ComponentOne

PDF for Windows Phone

By GrapeCity, Inc.

Corporate Headquarters
ComponentOne, a division of GrapeCity
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

Internet: **info@ComponentOne.com**

Web site: **http://www.componentone.com**

Sales

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne PDF for Windows Phone Overview

Easily create and email Adobe PDF documents with **ComponentOne PDF™ for Windows Phone**. Create dynamic reports with support for security, compression, outlining, hyper-linking, and attachments, and then display them using **C1PdfViewer** or take them to the cloud.

Getting Started

Get started with the following topics:

- Key Features
- Quick Start

Help with ComponentOne Studio for Windows Phone

Getting Started

For information on installing **ComponentOne Studio for Windows Phone**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Studio for Windows Phone.

What's New

For a list of the latest features added to **ComponentOne Studio for Windows Phone**, visit What's New in Studio for Windows Phone.

PDF for Windows Phone

Create Adobe PDF documents from your apps using **ComponentOne PDF™ for Windows Phone**. The commands provided for adding content to documents are similar to the ones available in the .NET Graphics class. PDF for Windows Phone gives you security, compression, outlining, hyper-linking, and attachments.

One of the main features in the **C1PdfDocument** class is its ease of use. The commands provided for adding content to documents are similar to the ones available in the WinForms Graphics class. If you know how to display text and graphics in WinForms, then you already know how to use **C1PdfDocument**.

C1PdfDocument uses a Point-based coordinate system with the origin at the top-left corner of the page. This is similar to the default coordinate system used by .NET, but is different from the default PDF coordinate system (where the origin is on the bottom-left corner of the page).

C1PdfDocument supports many advanced features included in the PDF specification, including security, compression, outlining, hyper-linking, and file attachments.

The main limitation of the Windows Phone version is that it only supports the Acrobat Reader built-in fonts: Times, Helvetica, Courier, and Symbol. This is because embedding other fonts types would require access to font outline information which is not available to Windows Phone. In future versions, we may add support for downloading this information from the server. Acrobat Forms and text annotations are also not supported at this time.

Key Features

ComponentOne PDF for Windows Phone allows you to create customized, rich applications. Make the most of **PDF for Windows Phone** by taking advantage of the following key features:

- **Easily Add Content**

  The C1PdfDocument class is easy to use. The commands provided for adding content to documents are similar to the ones available in the WinForms Graphics class. If you know how to display text and graphics in WinForms, you already know how to use C1PdfDocument in Windows Phone. Add text, images, lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and more.

- **Easily Add Images**

  Adding images to PDF documents is easy; all the work is done by the DrawImage method. DrawImage draws a given image at a specified location and has parameters that provide control over the image alignment and scaling. You can render any regular .NET Image object, including metafiles.

- **Fast Rendering and Compression of Images**

  C1PdfDocument allows multiple levels of compression giving options for high quality and small file size. Metafiles are parsed and converted into vector graphics commands and thus retain the best possible resolution. If you want to add charts or technical drawings to your PDF document, metafiles are better than bitmap images.

- **Security and Permission**

  If your PDF documents contain sensitive information, you can encrypt them so that only authorized users can access it. There is a separate password for the owner of the document and for all other users. The user's access can be selectively restricted to allow only certain operations, such as viewing, copying, printing, or editing the document.

- **Bookmarks**

Most long PDF documents contain an outline structure that is displayed on a pane on the left of the reader. The outline makes it easy to browse through a document's structure and find specific topics. With PDF for Windows Phone, you can build this outline structure by adding outline entries (bookmarks).

- **Hyperlinks and Local Links**

  C1PdfDocument provides methods for adding hyperlinks and hyperlink targets to your PDF documents. You can also add local links, that when clicked take the user to another location within the same PDF document. This type of link is useful when you want to implement some type of cross-referencing within the document, such as a table of contents or an index.

- **Document Information**

  C1PdfDocument allows you to add meta data to the PDF documents you create. Specify author, creation date, keywords, and so on. You can also provide default viewer preferences to be applied when the document is opened in the Adobe Reader. Specify the initial page layout, window position, as well as reader toolbar and menu visibility.

PDF for Windows Phone Quick Start

The following quick start guide is intended to get you up and running with **PDF for Windows Phone**.

