes, paper sources, and other properties, respectively. The `LoadPrinters` method is given in Listing 31. We simply read the `InstalledPrinters` property of `PrinterSettings` and add printer to the `printersList` combo box.

Listing 31: Loading printers

```
private void LoadPrinters()
{
    //Load all avaiable printers
    foreach (String printer in
    PrinterSettings.InstalledPrinters)
    {
        printersList.Items.Add(printer.ToString());
    }
    printersList.Select(0, 1);
}
```

The `LoadPaperSizes` method (see Listing 32), loads all available paper sizes to the combo box. We read the `PaperSizes` property of `PrinterSettings` and add the paper type to the combo box. Then we create a custom paper size and add this to the combo box as well. This example will give you an idea of how to create your own custom paper sizes.

Listing 32: Loading paper sizes

```
private void LoadPaperSizes()
{
    PaperSizeCombo.DisplayMember = "PaperName";
    PrinterSettings PrSetting = new PrinterSettings();
    //Get all paper sizes and add them to the combo box list
    foreach (PaperSize size in PrSetting.PaperSizes)
    {
        PaperSizeCombo.Items.Add(size.Kind.ToString());
        //You can even read the paper name and all PaperSize properties
by uncommenting these two lines:
        PaperSizeCombo.Items.Add(size.PaperName.ToString());
        PaperSizeCombo.Items.Add(size.ToString());
    }
    //Create a custom paper size and add it to the list
    PaperSize customPaperSize = new PaperSize("Custom Size", 50, 100);
    //You can also change properties
    customPaperSize.PaperName = "New Custo, Size";
    customPaperSize.Height = 200;
    customPaperSize.Width = 100;
    //Don't assign the Kind property. It's read-only.
    //customPaperSize.Kind = PaperKind.A4;
    //Add custom size
    PaperSizeCombo.Items.Add(customPaperSize);
}
```

The `LoadPaperSources` methods (see Listing 33), reads all available paper sources and adds them to the `PaperSourceCombo combo`box. We use the `PaperSoruces` property of `PrinterSettings` to read the paper sources.

Listing 33: Loading paper sources

```
private void LoadPaperSources()
{
    PrinterSettings PrSetting = new PrinterSettings();
    PaperSourceCombo.DisplayMember = "SourceName";
    //Add all paper sources to the combo box
    foreach (PaperSource source in PrSetting.PaperSources)
    {
        PaperSourceCombo.Items.Add(source.ToString());
        //You can even add Kind and SourceName
        //by uncommenting the following two lines:
        //PaperSourceCombo.Items.Add
        //(Source.Kind.ToString());
        //PaperSourceCombo.Items.Add
        //(source.SourceName.ToString());
    }
}
```

The last method, `ReadOtherSettnigs`, reads other properties of a printer, such as whether it supports color, margins, and bounds. Listing 34 shows the `ReadOtherSettings` method.

Listing 34: Loading other properties of a printer

```
private void ReadOtherSettings()
{
     //Set other default properties
     PrinterSettings PrSetting = new PrinterSettings();
     PageSettings pgSettings =
     PrSetting.DefaultPageSettings;
     //Color printing
     if (pgSettings.Color)
     ColorPrintingBox.Checked = true;
     else
     ColorPrintingBox.Checked = false;

     //Page margins
     leftMarginBox.Text = pgSettings.Bounds.Left.ToString();
     rightMarginBox.Text = pgSettings.Bounds.Right.ToString();
     topMarginBox.Text = pgSettings.Bounds.Top.ToString();
     bottomMarginBox.Text = pgSettings.Bounds.Bottom.ToString();

     //Landscape or portrait
     if (pgSettings.Landscape)
     landscapeButton.Checked = true;
     else
     portraitButton.Checked = true;
     //Bounds
     boundsTextBox.Text =
     pgSettings.Bounds.ToString();
}
```


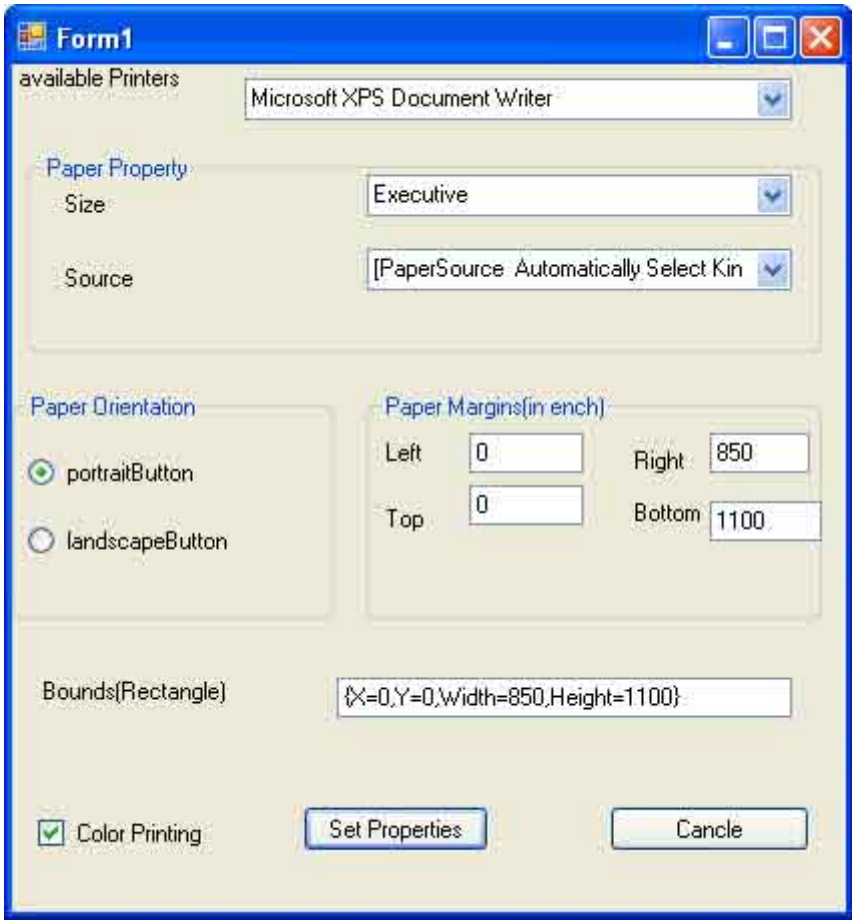

Figure 16: The PageSetupDialog sample in action

Now if we run the application, its form looks like Figure 16. Each of the Windows controls displays its intended property.

Finally, we want to save settings through the **Set Properties** button click and write code for a **Cancel** button. On the **Set Properties** button click, we set the properties using `PrinterSettings`. Make sure printer is available in the **Available Printers** combo box. The **Cancel** button simply closes the dialog.

The code for the **Set Properties** and **Cancel** button click event handlers is given in Listing 35, in which we set the page settings, color, and landscape properties of a page.

Listing 35: Saving paper settings

```
private void SetPropertiesBtn_click (object sender, System.EventArgs e)
{
    //Set other default properties
    PrinterSettings PrSetting = new PrinterSettings();
    PageSettings pgSettings =
    PrSetting.DefaultPageSettings;
    //Color printing?
```



```
    if (ColorPrintingBox.Checked)
    pgSettings.Color = true;
    else
    pgSettings.Color = false;

     //Landscape or portrait?

    if (landscapeButton.Checked)
    pgSettings.Landscape = true;
    else
    pgSettings.Landscape = false;
}
private void button2_Click(object sender, EventArgs e)
{
    this.Close();
}
```

The preceding discussion should enable you to customize page settings in the way that you want, instead of using the standard page settings dialog provided in the `PageSettingsDialog` class.

NOTE

Even though the printing facility defined in the `System.Drawing.Printing` namespace allows developers to customize the standard Windows dialogs, I recommend that you use the standard Windows dialogs unless you can't live without customizing them.

The PrintRange Enumeration

The `PrintRange` enumeration is used to specify the part of a document to print. This enumeration is used by the `PrinterSettings` and `PrintDialog` classes. Table 9 describes the members of the `PrintRange` enumeration.

You can use the `PrintRange` property of the `PrinterSettings` object to set the print range. Here's an example of code that does this:

```
PrinterSettings.PrintRange = PrintRange.SomePages;
```

Table 9: PrintRange members

Member	Description
`AllPages`	All pages are printed.
`Selection`	The selected pages are printed.
`SomePages`	The pages between FromPage and ToPage are printed.

Printing Multiple Page Documents

So far we have discussed printing only an image or a single-page file. Printing multipage files is another important part of printing functionality that developers may need to implement when writing printer applications. Unfortunately, the .NET Framework does not keep track of page numbers for you, but it provides enough support for you to keep track of the current page, the total number of pates, the last page, and a particular page number. Basically, when printing a multipage document, you need to find out the total number of pages and print them from first to last. You can also specify a particular page number. If you are using the default Windows printing dialog, then you don't have to worry about it because you can specify the pages in the dialog, and the framework takes care of this for you.

To demonstrate how to do this, our next programs produces a useful printout showing all the fonts installed on your computer. This program is a useful tool for demonstrating the calculation of how many pages to print when you're using graphical commands to print.

We will use the `PrintPreview` facility to display the output in case you don't have access to a printer and how far down the page we are. If we're going to go over the end of the page, we drop out of the `pd_PrintPage` event handler and set `ev.HasMorePages` to `true` to indicate that we have another page to print.

To see this functionality in action, let's create a Windows application and add a menu with three menu items and a `RichTextBox` control to the form. The final form is shown in Figure 17.

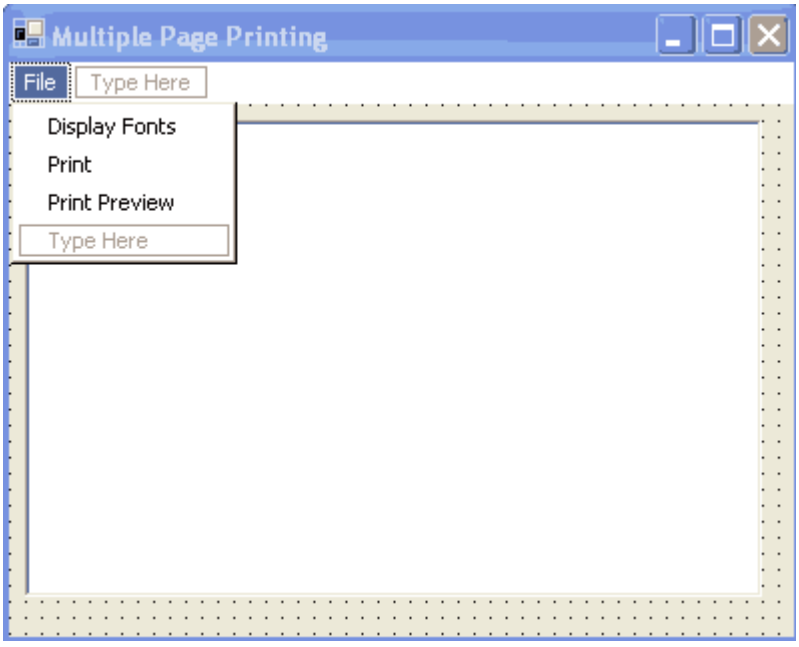

Figure 17: A form for printing multiple pages

The **Display Fonts** menu display available fonts on the machine. Before we add code to this menu, we add the following variables:

```
private int fontcount;
private int fontposition = 1;
private float ypos = 1;
private PrintPreviewDialog previewDlg = null;
```

The code for the **Display Fonts** menu click is given in Listing 36. Here we read installed fonts on the system and display them in the rich text box. We use `InstalledFontCollection` to read all installed fonts on a machine. Then we use the `InstalledFontCollection.Families` property and make a loop to read all the font families. We also check if these families support different styles, including regular, bold, italic, and underline, and wee add some text to the rich text box with the current font.