To create PDF documents using **C1PdfDocument**, three steps are required:

1. Create a **C1PdfDocument** object.
2. Add content to the document. This usually involves calling the **DrawString** method.
3. Save the document to a stream using the **Save** method.

In this quick start you will create a new project with a **C1PdfDocument** object, add content to the document, and save the document. Note that this tutorial also uses the **C1PdfViewer** control included in **PdfViewer for Windows Phone** to view the resulting PDF file.

Step 1 of 4: Creating an Application with the C1PdfDocument Object

In this step, you'll create a Windows Phone application and add a **C1PdfDocument** object.

1. In Visual Studio 2010, select **File | New | Project** to open the **New Project** dialog box.
2. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **Windows Phone Application**. Enter a **Name** for your project and click **OK**. The **New Windows Phone Application** dialog box will appear.
3. Click **OK** to close the **New Windows Phone Application** dialog box and create your project.
4. Right-click the project in the Solution Explorer and select **Add Reference**.
5. In the **Add Reference** dialog box, locate and select the **C1.Phone.dll**¸ **C1.Phone.Pdf.dll**¸ and **C1.Phone.PdfViewer.dll** files and click **OK** to add the references to your project.
6. Open the **MainPage.xaml** file.
7. Edit the **TitlePanel** content to change the text in the **TextBlock** controls. It will appear similar to the following:

```
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="ComponentOne Studio for
Windows Phone" Style="{StaticResource PhoneTextNormalStyle}"/>
```

```
    <TextBlock x:Name="PageTitle" Text="PDF" FontSize="56" Margin="9,-
7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
```

8. Add the **C1PdfViewer** XAML namespace to the `<phone:PhoneApplicationPage>` tag by adding `xmlns:my="clr-namespace:C1.Phone;assembly=C1.Phone.PdfViewer"` so it appears similar to the following:

```
<phone:PhoneApplicationPage x:Class="C1WP7.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="Portrait" Orientation="Portrait"
shell:SystemTray.IsVisible="True" xmlns:my="clr-
namespace:C1.Phone.PdfViewer;assembly=C1.Phone.PdfViewer">
```

9. Edit the **TitlePanel** content to change the text in the **TextBlock** controls. It will appear similar to the following:

```
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="ComponentOne Studio for
Windows Phone" Style="{StaticResource PhoneTextNormalStyle}"/>
    <TextBlock x:Name="PageTitle" Text="PDF" FontSize="56" Margin="9,-
7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
```

10. Add the following markup between the `<Grid x:Name="ContentPanel"></Grid>` tags to add a **Button** and a **C1PdfViewer** control to the application:

```
<StackPanel>
    <Button x:Name="btnText" Content="Create PDF"
Click="HandleButtonClick" Margin="20" />
    <my:C1PdfViewer Name="pdfViewer" />
</StackPanel>
```

    You'll add code to initialize the **HandleButtonClick** event in the next step.

You've successfully created a Windows Phone application.

Step 2 of 4: Adding Content to the Page

In this step you'll switch to Code View and add some content to the document using the **DrawString** method.

1. Open the **MainPage.xaml.cs** file in Visual Studio and add the following code at the top of the page with the **using** statements:

```
using C1.Phone.Pdf;
using System.IO.IsolatedStorage;
using C1.Phone.PdfViewer;
```

2. Then add the following code just after the **MainPage** constructor:

```
public static void CreateDocumentText(C1PdfDocument pdf)
```

```
{
    // use landscape for more impact
    pdf.Landscape = true;

    // measure and show some text
    var text = "Hello World!";
    var font = new Font("Times New Roman", 60, PdfFontStyle.Italic);

    // create StringFormat used to set text alignment and line spacing
    var fmt = new StringFormat();
    fmt.LineSpacing = -1.5; // 1.5 char height
    fmt.Alignment = HorizontalAlignment.Center;

    // measure it
    var sz = pdf.MeasureString(text, font, 72 * 3, fmt);
    var rc = new Rect(pdf.PageRectangle.Width / 2, 72, sz.Width,
sz.Height);


    // draw the text
    pdf.DrawString(text, font, Colors.Black, rc, fmt);
}
```

This code creates content that will appear in the PDF file.

Step 3 of 4: Saving and Loading the Document

In this step you'll add code to save the document using the **Save** method and then display the PDF file the **C1PdfViewer** control.