Listing 36: Displaying fonts

```
private void DisplayFonts_Click_1 (object sender, System.EventArgs e)
{
    //Create InstalledFontCollection objects
    InstalledFontCollection insFontColl = new
InstalledFontCollection();

    //Get font families
    FontFamily[] ffs = insFontColl.Families;
    Font f;
    //Make sure rich text box is empty
    richTextBox1.Clear();
    //Read font families one by one,
    //set font to some text,
    //and add text to the text box
    foreach (FontFamily FontFmly in ffs)
    {
        if (FontFmly.IsStyleAvailable(FontStyle.Regular))
        f = new Font(FontFmly.GetName(1),12, FontStyle.Regular);
        else if (FontFmly.IsStyleAvailable(FontStyle.Bold))
        f = new Font(FontFmly.GetName(1),12, FontStyle.Bold);
        else if (FontFmly.IsStyleAvailable(FontStyle.Italic))
        f = new Font(FontFmly.GetName(1),12, FontStyle.Italic);
        else
        f = new Font(FontFmly.GetName(1),12, FontStyle.Underline);
        richTextBox1.SelectionFont = f;
        richTextBox1.AppendText(FontFmly.GetName(1) + "\r\n");
        richTextBox1.SelectionFont = f;
        richTextBox1.AppendText("abcdefghijklmnopqrstuvwxyz\r\n");
        richTextBox1.SelectionFont = f;
        richTextBox1.AppendText("ABCDEFGHIJKLMNOPQRSTUVWXYZ\r\n");
        richTextBox1.AppendText("==========================\r\n");
    }
}
```


The code for the **Print Preview** and **Print** menu items is given in Listing 37. This code should look familiar to you. We simply create `PrintDocument` and `PrintPreviewDialog` objects, set their properties, add a print-page event handler, and call the `Print` and `Show` methods.

Listing 37: The Print Preview and Print menu items

```
private void PrintPreviewMenuClick (object sender, System.EventArgs e)
{

    //Create a PrintPreviewDialog object
    previewDlg = new PrintPreviewDialog();
    //Create a PrintDocument object
    PrintDocument PrintDc = new PrintDocument();
    //Add print-page event handler
    PrintDc.PrintPage +=new PrintPageEventHandler(PrintDc_PrintPage);
    //Set Document property of PrintPreviewDialog
    previewDlg.Document = PrintDc;
    //Display dialog
    previewDlg.Show();
}

private void PrintMenuClick (object sender, System.EventArgs e)
{
    //Create a PrintPreviewDialog object
    previewDlg = new PrintPreviewDialog();
    //Create a PrintDocument object
    PrintDocument PrintDc = new PrintDocument();
    //Add print-page event handler
    PrintDc.PrintPage +=
    new PrintPageEventHandler(PrintDc_PrintPage);
    //Print
    PrintDc.Print();
}
```

The print-page event handler, `pd_PrintPage`, is given in Listing 38. We print fonts using `DrawString`, and we set `PrintPageEventArgs.HasMorePages` to `true`. To make sure the text fits, we increase the **y**-position by 60 units.

Listing 38: The print-page event handler

```
public void pd_PrintPage (object sender, PrintPageEventArgs ev)
{
    ypos = 1;
    float pageheight = ev.MarginBounds.Height;
    //Create a Graphics object
    Graphics g = ev.Graphics;
    //Get installed fonts
    InstalledFontCollection insFontColl = new
InstalledFontCollection();

    //Get font families
    FontFamily[] ffs = insFontColl.Families;
```



```
    //Draw string on the paper
    while (ypos + 60 < pageheight && fontposition < ffs.GetLength(0))
    {
        //Get the font name
        Font f = new Font(ffs[fontposition].GetName(0), 25);
        //Draw string
        g.DrawString(ffs[fontposition].GetName(0), f, new
SolidBrush(Color.Black), 1, ypos);
        fontposition = fontposition + 1;
        ypos = ypos + 60;
    }
    if (fontposition < ffs.GetLength(0))
    {
        //Has more pages??
        ev.HasMorePages = true;
    }
}
```

That's it. If we run the program, the **Print** menu prints multiple pages, and the **Print Preview** menu shows the print preview on two pages (see Figure 18).

As you can see, it's pretty easy to create multipage report generators. Now you can use the print option to print documents with multiple pages.

The DocumentName Property

If you want to display the name of the document you're printing, you can use the `DocumentName` property of the `PrintDocument` object:

```
pd.DocumentName ="A Text Document";
```

The new result is shown in Figure 19.

We have seen that using the `DocumentPrintPreview` class is fairly straightforward. In reality, all that's happening is that this control is passed a graphics class representing each page in a printout.

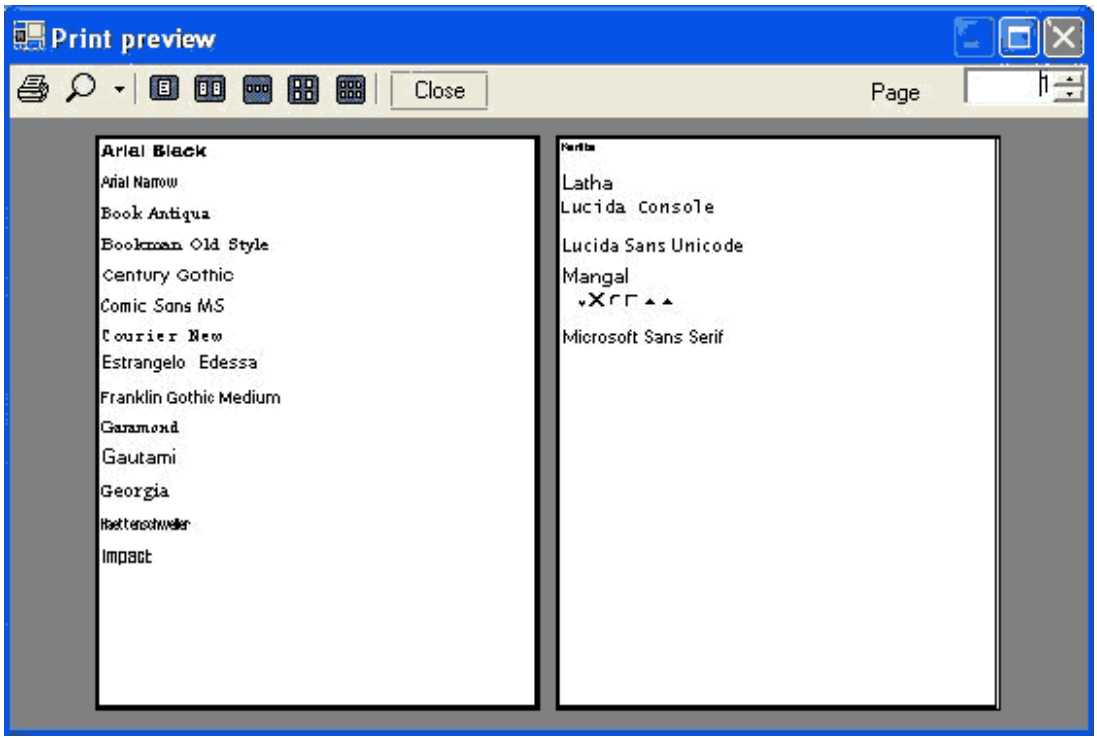

Figure 18: Print preview of multiple pages

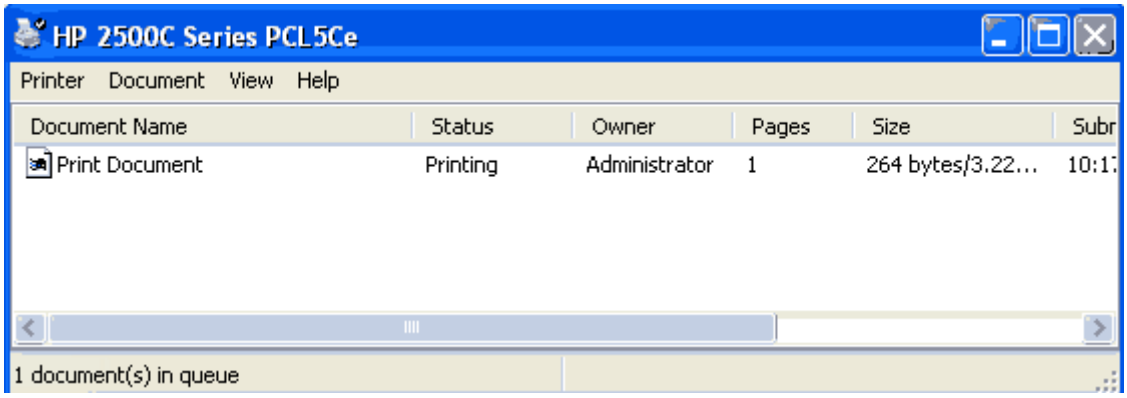

Figure 19: Setting a document name

Marginal Printing: A Caution

Although it's exciting to be able to draw graphics on a printout, keep in mind that printers have limits. Never try to print at the extreme edges page because you cannot be sure that a printer will print in exactly the same place. You could have two printers of the same model and manufacturer and yet when you print you may notice they print in different places. Some printers are more accurate than others, but usually a sheet of paper will move slightly as it moves through the printer. Laser printers tend to be able to print closer to the edges of the paper than inkjet printers because of the mechanism that is used to transport the sheet of paper thought the printer.

To see a marginal-printing sample, let's create a Windows application. We add two buttons to the form. The final form is shown in Figure 20.

Now we add code for the **Normal Printing** and **Marginal Printing** button click event handlers, as in Listing 39. Each handler creates a `PrintDocument` object, adds a `PrintPage` event handler, and calls the `Print` method. The `PrintPage` event handlers for **Normal Printing** and **Marginal Printing** are `NormalPrinting` and `MarginPrinting`, respectively.

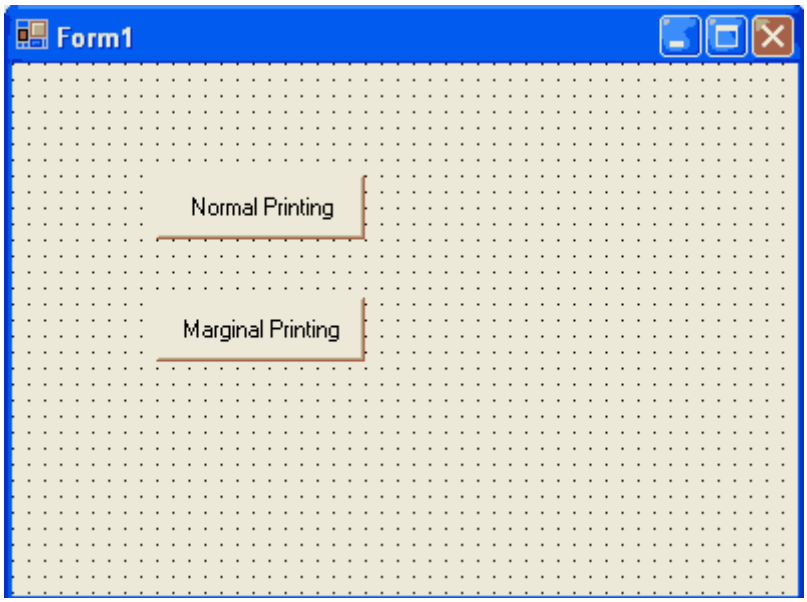

Figure 20: Marginal-printing test application

Listing 39: The Normal Printing and Marginal Printing button event handlers

```
private void NormalBtn_Click (object sender, System.EventArgs e)
{
    //Create a PrintDocument object
    PrintDocument PrintDc = new PrintDocument();
    //Add PrintPage event handler
    PrintDc.PrintPage += new PrintPageEventHandler(NormalPrinting);
    //Print
    PrintDc.Print();
}

private void MarginalBtn_Click (object sender, System.EventArgs e)
{
     //Create a PrintDocument object
     PrintDocument PrintDc = new PrintDocument();
     //Add PrintPage event handler
     PrintDc.PrintPage +=
     new PrintPageEventHandler(MarginPrinting);
     //Print
     PrintDc.Print();
}
```


Now let's look at the `NormalPrinting` handler (see Listing 40). We start with the top location of the text as unit 1. Then we calculated the nest line's position using the height of the font and draw lines with the values of the top, left, bottom, and right margins. In the end we draw a rectangle with the default bounds of the page.

Listing 40: The NormalPrinting event handler

```
public void NormalPrinting (object sender, PrintPageEventArgs ev)
{
     //Set the top position as 1
     float ypos = 1;
     //Get the default left margin
     float leftMargin = ev.MarginBounds.Left;
     //Create a font
     Font font = new Font("Arial", 16);
     //Get the font's height
     float fontheight = font.GetHeight(ev.Graphics);
     //Draw four strings
     ev.Graphics.DrawString("Top Margin = "
     + ev.MarginBounds.Top.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Bottom Margin = "
     + ev.MarginBounds.Bottom.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Left Margin = "
     + ev.MarginBounds.Left.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Right Margin = "
     + ev.MarginBounds.Right.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     //Draw a rectangle with defualt margins
     ev.Graphics.DrawRectangle(
     new Pen(Color.Black),
     ev.MarginBounds.X,
     ev.MarginBounds.Y,
     ev.MarginBounds.Width,
     ev.MarginBounds.Height);
}
```

If we run the application, we will see text describing the four margins values printed outside the rectangle.

Next comes code for the `MarginPrinting` event handler (see Listing 41). We use the default margin of the page as the top location for the first text. Everything else is the same as in Listing 40.

Listing 41: The MarginPrinting event handler

```
public void MarginPrinting (object sender, PrintPageEventArgs ev)
{
     //Set the top position as the default margin
     float ypos = ev.MarginBounds.Top;
     //Get the default left margin
     float leftMargin = ev.MarginBounds.Left;
     //Create a font
     Font font = new Font("Arial", 16);
     //Get the font's height
     float fontheight = font.GetHeight(ev.Graphics);
     //Draw four strings
     ev.Graphics.DrawString("Top Margin = "
     + ev.MarginBounds.Top.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Bottom Margin ="
     + ev.MarginBounds.Bottom.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Left Margin = "
     + ev.MarginBounds.Left.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     ev.Graphics.DrawString("Right Margin = "
     + ev.MarginBounds.Right.ToString(),
     font, Brushes.Black,
     leftMargin, ypos);
     ypos = ypos + fontheight;
     //Draw a rectangle with default margins
     ev.Graphics.DrawRectangle(
     new Pen(Color.Black),
     ev.MarginBounds.X,
     ev.MarginBounds.Y,
     ev.MarginBounds.Width,
     ev.MarginBounds.Height);
}
```

When we run this code, we will se text appearing inside the rectangle printed using the page margin values.

Getting into the Details: Custom Controlling and the Print Controller

At this point you must feel like a printer master and have the confidence you need to write a printing application. We have covered almost every aspect of printing in .NET, but guess what! There are still a few surprises hidden in `System.Drawing.Printing`. You will probably never use the classes that we're going to discuss in this section, but its' not a bad idea to know about them.

So far in this article we've created a `PrintDocument` object, created a `PrintPage` event handler, and called the `Print` method of `PrintDocument`. `PrintDocument` took care of everything internally for us. Now we will see how to control `PrintDocument` objects handles printing.

The `PrintController` class represents print controllers in the .NET Framework library. It's an abstract base class, so its functionality comes from its three derived classes: `PreviewPrintController, StandardPrintController`, and `PrintControllerWithStatusDialog. PrintController` and its derived classes are shown schematically in Figure 21.

Normally `PrintController` is used by `PrintDocument.` When `PrintDocument` starts printing by calling the `Print` method, it invokes the print controller's `OnStartPrint, OnEndPrint, OnStartPage,` and `OnEndPage` methods, which determine how a printer will print the document. Usually the `OnStartPrint` method of `PrintController` is responsible for obtaining the `Graphics` object, which is later used by the `PrintPage` event handler.

The `StandardPrintController` class is used to send pages to the printer. We set the `PrintController` property of `PrintDocument` to `PrintController`. `Standard- PrintController.PrintControllerWithStatusDialog` adds a status dialog to the printing functionality. It shows the name of the document currently being printed. To attach `PrintControllerWithStatusDialog`, we set `PrintDocument's PrintController` property to `Printcontroller.PrintControllerWithStatusDialog`.

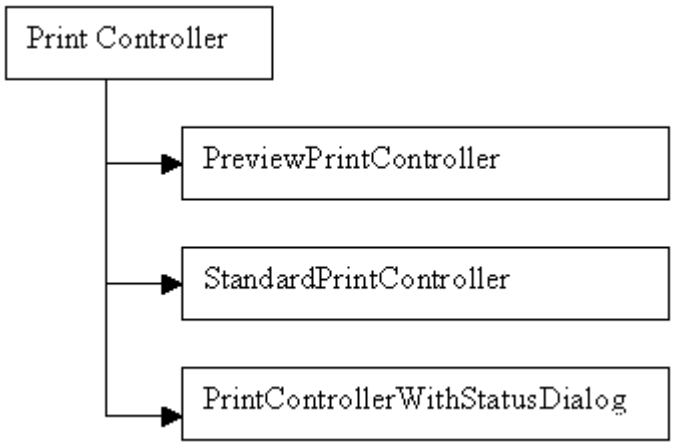

Figure 21: PrintController-derived classes

The `PreviewPrintController` class is used for generating previews of pages being printed. Beside the methods defined in the `PrintController` class, `PreviewPrintController` provides one property (`UseAntiAlias`) and one

method (`GetPreviewPageInfo`). The `UseAntiAlias` property indicates whether anti-aliasing will be used when the print preview is being displayed.

The `GetPreviewPageInfo` method captures the pages of a document as a series of images and returns them as an array called `PreviewPageInfo`. The `PreviewPageInfo` class provides print preview information for a single page. This class has two properties: `Image` and `PhysicalSize`. The `Image` property returns an Image object, which represents an image of the printed page, and `PhysicalSize` represents the size of the printed page in hundredths of an inch.

Let's write a sample application. We create a Windows application, and we add a `MainMenu` control, and item and a `StatusBar` control to the form. Our final form looks like Figure 22.

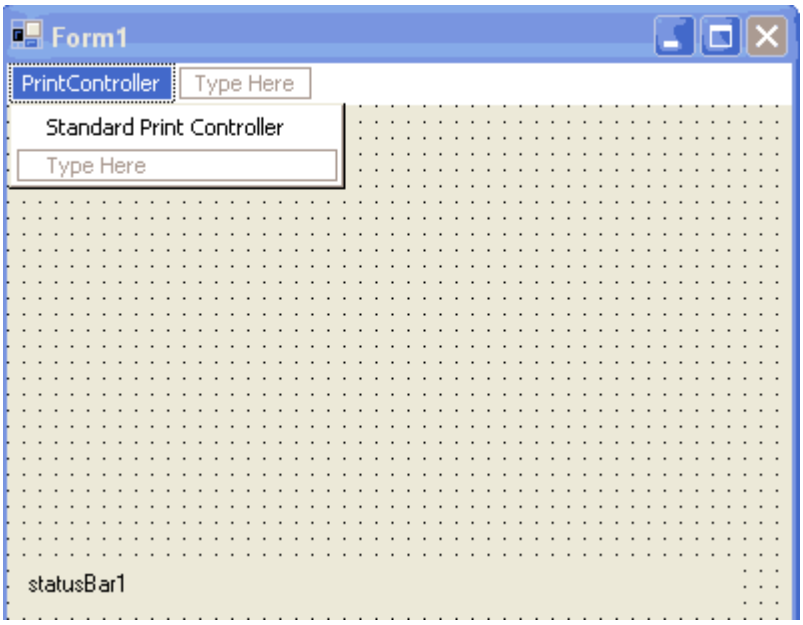

Figure 22: Print controller test form

Before adding any code to this form, we create a `MyPrintcontroller` class, which is inherited from `StandardPrintController`. You can use the `PreviewPrintController` or `PrintControllerWithStatusDialog` classes in the same way. The code for the `MyPrintController` class is given in Listing 42. We override all four methods: `OnStartPrint`, `OnStartPage`, `OnEndPrint`, and `OnEndPage`. On these methods we notify the status bar about the status of the printing process. This information could be useful for displaying page numbers or other print status information when we're printing multipage documents.

Listing 42: The MyPrintController class

```
//Print controller class
class MyPrintController: StandardPrintController
{
      private StatusBar statusBar;
      private string str = string.Empty;
      public MyPrintController (StatusBar sBar) : base()
      {
            statusBar = sBar;
      }
      public override void OnStartPrint(PrintDocument printDoc,
PrintEventArgs peArgs)
      {
            statusBar.Text = "OnStartPrint Called";
            base.OnStartPrint (printDoc, peArgs);
      }
      public override Graphics OnStartPage( PrintDocument printDoc,
PrintPageEventArgs ppea)
      {
            statusBar.Text ="OnStartPAget Called";
            return base.OnStartPage (printDoc, ppea);
      }
      public override void OnEndPage(PrintDocument printDoc,
PrintPageEventArgs ppeArgs)
      {
            statusBar.Text = "OnEndPage Called";
            base.OnEndPage (printDoc, ppeArgs);
      }
      public override void OnEndPrint(PrintDocument printDoc,
PrintEventArgs peArgs)
      {
            statusBar.Text = "OnEndPrint Called";
            statusBar.Text = str;
            base.OnEndPrint (printDoc, peArgs);
      }
}
```

To call the `MyPrintController` class, we need to set the `PrintController` property of `PrintDocument` to invoke `MyPrintController's` overridden methods. Let's write a menu click event handler and set the create a `PrintDocument` object, set its `DocumentName` and `PrintController` properties, enable the `PrintPage` event handler, and call `Print` to print the document.

Listing 43: Setting the PrintController property of PrintDocument

```
private void StandardPrintControllerMenu_Click (
object sender, System.EventArgs e)
{
    PrintDocument printDoc = new PrintDocument();
    printDoc.DocumentName ="PrintController Document";
    printDoc.PrintController =new MyPrintController (statusBar1);
    printDoc.PrintPage +=new PrintPageEventHandler (PrintPageHandler);
    printDoc.Print();
}
```


Listing 44 gives the code for the `PrintPage` event handler which, just draws some text on the printer.

Listing 44: The PrintPage event handler

```
void PrintPageHandler (object obj, PrintPageEventArgs ppeArgs)
{
    Graphics g = ppeArgs.Graphics;
    SolidBrush brush = new SolidBrush (Color.Red);
    Font verdana20Font = new Font ("Verdana", 20);
    g.DrawString ("Print Controller Test", verdana20Font, brush, 20,
20);
}
```

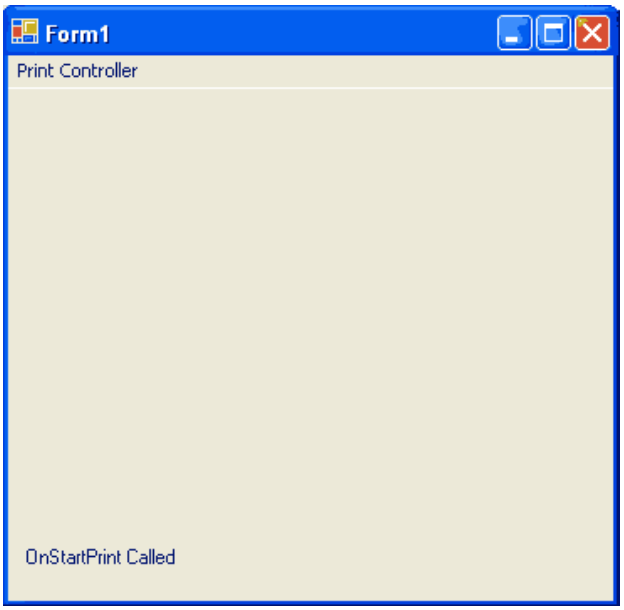

Figure 23: Print controller output

If we run the application and print, we will see that the status bar displays the status of the printing process. The first event message is shown in Figure 23.