Add the following code to the **MainPage.xaml.cs** just below the code added in the previous step:

```
public void HandleButtonClick(object sender, RoutedEventArgs e)
{
    // create document
    var pdf = new C1PdfDocument(PaperKind.Letter);
    pdf.Clear();

    // create document
    CreateDocumentText(pdf);

    // save document
    var fileName = "TestFile.pdf";
    var file = IsolatedStorageFile.GetUserStoreForApplication();
    using (var stream = file.CreateFile(fileName))
    {
        pdf.Save(stream);

    }
```

```
        using (var stream = file.OpenFile(fileName, System.IO.FileMode.Open))
        {
            this.pdfViewer.LoadDocument(stream);
            this.pdfViewer.ViewMode = C1.Phone.PdfViewer.ViewMode.FitWidth;
        }
    }
```

Now when the button is pressed, a PDF file will be created, saved to a stream, and then loaded into the **C1PdfViewer** control.

Step 4 of 4: Running the Application

In this step, you will run the application and view the PDF document you created. Complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view the application in the Windows Phone Emulator.
2. Click the **Create PDF** button to creat and display a PDF file. The application will appear similar to the following:

The code creates a **Font** object and calls the **C1PdfDocument.DrawString** method to write "Hello World!" on the page. This is exactly what you would do if you were writing to a **Graphics** object in .NET and is what makes **PDF for Windows Phone** so easy to use.

One important thing to remember is that **PDF for Windows Phone** uses a point-based coordinate system with the origin at the top-left corner of the page. This is similar to the default coordinate system used by .NET, but is different from the default PDF coordinate system (where the origin is on the bottom-left corner of the page). In this example, the top left point of the "H" in "Hello World" is located at [1,1].

Because the coordinate system is based on points, rather than pixels, **PDF for Windows Phone** uses **Rect**, **SizeF**, and **PointF** structures, which have members of type **float**, rather than **Rectangle**, **Size**, and **Point**, which have members of type **int**.

Congratulations! You have completed the **PDF for Windows Phone** quick start tutorial.

Using ComponentOne PDF for Windows Phone

The following topics provide details on how to add text, images, graphics, pages and overlays, bookmarks and annotations, and security and permissions to **PDF for Windows Phone** documents.

Adding Text

The following topics provide information on drawing, measuring, and managing the flow of text.

Drawing Text

Adding text to **PDF for Windows Phone** documents is easy – all the work is done by the **C1PdfDocument.DrawString** method.

C1PdfDocument.DrawString draws a given string at a specified location using a given font and brush. For example:

- Visual Basic
```
pdf.DrawString("Hello World!", font, Colors.Black, rect)
```
- C#
```
pdf.DrawString("Hello World!", font, Colors.Black, rect);
```

By default, **C1PdfDocument.DrawString** will align the text to the left and to the top of the given rectangle, will wrap the string within the rectangle, and will not clip the output to the rectangle. You can change all these options by specifying a *StringFormat* parameter in the call to **C1PdfDocument.DrawString**. The **StringFormat** has members that allow you to specify the horizontal alignment (**Alignment**), vertical alignment (**LineAligmnent**), and flags that control wrapping and clipping (**FormatFlags**).

For example, the code below creates a **StringFormat** object and uses it to align the text to the center of the rectangle, both vertically and horizontally:

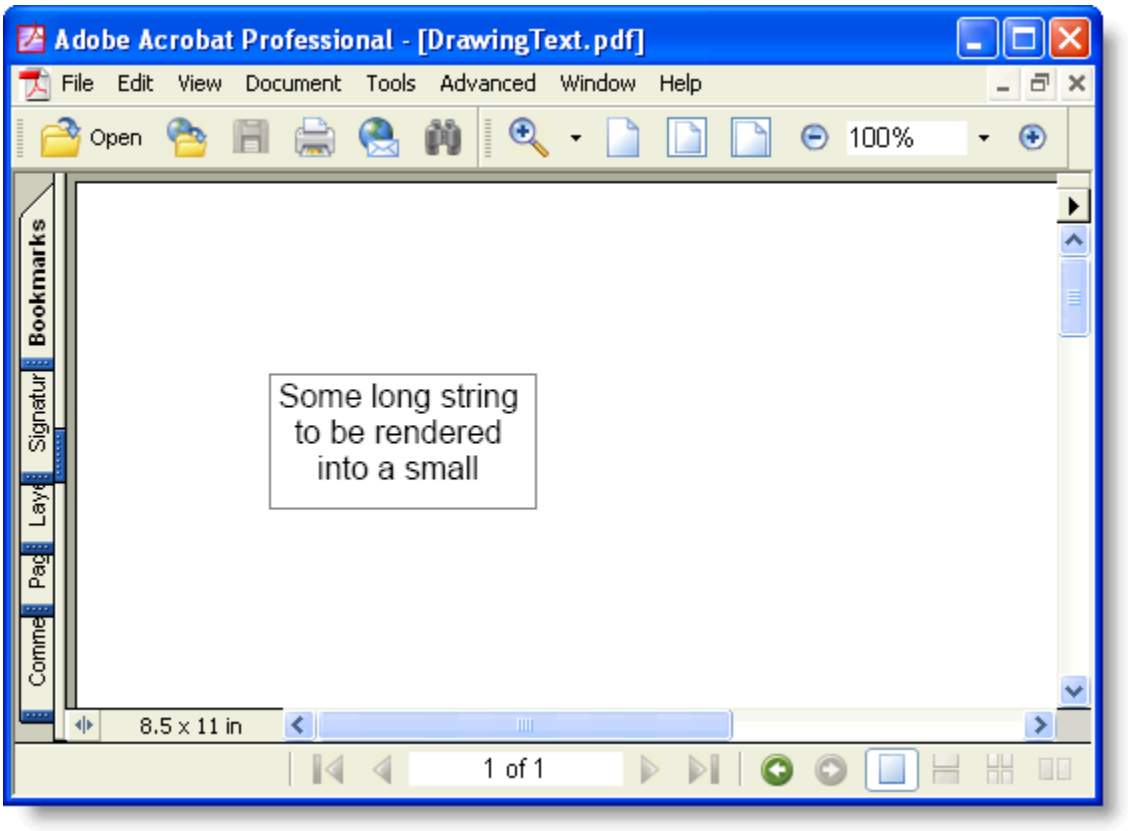
- Visual Basic
```
Dim font As New Font("Arial", 12)
Dim rect As New Rect(72, 72, 100, 50)
Dim text As String = "Some long string to be rendered into a small
rectangle. "
text = text & text & text & text & text & text

' Center align string.
```

```
Dim sf As New StringFormat()
sf.Alignment = HorizontalAlignment.Center
sf.LineAlignment = VerticalAlignment.Center
pdf.DrawString(text, font, Colors.Black, rect, sf)
pdf.DrawRectangle(Pens.Gray, rect)
```

- C#

```
Font font = new Font("Arial", 12);
Rect rect = new Rect(72, 72, 100, 50);
string text = "Some long string to be rendered into a small rectangle. ";
text = text + text + text + text + text + text;

// Center align string.
StringFormat sf = new StringFormat();
sf.Alignment = HorizontalAlignment.Center;
sf.LineAlignment = VerticalAlignment.Center;
pdf.DrawString(text, font, Colors.Black, rect, sf);
pdf.DrawRectangle(Pens.Gray, rect);
```

Here what the resulting PDF document looks like in Adobe's Acrobat Reader:



Measuring Text

In many cases, you will need to check whether the string will fit on the page before you render it. You can use the **C1PdfDocument.MeasureString** method for that. **C1PdfDocument.MeasureString** returns a **SizeF** structure that contains the width and height of the string (in points) when rendered with a given font.

For example, the code below checks to see if a paragraph will fit on the current page and creates a page break if it has to. This will keep paragraphs together on a page:

- Visual Basic

```
Private Function RenderParagraph(text As String, font As Font, rect As
Rect, rectPage As Rect) As Rect

    ' Calculate the necessary height.
    Dim sz As SizeF = _c1pdf.MeasureString(text, font, rect.Width)
    rect.Height = sz.Height

    ' If it won't fit this page, do a page break.
    If rect.Bottom > rectPage.Bottom Then
        _c1pdf.NewPage()
        rect.Y = rectPage.Top
    End If

    ' Draw the string.
    _c1pdf.DrawString(text, font, Colors.Black, rect)

    ' Update rectangle for next time.
    Rect.Offset(0, rect.Height)
    Return rect
End Function

' Use the RenderParagraph method.
Dim font As New Font("Arial", 10)
Dim rectPage As Rect = _c1pdf.PageRectangle()
rectPage.Inflate(-72, -72)
Dim rect As Rect = rectPage
Dim s As String
For Each s In  myStringList
    rect = RenderParagraph(s, font, rect, rectPage)
Next s
```