You can extend this functionality to write your own custom print controllers.

Printing a Form

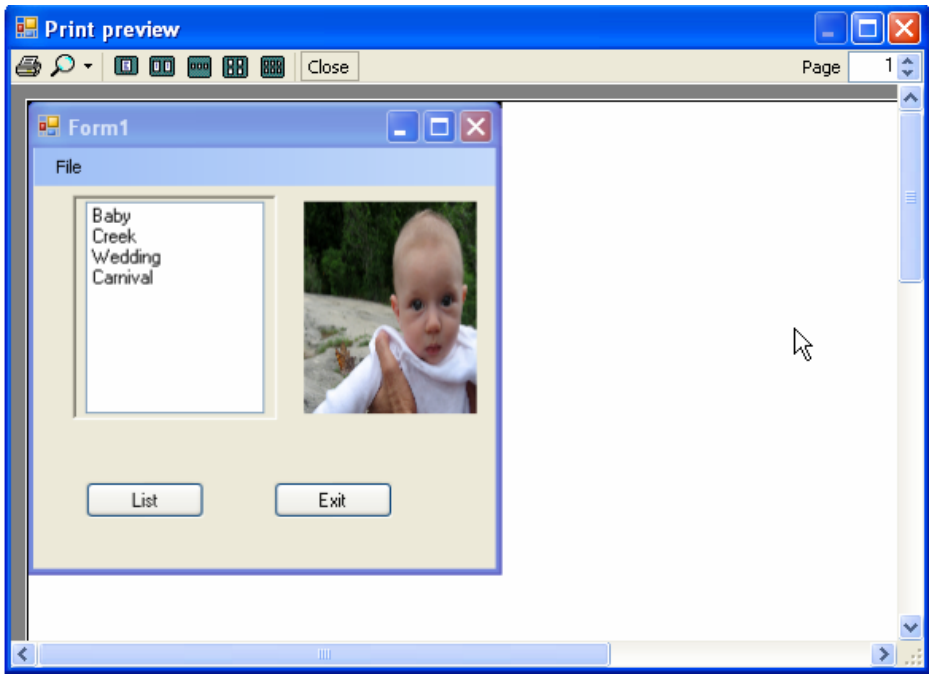

Figure 24: Printing a Form in .NET

Printing a Form is easily accomplished using the **`DrawToBitmap`** method provided with the .NET 2.0 Framework. The code in listing 45 can be placed inside the **`PrintPage`** event handler to print out an existing snapshot of your form. This code simply draws the entire form to a bitmap that is the same size as the form on your screen. The bitmap is then drawn to the printing surface using the **`DrawImage`** method.

Listing 45: Printing a Form through a Bitmap of the Form

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    // create a bitmap the size of the form
    Bitmap bmp = new Bitmap(this.Width, this.Height);

    // draw the form image to the bitmap
    DrawToBitmap(bmp, new Rectangle(0, 0, Width, Height));

    // draw the bitmap image of the
    // form onto the graphics surface
    e.Graphics.DrawImage(bmp, new Point(0, 0));
}
```

If you want to activate the **`PrintPage`** event handler of the **`PrintDocument`**, just use the code in Listing 46. The code will call the PrintDocument's **`Print`** method if the setup dialog was successfully Okayed.

Listing 46: Activating the PrintPage Event Handler to Print the Form

```
private void printToolStripMenuItem_Click(object sender,
EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        printDocument1.Print();
    }
}
```

Printing the DataGridView Control

The **`DataGridView`** is printed by brute force. We need to draw all the grid lines, text, pictures, and controls directly to the graphics surface. We obtain the contents of the grid directly through the **`DataSet`** bound to the grid. Listing 47 kicks off the drawing of the `DataGridView` through the `PrintPage` event handler.

Listing 47: Printing a DataGridView

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    Font boldFont = new Font(this.Font, FontStyle.Bold);
    Graphics g = e.Graphics;
    DrawDataGridView(boldFont, g, e);
}

private void DrawDataGridView(Font boldFont, Graphics g,
PrintPageEventArgs e)
{

    // Print the data and time
    int columnPosition = 0;
    int rowPosition = 25;
    g.DrawString(dateTimePicker1.Value.ToString(), this.Font,
 Brushes.Black, 0, 0);

    // draw headers
    DrawHeader(boldFont, g, ref columnPosition, ref rowPosition);

    rowPosition += 65;
    int height = e.PageBounds.Height;

    // draw each row
    DrawGridBody(g, ref columnPosition, ref rowPosition, e);
}
```

In order to print the contents of the `DataGridView` control, we'll need to cycle through each row of the **`DataSet`** bound to the grid. Then we go through each `DataGridViewColumn` and determine the value of the **`DataPropertyName`** or the

ValueType. If the **ValueType** is a **Boolean**, we'll literally draw the checkbox to the graphics surface using DrawString. If the **ValueType** is a **String**, we'll draw a string to the graphics surface using DrawRectangle. To get the value of the string we can use the DataPropertyName of the DataGridViewColumn to reference the DataRow's value at the column in which we are interested. For example, to get the value at a particular column of the instance of the DataRow called dr, we could obtain it's instance through dr[dataColumn.DataPropertyName]. Listing 48 draws the body of the grid to our printing graphics surface. Note that printing a picture in the DataGridView is handled by using the DrawImage method of the Graphics object.

Listing 48: Drawing the Body of the DataGridView

```
private int _lastRow = 0;
private void DrawGridBody(Graphics g, ref int columnPosition, ref int
rowPosition, PrintPageEventArgs e)
{
    // loop through each row and draw the data to the graphics
    // surface.
    DataSet ds = (DataSet)((BindingSource)gridExercise.DataSource).
DataSource;
    for (int count = _lastRow; count < ds.Tables[0].Rows.Count;
count++)
    {
        DataRow dr = ds.Tables[0].Rows[count];
        columnPosition = 0;

       // draw a line to separate the rows

       g.DrawLine(Pens.Black, new Point(0, rowPosition),new
Point(this.Width, rowPosition));

       // loop through each column in the row, and
       // draw the individual data item
       foreach (DataGridViewColumn dc in gridExercise.Columns)
       {
           // if its a picture, draw a bitmap
           if (dc.DataPropertyName == "Picture")
           {
               if (dr[dc.DataPropertyName].ToString().Length
 != 0)
               {
                   if (CustomImageCell.Images.ContainsKey(
dr[dc.DataPropertyName].ToString()))
                   {
                       g.DrawImage(CustomImageCell.Images[
dr[dc.DataPropertyName].ToString()], new Point(columnPosition,
rowPosition));
                   }
               }
           }
           else if (dc.ValueType == typeof(bool))
           {
               // draw a check box in the column
```

```
                g.DrawRectangle(Pens.Black,new Rectangle(
columnPosition, rowPosition + 20, 10, 10));
            }
            else
            {
                // just draw string in the column
                string text =dr[dc.DataPropertyName].ToString();
                if (dc.DefaultCellStyle.Font != null)
g.DrawString(text, dc.DefaultCellStyle.Font, Brushes.Black,
(float)columnPosition, (float)rowPosition + 20f);
                else
                g.DrawString(text, this.Font, Brushes.Black,
(float)columnPosition,(float)rowPosition + 20f);

            }

            // go to the next column position
            columnPosition += dc.Width + 5;
        }

        // go to the next row position
        rowPosition = rowPosition + 65;

       // if the row Position is greater than the size
       // of the page (minus the bottom margin)
       // return so we print to the next page
       if (rowPosition > e.PageBounds.Height - 65)
       {
           _lastRow = count +1;
           e.HasMorePages = true;
           return;
       }
    }
}
```

Drawing the header is accomplished in much the same way as drawing the body of the **`DataGridView`**. You simply go through each column of the **`DataGridView`** and pick out the header text. Then draw the header text to the graphics surface using **`DrawString`**. The next position of the text in the header is determined by the accumulated sum of all the previous column widths as shown in listing 49.

Listing 49: Drawing the Header of the DataGridView

```
private int DrawHeader(Font boldFont, Graphics g,
ref int columnPosition, ref int rowPosition)
{
    foreach (DataGridViewColumn dc in gridExercise.Columns)
    {
        g.DrawString(dc.HeaderText, boldFont, Brushes.Black,
(float)columnPosition, (float)rowPosition);
        columnPosition += dc.Width + 5;
    }
    return columnPosition;
}
```


Printing an Invoice

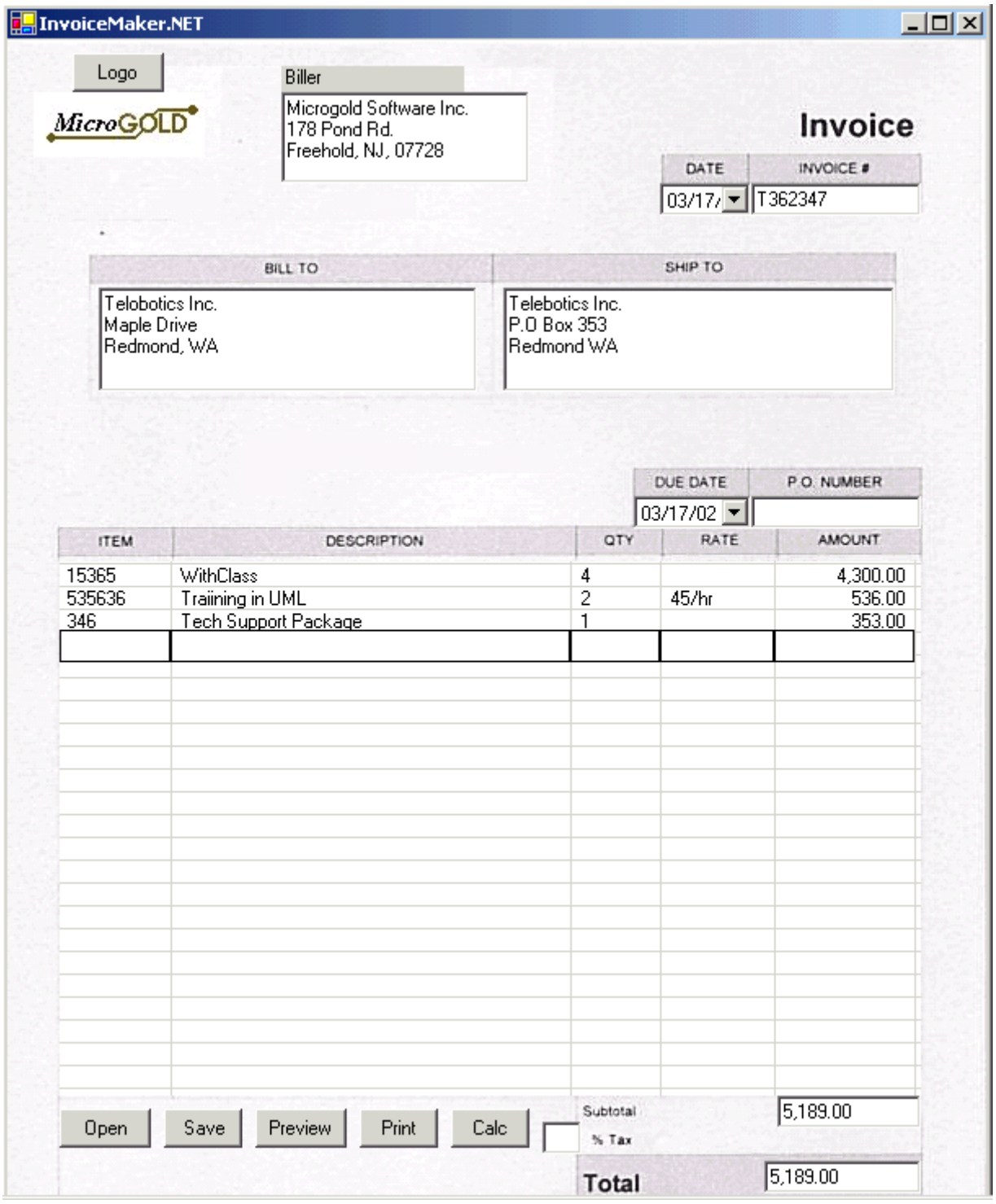

Figure 25: The Invoice in a Windows Form

This program could be improved but it will allow you to create and print invoices. You can customize the invoice by changing the bitmap supplied in the download to an invoice of your choice and then moving the controls to fit into the proper locations on the background bitmap. This invoice layout was scanned from a standard invoice form and modified to add a few features.

Below is the design for the InvoiceMaker.NET program in UML:

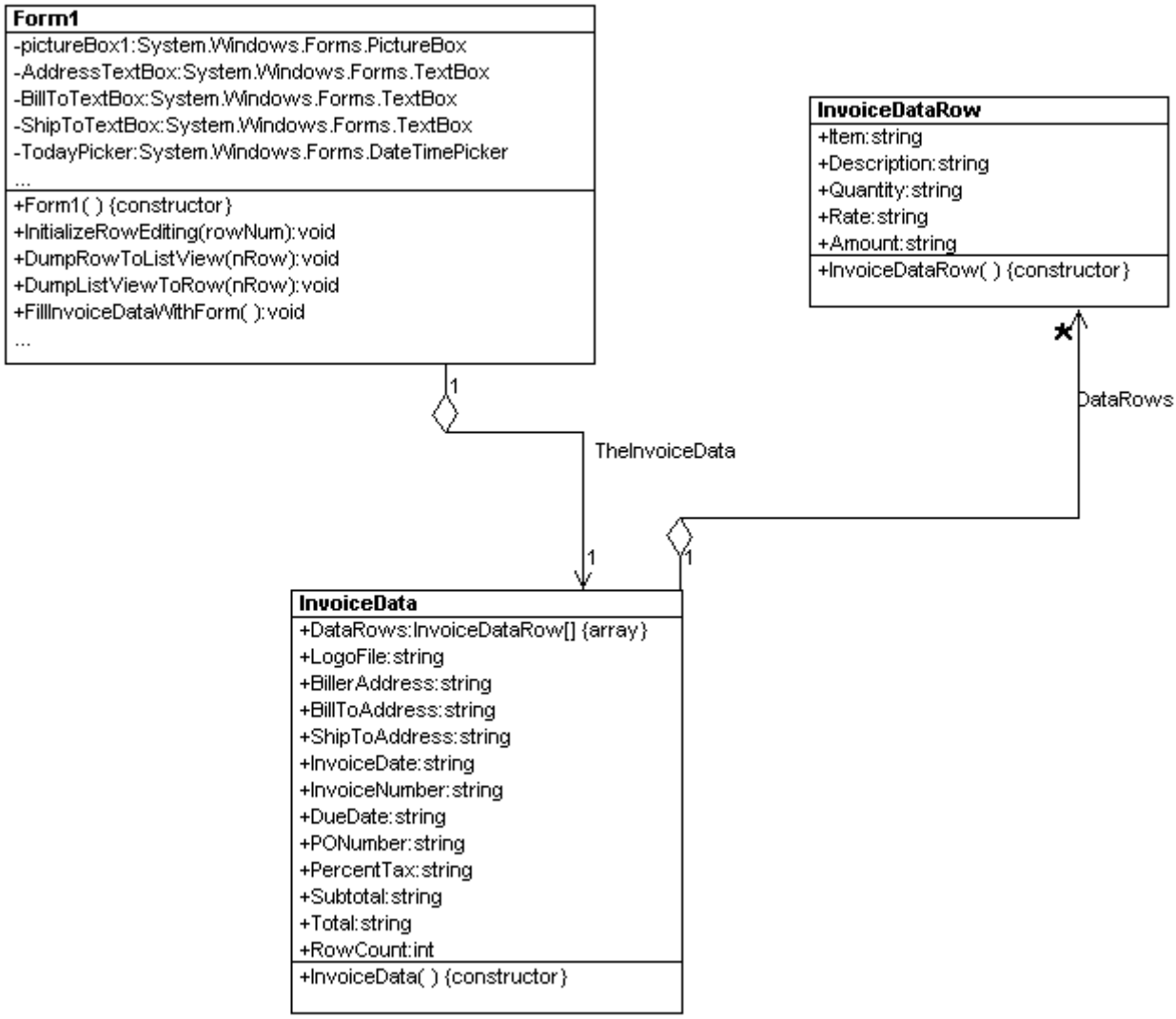

Figure 26: UML Diagram of InvoiceMaker.NET

The program has several useful pieces of code. Printing the invoice is accomplished through the following steps: a) draw the empty form bitmap to the graphics surface, b) loop through each control in the form, and c) individually draw the control to the graphics surface. It is also important to position each control correctly to the graphics surface so that they all align with the empty form bitmap. The position of each control is scaled to a printed page so they all line up properly.

Each type of control is handled separately in order to draw its specific internal shapes. There are three types of controls in our invoice: a **`TextBox`**, a **`CheckBox`**, and a **`ListView`** control.

Printing the **`TextBox`** is accomplished by drawing a rectangle border to the graphics surface with text inside. Printing a **`CheckBox`** is accomplished by drawing a square to the graphics surface with or without an "x", depending upon the state of the **`CheckBox`**. Printing the **`ListView`** is a bit more complex. The code for printing the **`ListView`** simply cycles through the **`ListView`** Items and through each **`ListView`** Item's Subitem and prints them to the graphics surface using **`DrawString`**. The horizontal position of the string is determined using the widths in the **`Columns`** collection of the **`ListView`**. Note that the position is scaled to the printing surface.

Listing 50: Drawing the Invoice to the Print Graphics surface

```
private void DrawTheInvoice(Graphics g)
{
    // Create the source rectangle from the BackgroundImage Bitmap
    // Dimensions
    RectangleF srcRect = new Rectangle(0, 0,
this.BackgroundImage.Width, BackgroundImage.Height);
    // Create the destination rectangle from the printer settings
// holding printer page dimensions
    int nWidth =
printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Width;
    int nHeight =
printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Height;
    RectangleF destRect = new Rectangle(0, 0, nWidth, nHeight/2);
    // Draw the empty form image scaled to fit on a printed page
    g.DrawImage(this.BackgroundImage, destRect, srcRect,
GraphicsUnit.Pixel);
    // Determine the scaling factors of each dimension based on the
// bitmap and the printed page dimensions
    // These factors will be used to scale the positioning of the
// contro contents on the printed form
    float scalex = destRect.Width/srcRect.Width;
    float scaley = destRect.Height/srcRect.Height;
    Pen aPen = new Pen(Brushes.Black, 1);
   // Cycle through each control. Determine if it's a checkbox or a
// textbox and draw the information inside
   // in the correct position on the form
    for (int i = 0; i < this.Controls.Count; i++)
    {
    // Check if its a TextBox type by comparing to the type of one of
// the  textboxes
        if (Controls[i].GetType() == this.Wages.GetType())
        {
            // Unbox the Textbox
            TextBox theText = (TextBox)Controls[i];
            // Draw the textbox string at the position of the textbox
// on the
            //form, scaled to the print page
            g.DrawString(theText.Text, theText.Font, Brushes.Black,
```



```
theText.Bounds.Left*scalex, theText.Bounds.Top * scaley, new
StringFormat());
        }
            if (Controls[i].GetType() ==
this.RetirementPlanCheck.GetType())
            {
                // Unbox the Checkbox
                CheckBox theCheck = (CheckBox)Controls[i];
                // Draw the checkbox rectangle on the form scaled to
// the print page
                Rectangle aRect = theCheck.Bounds;
                g.DrawRectangle(aPen, aRect.Left*scalex,
aRect.Top*scaley, aRect.Width*scalex, aRect.Height*scaley);
       // If the checkbox is checked, Draw the x inside the checkbox on
       // the form scaled to the print page
                if (theCheck.Checked)
                {
                    g.DrawString("x", theCheck.Font,
Brushes.Black,theCheck.Left*scalex + 1,theCheck.Top*scaley + 1, new
StringFormat());
                }
            }
        }
// handle List View Control printing
  if (Controls[i].GetType() == this.listView1.GetType())
  {
     for (int row = 0; row < listView1.Items.Count; row++)
     {
        int nextColumnPosition = listView1.Bounds.X;
          for (int col = 0; col < listView1.Items[row].SubItems.Count;
col++)
        {
            g.DrawString(listView1.Items[row].SubItems[col].Text,
listView1.Items[row].Font, Brushes.Black,      (nextColumnPosition +
3)*scalex,(listView1.Items[row].Bounds.Y +
listView1.Bounds.Y)* scaley, new StringFormat());
            nextColumnPosition += listView1.Columns[col].Width;
       }
}
}
```

Printing a Ruler

If you misplaced your ruler, here's an application that will create one for you on your printer! You will still need a ruler when you begin so that you can calibrate the measurement for your particular printer. Fortunately, once you know the calibration value, you are all set with a fairly accurate ruler. Below is the simple design of our ruler. The ruler itself is drawn in a Form. The only other class used is the calibration dialog used to enter and retrieve a calibration value:

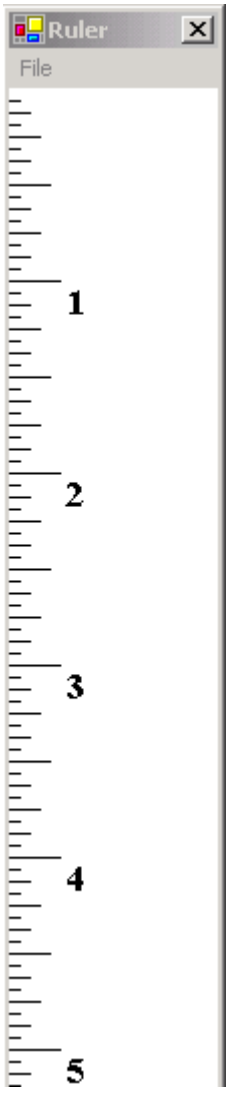

Figure 27: Part of the ruler

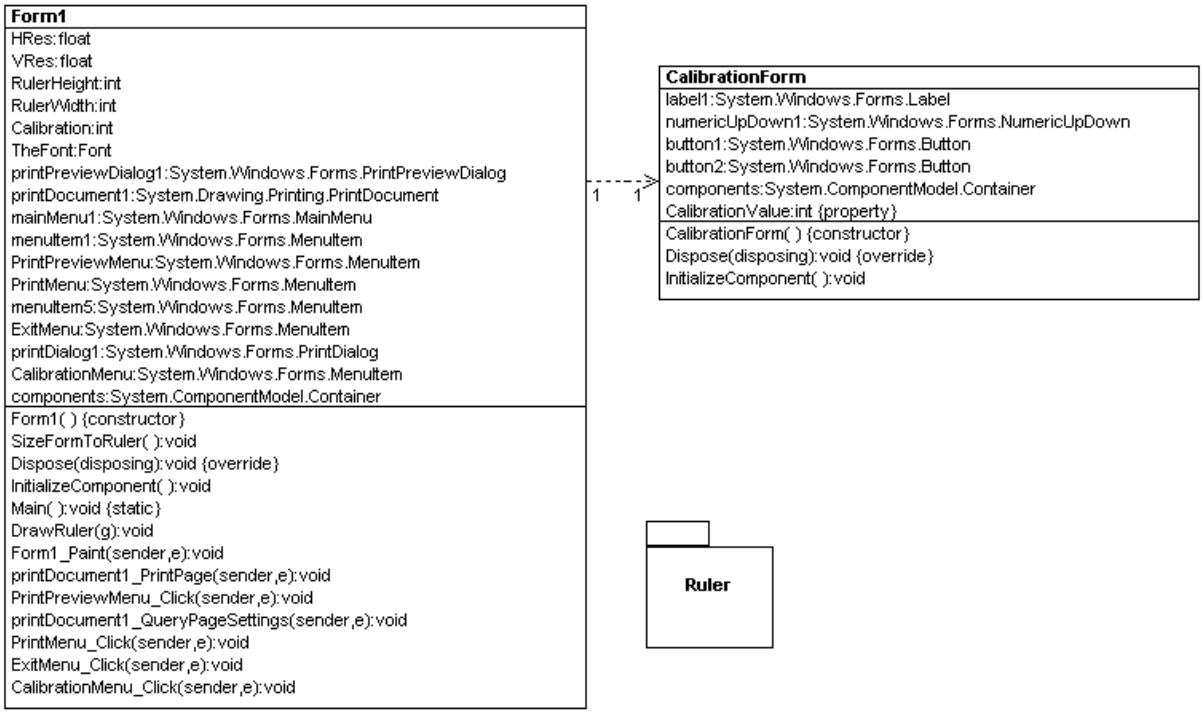

Figure 28: UML Diagram of Ruler

The ruler is created by simply determining the resolution in pixels per inch. This information can be retrieved from the graphics object itself:

Listing 51: Getting the Pixels per Inch of the Ruler

```
void SizeFormToRuler()
{
    // get resolution, most screens are 96 dpi, but you never know...
    Graphics g = this.CreateGraphics();
    this.HRes = g.DpiX;
    this.VRes = g.DpiY;
    Width = System.Convert.ToInt32(HRes);
    Height = System.Convert.ToInt32(VRes) * 11;
    Left = 250;
    Top = 5;
}
```

The actual tick marks and numbers are drawn in the **`Form_Paint`** event handler method. This method establishes a ruler height in inches and then calls the method **`DrawRuler`** to draw the ruler:

Listing 52: Drawing the Ruler in the Paint Event Handler

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
```



```
    RulerHeight = 11;
    DrawRuler(g);
}
```

The DrawRuler method is the heart of the application. It draws the tick marks and numbers and then spaces them according to the number of pixels in an inch (+ a calibration value to make the inch measurement accurate). The modulus function is used to determine when it is appropriate to draw a particular tick mark. For example when (**`i mod PixelsPerInch`**) is equal to zero, then we've found an inch marker.

Listing 53: Drawing the Ruler on the Graphics Surface

```
private void DrawRuler(Graphics g)
{
    g.DrawRectangle(Pens.Black, ClientRectangle);
    g.FillRectangle(Brushes.White, 0, 0, Width, Height);

    float PixelsPerInch = (float)(VRes +
System.Convert.ToSingle(Calibration));

    int count = 1;
    float i = 0F;
    for (i = 1; i <= (System.Convert.ToSingle(PixelsPerInch) *
RulerHeight + 10) - 0.0625F; i += 0.0625F) // mark every 1/8th inch
    {
        if (i % PixelsPerInch == 0)
        {
            g.DrawLine(Pens.Black, 0, i, 25, i);
            g.DrawString(count.ToString(), TheFont, Brushes.Black, 25F,
i, new StringFormat());
            count += 1;
        }
        else
        {
            if (i * 2 % PixelsPerInch == 0)
            {
                g.DrawLine(Pens.Black, 0, i, 20, i);
            }
            else
            {
                if (i * 4 % PixelsPerInch == 0)
                {
                    g.DrawLine(Pens.Black, 0, i, 15, i);
                }
                else
                {
                    if (i * 8 % PixelsPerInch == 0)
                    {
                        g.DrawLine(Pens.Black, 0, i, 10, i);
                    }
                    else
                    {
                        if (i * 16 % PixelsPerInch == 0)
                        {
                            g.DrawLine(Pens.Black, 0, i, 5, i);
```



```
                        }
                    }
                }
            }
        }
    }
}
```

The calibration value is retrieved by using the calibration dialog. This dialog brings up a NumericUpDown control for choosing the number of pixels to offset the inch. The offset can be a negative or positive number of pixels depending upon how much your printer is off. I found that my printer was 4 pixels too short, so my calibration was +4. Below is the calibration dialog:

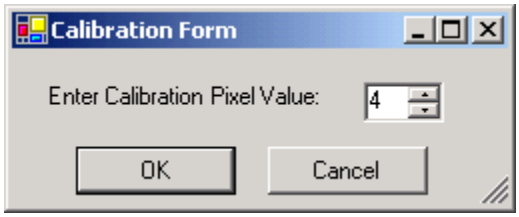

Figure 29: Ruler Calibration Dialog

The calibration dialog uses **`ShowDialog`** to display the form and retrieves the calibration value in the **`CalibrationValue`** property of the dialog:

Listing 54: Entering a Calibration Value for the Ruler

```
private void CalibrationMenu_Click(object sender, System.EventArgs e)
{
    CalibrationForm CalibrForm = new CalibrationForm();
    CalibrForm.CalibrationValue = Calibration;
    if (CalibrForm.ShowDialog() ==
System.Windows.Forms.DialogResult.OK)
    {
        Calibration = CalibrForm.CalibrationValue;
        WriteCalibration();
    }
    Invalidate();
}
```

Printing the ruler is accomplished using three different print controls: The **`PrintDocument`**, The **`PrintDialog`**, and the **`PrintPreviewDialog`** control. Once you've dropped these controls on your form and assigned the **`PrintDocument`** instance to the corresponding **`Document`** properties in the **`PrintDialog`** and the **`PrintPreviewDialog`**, it's fairly easy to set up printing. Below is the event handler for the Print Preview menu item. As you can see, there is not really much to it.

Listing 55: Executing Print Preview Dialog

```
private void PrintPreviewMenu_Click(object sender, System.EventArgs e)
{
    this.printPreviewDialog1.ShowDialog();
}
```

Printing is not much more complicated. Simply open the **`PrintDialog`** and once the user has assigned properties, call the Print method on the **`PrintDocument`**:

Listing 56: Executing Printing for the Ruler

```
private void PrintMenu_Click(object sender, System.EventArgs e)
{
    if (printDialog1.ShowDialog() ==
System.Windows.Forms.DialogResult.OK)
    {
        printDocument1.Print();
    }
}
```

The actual printing occurs in the PrintDocument's **`PrintPage`** Event Handler. This is where you put all the GDI routines for drawing the ruler. You can actually use the same drawing routines in your print handler as you do for your Paint Handler. You're just passing in a printer graphics object instead of a screen graphics object:

Listing 57: The Print Event Handler for Printing the Ruler

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    Graphics g = e.Graphics;
    PageSettings ps = e.PageSettings;
    RulerHeight = System.Convert.ToInt32(ps.Bounds.Height / 100.0);
    DrawRuler(g);
}
```

We've also added persistence to this application to remember the calibration value. The two routines below read and write the calibration value using the **`StreamReader`** and **`StreamWriter`** classes:

Listing 58: Reading the Calibration Value of the Ruler from a File

```
void ReadCalibration()
{
    // Determine the path of the file from the application itself
    string calfile = Application.StartupPath + "\\cal.txt";
    // If the calibration file exists, read in the initial calibration
value
    if (File.Exists(calfile))
    {
```



```
        StreamReader tr = new StreamReader(calfile);
        string num = tr.ReadLine();
        Calibration = Convert.ToInt32(num);
        tr.Close();
    }
}
```

Listing 59: Writing out a Calibration Value for the Ruler to a File

```
void WriteCalibration()
{
    string calfile = Application.StartupPath + "\\cal.txt";
    StreamWriter stWriter = new StreamWriter(calfile);
    stWriter.Flush();
    stWriter.WriteLine(Calibration.ToString());
    stWriter.Close();
}
```

Improvements

This control could be improved by adding a metric ruler option. This can be accomplished simply by figuring out the pixels per centimeter and then using modulus 10 to get the millimeters. Also a large ruler can be created by printing to multiple pages. I'm sure you can think of some other ideas with a little measured thought.

Bar Code Printing

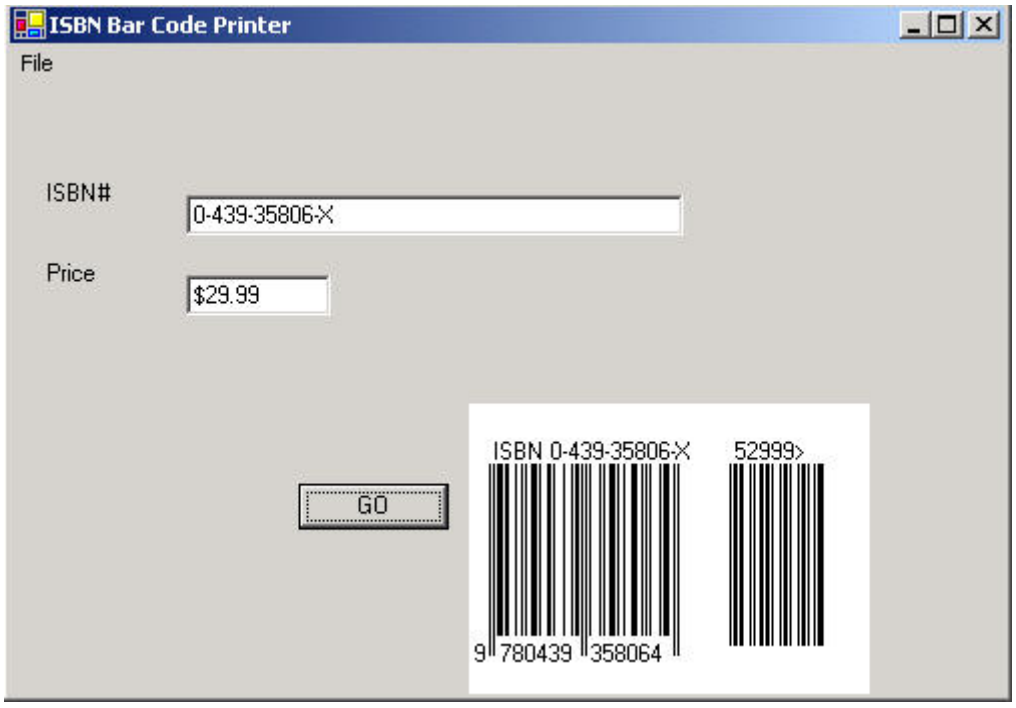

Figure 30: Bar Code Printer Form for ISBN and Book Pricing for Harry Potter and the Order of the Phoenix

Bar Codes are the way we label food, books, and other manufactured items to identify them at the register. A bar code can contain information about the product and the price of the product. EAN-13 is one of the standard bar code formats and it is used by publishers to create ISBN numbers for books. In this article we will show you how to generate and print an EAN-13 bar code.

Forming the EAN-13 Number

Before an ISBN number is encoded, you need to form a number to encode. A country prefix is added to the ISBN. For books published in the USA, this number is **978**. For other countries this number will differ. The first nine numbers of the ISBN are used to complete the code, and then a checksum is tagged on as the final number. For the HarryPotter Book the EAN-13 number is encoded as:

Country Code + ISBN + Checksum

or 978 + 043935806 + 4

Understanding the EAN-13 Bar Code

Bar Codes are nothing more than binary numbers encoding a serial number for a book. The black lines represent ones and the white lines represent zeroes. All EAN-13 bar codes are made up of the following components:

Left Guard Bars - This is a **101**, which means ”black line, white line, black line”. The height of the lines is extended to exaggerate the starting point.

Characters 2 through 7 (left side sequence) - These characters are encoded by a parity system. The parity of each number at a particular position in the entire bar code sequence is determined by character #1. Also, character #1 is implicitly encoded through the type of parity used for every character in the bar code, so you don't need bars for character #1. (Don't even begin to ask me why it’s done this way; I guess it saves you a few bars for character #1.) There is a look-up table used by the program to determine how Characters 2-7 are encoded based on both their position in the sequence and character #1. Also note that characters on the left side always start with 0 (as a white line). They are encoded differently from the right side.

Center Guard - encoded as **01010**.

Characters 8 through 12 (right side sequence) - The right side sequence is a little more straightforward and doesn't depend on positions or the first character. There is a one to one binary number correspondence to a particular number. For example 2 is **1101100** on the right side**.** All right side characters start with a binary 1 (as a black line).

Character 13 (The CheckSum) - This character is computed by summing all of the alternating positions of the 12 digit number (numbers at position 1,3,5,7,9,11) and adding a 3X weighted sum of the rest of the numbers (at positions 2,4,6,8,12). The least significant digit of this sum is taken and subtracted from 10. If the answer is 10 then we use 0 as the checksum. (See Bar Code Island for an example of this calculation.)
Right Guard - Same as the Left Guard, **101**.

Supplemental 5-digit Price Code

This is handled much differently from the EAN-13 number. First of all, the number prefixed on the front of the bar code represents the currency. The dollar currency (USD) is represented by a **5**. There is no decimal character in the numeric representation but it is assumed before the last 2 numbers. For example, a book priced at $29.99 would be represented as 52999.

Supplemental Bar Code

1. First, a checksum is computed and used to determine parity for bar coding. The checksum is determined by adding numbers at positions (1,3,5) and multiplying by 3. Then numbers are taken at positions (2 and 4) and multiplied by 9. Then these numbers are added together and the least significant digit is taken.
2. **Left Guard - 1011** or “black line, white line, thick black line” (2 black lines together)
3. **Characters 1-5,** encoded using the appropriate parity pattern based on the checksum and separated by **01**

Bar Code Printing Application

The Bar Code Printing application was written in C# and allows you to print or print preview the bar code as well as save the bar code to a bitmap. You can't print a series of bar code labels on a sheet of paper, but .NET will allow you to convert the application over.

Below is the simple UML design of the Bar CodePrinter:

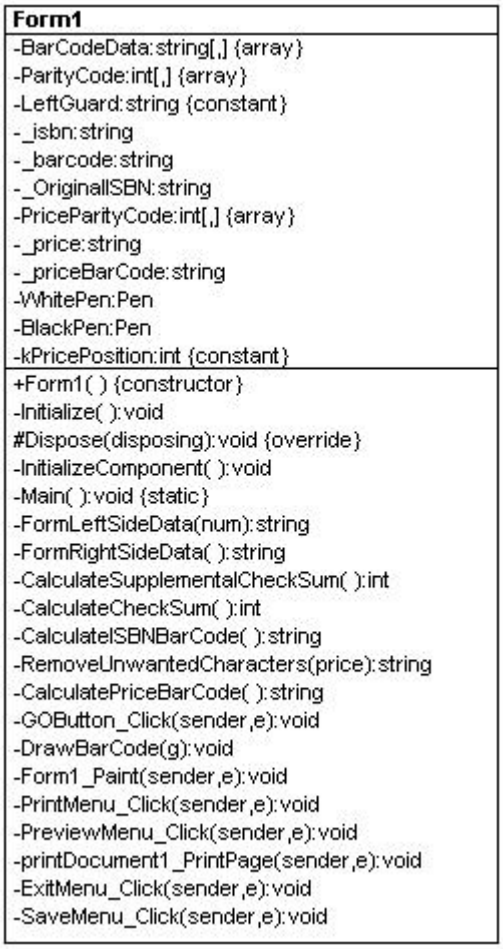

Figure 31: UML Design of Bar Code Printer reverse engineered from C# using WithClass

This could probably be broken out into 3 more classes: **`Bar CodeCalculator`**, **`EAN13Calculator`**, and **`SupplementalCalculator`**. However, in this article all the methods for calculation are contained in the form: **`CalculateISBNBar Code`** and **`CalculatePriceBar Code`**. **`CalculateISBNBar Code`** is shown below. Note that it follows the steps we described at the beginning of this article:

Listing 60: Calculating the EAN-13 number in C#

```
private string CalculateISBNBar Code()
{

try
 {
    _isbn = this.ISBNTextBox.Text.Trim();
```

```
    // save the original
     _OriginalISBN = _isbn;
      // add U.S. country code
  _isbn = "978" + RemoveUnwantedCharacters(_isbn);
  // left guard is "101"
_bar code = "101";

    // first number is encoded by parity code
 string firstNumber = _isbn[0].ToString();
 int nFirstNumber = Convert.ToInt32(firstNumber);
  // form left side binary string based on the first number
     (9 for U.S.)
   string leftSideData = FormLeftSideData(nFirstNumber);
  // add it to bar code
 _bar code += leftSideData;
  // add center guard
 _bar code += "01010";
  // get right side data
       string rightSideData = FormRightSideData();
 // add right side data to bar code
 _bar code += rightSideData;
    // calculate Checksum and add it using right side data
  int checkSum = CalculateCheckSum();
   // last character of 13 digits is checksum
_isbn = _isbn.Substring(0, 12) + Convert.ToString(checkSum);
   // tag on the checksum to the end
  _bar code += Bar CodeData[checkSum, 2];
  // add right guard
 _bar code += "101";
 }
catch (Exception ex)
 {
MessageBox.Show(ex.ToString());
 }
return _bar code;
}
```

Printing the Bar Code

Printing the bar code is accomplished in a routine called **`DrawBar Code`** which is called from the **`PrintPage`** event handler. This drawing routine takes the final EAN-13 bar code binary string and the Supplemental Price binary string and prints it out in its bar code form with 1's represented by black lines and 0's represented by white lines. The draw routine also adds the appropriate strings above and below the bar code graphics.

Listing 61: Drawing the Bar Code to the Printer

```
private void DrawBar Code(Graphics g)
{
   // establish painting rectangle
   Rectangle r = GOButton.Bounds;
   Rectangle ISBNBounds = new Rectangle(r.Right + 10, r.Top - 40, 200,
150);
   g.FillRectangle(Brushes.White, ISBNBounds);
```



```
// set up positions of the extended bars in the EAN-13 code
   int[] LongPositions = new int[]{0,1,2,3, 45, 46, 47, 48, 92, 93, 94,
95};
//  set up the termination positions of left, center, and right guards
  int[] TerminationPositions = new int[]{3, 48, 93};
// used to adjust vertical length of bars
int adjustment = 25;
// Stick EAN-13 Bar Code in Painting Rectangle
for (int i = 0; i < _bar code.Length; i++)
   {
    // see if the position needs to be lengthened
   if (Array.BinarySearch(LongPositions, 0, LongPositions.Length, i) >=
0)
 adjustment = 25;
     else
 adjustment = 35;
// draw a vertical black line for 1's and
// a vertical white line for 0's
  if (_bar code[i] == '1')
     g.DrawLine(BlackPen, ISBNBounds.Left + 10 + i,
ISBNBounds.Top + 30,
ISBNBounds.Left + 10 + i,
       ISBNBounds.Bottom - adjustment);
  else
    g.DrawLine(WhitePen, ISBNBounds.Left + 10 + i,
       ISBNBounds.Top + 30,
       ISBNBounds.Left + 10 + i,
      ISBNBounds.Bottom - adjustment);
}

// Stick supplemental Bar Code in Painting Rectangle
for (int i = 0; i < _priceBar Code.Length; i++)
 {
  // draw a vertical black line for 1's and
  // a vertical white line for 0's
   if (_priceBar Code[i] == '1')
       g.DrawLine(BlackPen, ISBNBounds.Left + kPricePosition + i,
  ISBNBounds.Top + 30,
  ISBNBounds.Left + kPricePosition + i,
  ISBNBounds.Bottom - 30);
   else
g.DrawLine(WhitePen, ISBNBounds.Left + kPricePosition + i,
    ISBNBounds.Top + 30,
    ISBNBounds.Left + kPricePosition + i,
    ISBNBounds.Bottom - 30);
  }
if (this.ISBNTextBox.Text.Length > 0)
   {
     // write out numbers on top and bottom of the bar code graphic

      // first write out ISBN
     g.DrawString("ISBN " + _OriginalISBN, this.Font, Brushes.Black,
ISBNBounds.Left + 10, ISBNBounds.Top + 17);
// also write out the code in ascii
     g.DrawString(_isbn[0].ToString(), this.Font, Brushes.Black,
ISBNBounds.Left, ISBNBounds.Bottom - 32);
```



```
    g.DrawString(_isbn.Substring(1,6) , this.Font, Brushes.Black,
ISBNBounds.Left + TerminationPositions[0] + 10,
  ISBNBounds.Bottom - 32);
    g.DrawString(_isbn.Substring(7,6) , this.Font, Brushes.Black,
ISBNBounds.Left + TerminationPositions[1] + 10,
ISBNBounds.Bottom - 32);
// also write out price in ascii
     g.DrawString(_price + ">" , this.Font, Brushes.Black,
ISBNBounds.Left + kPricePosition, ISBNBounds.Top + 17);
 }
}
```

Saving the Bar Code to a Bitmap

It is useful to know how to save an image to a bitmap. Basically, we just rework the **`DrawBar Code`** method to draw to a bitmap rather than to the printer. Specifically, a **`Graphics`** object is created from a bitmap and then sent to the **`DrawBar Code`** method. Once the bitmap is drawn to, it is saved using the **`Save`** method of the **`Bitmap`** class.

Listing 62: Saving the Bar Code to a Bitmap

```
//  Create a Blank Bitmap
Image image = new Bitmap(ClientRectangle.Width,
 ClientRectangle.Height);
//  Get a drawing surface from the Bitmap
Graphics g = Graphics.FromImage(image);
g.SmoothingMode = SmoothingMode.HighQuality;

//  Draw the Image to the Bitmap
DrawBar Code(g);

//  Save the final drawn bitmap to a file.
image.Save(saveFileDialog1.FileName, ImageFormat.Bmp);
```

Well it is that time of the year again. Time to get in shape for the summer months, and what better way to do it then through a rigid exercise program. I was never very good at tracking my progress in the gym, but thanks to .NET (and my wife), I have a way to do just that. This program uses the DataGridView bound to a Microsoft Access Database to create a printed sheet with your latest work out plans. The workout chart includes the exercise, number of sets, number of reps, amount of weight, and most importantly, whether or not you completed the exercise. The best part of this program is that you can print out your exercise program and bring it with you to the weight room.

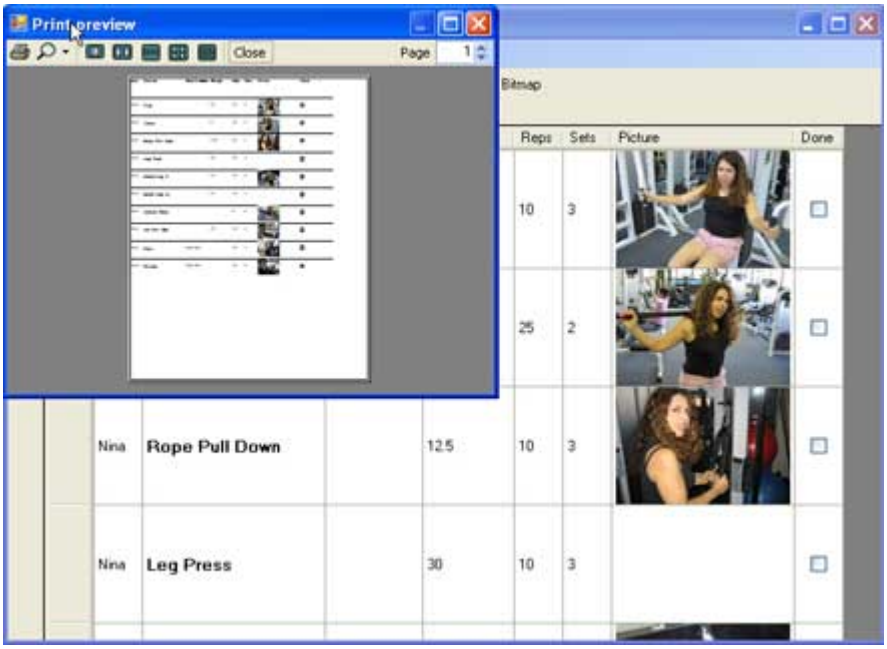

Figure 32: Exercise Program

New Form Capture Feature

. NET comes with a new form capture feature: the ability to copy the form as you see it on the screen into a bitmap. Although not useful for controls that you need to scroll, New Form Capture Feature works if you can fit all the contents you need to see on the screen into a bitmap. In this application, we stretched the DataGridView to allow for the full contents of the exercises, and we enabled AutoScrolling on the Form. This way the DataGridView will be much larger than the form, and the screen capture will print the entire DataGridView contents to the printer. The exercise application gives you two options: turning on the form capture feature to print out the form bitmap, or printing using a customized DataGridView printing. The customized printing is the default mode of the exercise program and it's more powerful because you can print the data in the grid view beyond any scrolling limitations of the DataGridView. Below is the code for capturing the form in a bitmap and drawing it to a graphics surface:

Listing 63: Printing the Form Capture to a Graphics Surface

```
private void DrawFormSurface(System.Drawing.Printing.PrintPageEventArgs
e)
{
    // create a bitmap the size of the form
    Bitmap bmp = new Bitmap(gridExercise.Width, gridExercise.Height);
    // draw the form image to the bitmap
    gridExercise.DrawToBitmap(bmp, new Rectangle(0, 0,
gridExercise.Width, gridExercise.Height));
    // draw the bitmap image of the form onto the graphics surface
    e.Graphics.DrawImage(bmp, new Point(0, 0));
}
```


Using a Filename to Map an Image in the DataGridView

We want to place an image of the exercise into the DataGridView without inserting the image in the database. We prefer to place all the images in an images directory. Then we will read the images file names from the database, retrieve the images from the images directory, and put the images into the DataGridView. In order to accomplish all this, we'll need to create a custom grid view column and grid view cell that inherits from **DataGridViewImageColumn** and **DataGridViewImageCell**. We then override the **GetFormattedValue** method in the custom DataGridViewImageCell class. In the GetFormatted value method, we create a mapping between the file names and the actual images. Below is the code for accomplishing the filename to image mapping in a DataGridView:

Listing 64: Mapping Images into the DataGridView from file name Strings

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

namespace ExerciseProgram
{
 /// <summary>
 /// Create a Custom DataGridViewImageColumn
 /// </summary>
    public class StringImageColumn : DataGridViewImageColumn
    {
        public StringImageColumn()
        {
            this.CellTemplate = new CustomImageCell();
            this.ValueType = typeof(string);
 // value of this column is a string, but it shows  images in the cells
after formatting
        }
    }

/// <summary>
/// Create a Custom DataGridViewImageCell
/// </summary>
    public class CustomImageCell : DataGridViewImageCell
    {
        // mapping between filename and image
        static System.Collections.Generic.Dictionary<string, Image>
dotImages = new Dictionary<string, Image>();
// load up custom dot images
        static public System.Collections.Generic.Dictionary<string,
Image> Images
        {
            get
```



```
            {
                return dotImages;
            }
        }
        public CustomImageCell()
        {
            this.ValueType = typeof(int);
        }
        protected override object GetFormattedValue(
object value, int rowIndex, ref DataGridViewCellStyle cellStyle,
TypeConverter valueTypeConverter, TypeConverter
formattedValueTypeConverter, DataGridViewDataErrorContexts context)
        {
            if (value.GetType() != typeof(string))
            return null;
 // get the image from the string and return it
            LoadImage(value);
 // return the mapped image
            return dotImages[(string)value];
        }
        public static void LoadImage(object value)
        {
 //  load the image from the images directory if it does not exist
            if (dotImages.ContainsKey((string)value) == false)
            {
                string path = Path.GetDirectoryName
(Application.ExecutablePath) + "\\images\\" + (string)value;
                // read the image file
                Image theImage = Image.FromFile(path);
                //  assign the image mapping
                dotImages[(string)value] = theImage;
            }
        }
        public override object DefaultNewRowValue
        {
            get
            {
                return 0;
            }
        }
    }
}
```

Updating the Grid

The DataGridView is bound to the Microsoft Access Database via a DataGridAdapter. We can edit the sets, reps, and weight values inside the grid while increasing body strength and the program will persist these values to the database. Sometimes it is useful to override the behavior of the DataGridView for updates and inserts in order to customize how the grid is updated. Below is the ADO.NET code that updates the grid in Microsoft Access without using the OleDb Adapter:

Listing 65: Updating the Microsoft Access Database after editing the GridView

```
/// <summary>
/// Event Handler called when Cell is finished editing
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void gridExercise_CellEndEdit(object sender,
DataGridViewCellEventArgs e)
{
    UpdateCurrentRow(e);
}
private void UpdateCurrentRow(DataGridViewCellEventArgs e)
{
    int index = e.RowIndex;
    DataRow dr = scheduleDataSet.Tables[0].Rows[e.RowIndex];
    string val =
dr[gridExercise.Columns[e.ColumnIndex].DataPropertyName].ToString();
    OleDbCommand updateCommand = new OleDbCommand();
    // construct update command and update from the data set
    updateCommand.CommandText = "UPDATE `Schedule` SET `Completed` =
{0}, `Reps` = {1}, `Weight` = {2}, `Sets` = {3} WHERE `ID` = {4}";
    updateCommand.CommandText = tring.Format(updateCommand.CommandText,
dr["Completed"], dr["Reps"], dr["Weight"], dr["Sets"], dr["ID"]);
    updateCommand.CommandType = System.Data.CommandType.Text;
    updateCommand.Connection = scheduleTableAdapter.Connection;
    updateCommand.Connection.ConnectionString = string.Format(
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source={0}\Schedule.mdb",
Path.GetDirectoryName(Application.ExecutablePath));
    updateCommand.Connection.Open();
    // execute the update on the database
    updateCommand.ExecuteNonQuery();
    updateCommand.Connection.Close();
}
```

Printing the DataGridView by Brute Force

Once we are satisfied with our exercise parameters, it is time to print out the exercise schedule and head to the gym. Printing is accomplished using the PrintDocument in conjuction with GDI+. Every row and column is painstakingly drawn to the print graphics surface. Listing 66 shows the code that draws the exercise rows below the header. The method loops through each row in the DataSet and draws the text corresponding to the column in the data row. If the row contains a picture, rather than text, the image that is mapped to the text in the cell is printed instead. We also need to track the row and column position after placing each object onto the graphics surface so we know where the next object goes. We do this by incrementing the local variables: columnPosition and rowPosition.

Listing 66: Printing out the Exercise Rows

```
private void DrawGridBody(Graphics g, ref int columnPosition, ref int
rowPosition)
{
    // loop through each row and draw the data to the graphics
    // surface.
    foreach (DataRow dr in scheduleDataSet.Tables[0].Rows)
    {
        columnPosition = 0;

        // draw a line to separate the rows
        g.DrawLine(Pens.Black, new Point(0, rowPosition), new
Point(this.Width, rowPosition));

         // loop through each column in the row, and
         // draw the individual data item
        foreach (DataGridViewColumn dc in gridExercise.Columns)
        {
            // if its a picture, draw a bitmap
            if (dc.DataPropertyName == "Picture")
            {
                if (dr[dc.DataPropertyName].ToString().Length != 0)
                {
                    if (CustomImageCell.Images.ContainsKey
(dr[dc.DataPropertyName].ToString()))
                    {
g.DrawImage(CustomImageCell.Images[dr[dc.DataPropertyName].ToString()],
new Point(columnPosition, rowPosition));
                    }
                }
             }
             else if (dc.ValueType == typeof(bool))
             {
                 // draw a check box in the column
                 g.DrawRectangle(Pens.Black, new
Rectangle(columnPosition, rowPosition + 20, 10, 10));
             }
             else
             {
                 if (dc.DefaultCellStyle.Font !=
null)g.DrawString(text, dc.DefaultCellStyle.Font,
Brushes.Black,(float)columnPosition, (float)rowPosition + 20f);

                 else
                 g.DrawString(text, this.Font, Brushes.Black,
(float)columnPosition, (float)rowPosition + 20f);
             }

             // go to the next column position
             columnPosition += dc.Width + 5;
         }
         // go to the next row position
         rowPosition = rowPosition + 65;
     }
}
```


The DataGridView is even more full featured than the DataGrid and provides a good interface for manipulating DataSets. This article explained two ways to print data inside the DataGridView. Hopefully this exercise will give you some insight into printing one of the more powerful controls inside the Windows Form in C# and .NET.

Conclusion

It is probably harder to deconstruct a bar code than to generate its code in .NET. The brunt of the work consists of creating lookup tables to handle parity checking for different positions. Some more lookup tables might be added to the program to improve its capability. For example, a look up table for country codes and currency could internationalize the program a bit. Or, you could break the form out into an inherited hierarchy of calculating bar code classes (as previously mentioned) in order to generalize the bar code methods and improve the program design. Then you could build upon existing bar code formats more easily. Anyway, this handy bar code program should get you started in creating bar codes for your products so they can NET them at the register.

Summary

Printing functionality in the .NET Framework library is defined in the `System.Drawing.Printing` namespace. In this article we discussed almost every possible aspect of printing. We began by discussing the history of printing in Microsoft windows. Then we explained printing-related functionality in the Microsoft .NET Framework.

After a basic introduction to printing in .NET, you learned the basic steps required to write a printing application. You also learned how printing differs from on-screen drawing, and how to print simple text; graphics object such as lines, rectangle, and circles, images, text files; and other documents.

The `PrinterSettings` class provides members to get and set printer settings. We discussed how to use this class and its members.

The .NET Framework library provides printing-related standard dialogs. You learned to use the `PrintDialog, PrintPreviewDialog`, and `PageSetupDialog` classes to provide a familiar Windows look and feel to your application.

Multipage printing can be tricky. You learned how to write an application with multipage printing functionality.

At the end of this article we discussed how to write custom printing and page setup dialogs using `PageSettings` and related classes. We also explained the advanced custom print controller-related classes and how to use them in an application.

Finally, we provided sample custom applications to practice the printing classes discussed throughout this article.

About C# Corner Authors Team

C# Corner Authors Team, led by **Mahesh Chand** and **Mike Gold**, founders of C# Corner is a team of Microsoft .NET MVPs, Authors, Experts, and Consultants, who is dedicated to bring .NET technology in an easy-to-understand approach with real-world ready-to-use applications. Contact C# Corner Authors Team at authors@c-sharpcorner.com.