- C#

```
private Rect RenderParagraph(string text, Font font, Rect rect, Rect
rectPage)
{
    // Calculate the necessary height.
    SizeF sz = _c1pdf.MeasureString(text, font, rect.Width);
    rect.Height = sz.Height;

    // If it won't fit this page, do a page break.
    If (rect.Bottom > rectPage.Bottom)
    {
        _c1pdf.NewPage();
        rect.Y = rectPage.Top;
    }

    // Draw the string.
    _c1pdf.DrawString(text, font, Colors.Black, rect);

    // Update rectangle for next time.
    Rect.Offset(0, rect.Height);
    return rect;
}

// Use the RenderParagraph method.
Font font = new Font("Arial", 10);
```

```
Rect rectPage = _c1pdf.PageRectangle();
rectPage.Inflate(-72, -72);
Rect rect = rectPage;
foreach (string s in myStringList)
{
    rect = RenderParagraph(s, font, rect, rectPage);
}
```

Making Text Flow from Page to Page

The **C1PdfDocument.DrawString** method returns an integer. This is the index of the first character that was not printed because it did not fit the output rectangle. You can use this value make text flow from page to page or from one frame to another within a page. For example:

- Visual Basic

```
' Render a string spanning multiple pages.
While True

    ' Render as much as will fit into the rectangle.
    Dim nextChar As Integer
    nextChar = _c1pdf.DrawString(text, font, Colors.Black, rectPage)

    ' Break when done.
    If nextChar >= text.Length Then
        Exit While
    End If

    ' Get rid of the part that was rendered.
    Text = text.Substring(nextChar)

    ' Move on to the next page.
    _c1pdf.NewPage()
End While
```

- C#

```
// Render a string spanning multiple pages.
While (true)
{
    // Render as much as will fit into the rectangle.
    Int nextChar = _c1pdf.DrawString(text, font, Colors.Black, rectPage);

    // Break when done.
    If (nextChar >= text.Length)
    {
        break;
    }

    // Get rid of the part that was rendered.
    Text = text.Substring(nextChar);

    // Move on to the next page.
    _c1pdf.NewPage();
}
```

By combining the **C1PdfDocument.MeasureString** and **C1PdfDocument.DrawString** methods, you can develop rendering routines that provide extensive control over how paragraphs are rendered, including keeping paragraphs together on a page, keeping with the next paragraph, and controlling widows and orphans (single lines that render on the current or next page).

Adding Images

Adding images to **PDF for Windows Phone** documents is also easy, all the work is done by the **C1PdfDocument.DrawImage** method.

C1PdfDocument.DrawImage draws a given image at a specified location and has parameters that provide control over the image alignment and scaling. In the following example, this image is:

- Stretched to fill the given rectangle
- Center-aligned within the rectangle, scaled to keep the aspect ratio
- Aligned to the top-left corner of the rectangle, with the original size

This code is used to draw the same image three times:

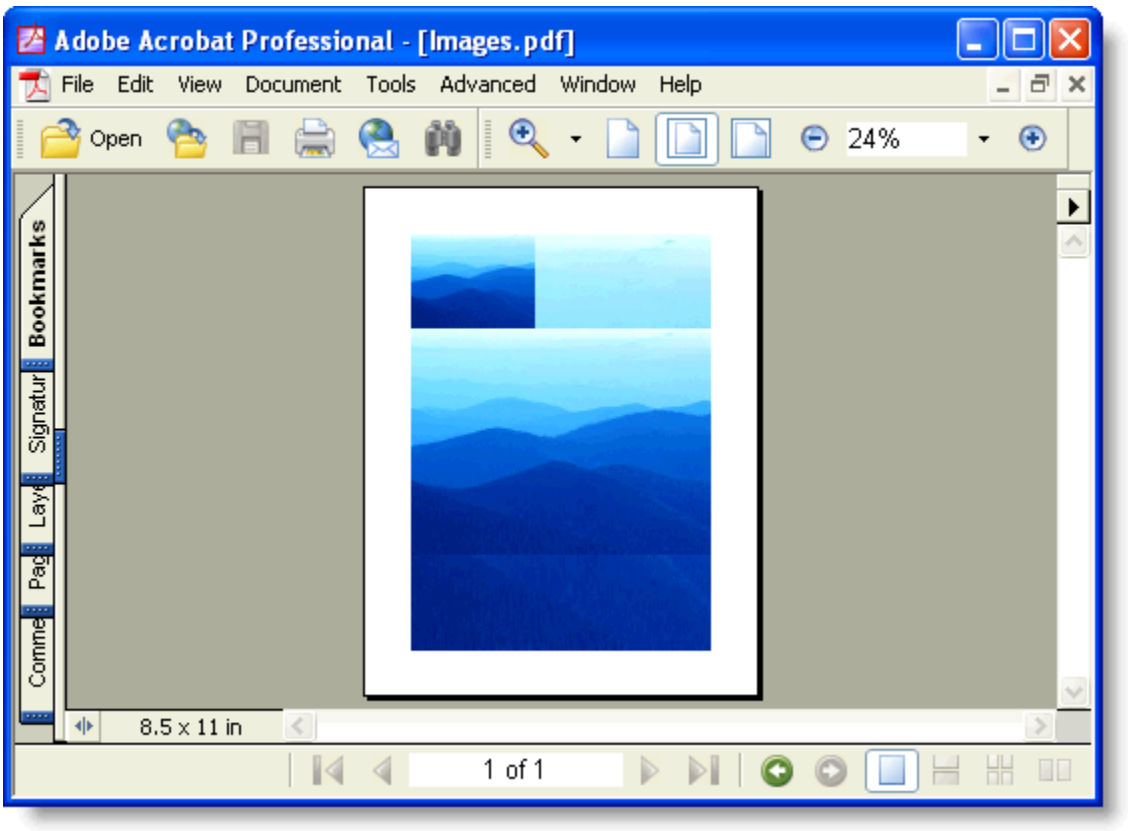
- Visual Basic

```
Dim rect As Rect = pdf.PageRectangle()
rect.Inflate(- 72, - 72)

' Stretch image to fill the rectangle.
pdf.DrawImage(pictureBox1.Image, rect)

' Center image within the rectangle, scale keeping aspect ratio.
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.MiddleCenter,
C1.Phone.Pdf.ImageSizeModeEnum.Scale)

' Render the image to the top left corner of the rectangle.
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.TopLeft,
C1.Phone.Pdf.ImageSizeModeEnum.Clip)
```

- C#

```
Rect rect = pdf.PageRectangle();
rect.Inflate(-72, -72);

// Stretch image to fill the rectangle.
pdf.DrawImage(pictureBox1.Image, rect);

// Center image within the rectangle, scale keeping aspect ratio.
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.MiddleCenter,
C1.Phone.Pdf.ImageSizeModeEnum.Scale);

// Render the image to the top left corner of the rectangle.
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.TopLeft,
1.C1Pdf.ImageSizeModeEnum.Clip);
```

The PDF document will look similar to this in Adobe's Acrobat Reader:



Notice that you can render any regular .NET Image object, including Metafiles. Metafiles are not converted into bitmaps; they are played into the document and thus retain the best possible resolution. If you want to add charts or technical drawings to your PDF document, Metafiles are better than bitmap images.

Bitmap images are managed automatically by **PDF for Windows Phone**. If you render the same image several times (in a page header for example), only one copy of the image is saved into the PDF file.

Adding Graphics

The **C1PdfDocument** class exposes several methods that allow you to add graphical elements to your documents, including lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and so on.

The methods are a subset of those found in the .NET **Graphics** class, and use the same **Brush** and **Pen** classes to control the color and style of the lines and filled areas.

It is important to remember that **PDF for Windows Phone** uses a coordinate system based on points, with the origin located at the top left of the page. (The default coordinate system for the .NET **Graphics** class is pixel-based.)

The example below illustrates how similar the graphics methods are between **PDF for Windows Phone** and the .NET **Graphics** class. The sample declares a **C1PdfDocument** class called 'g' and calls methods to draw pies, splines, and other graphical elements.

The point of the sample is that if you replaced the **C1PdfDocument** class with a regular .NET **Graphics** object, you would be able to compile the code and get the same results:

- Visual Basic

```
' Create PDF document.
Dim g As New C1.Phone.Pdf.C1PdfDocument()

' Set up to draw.
```

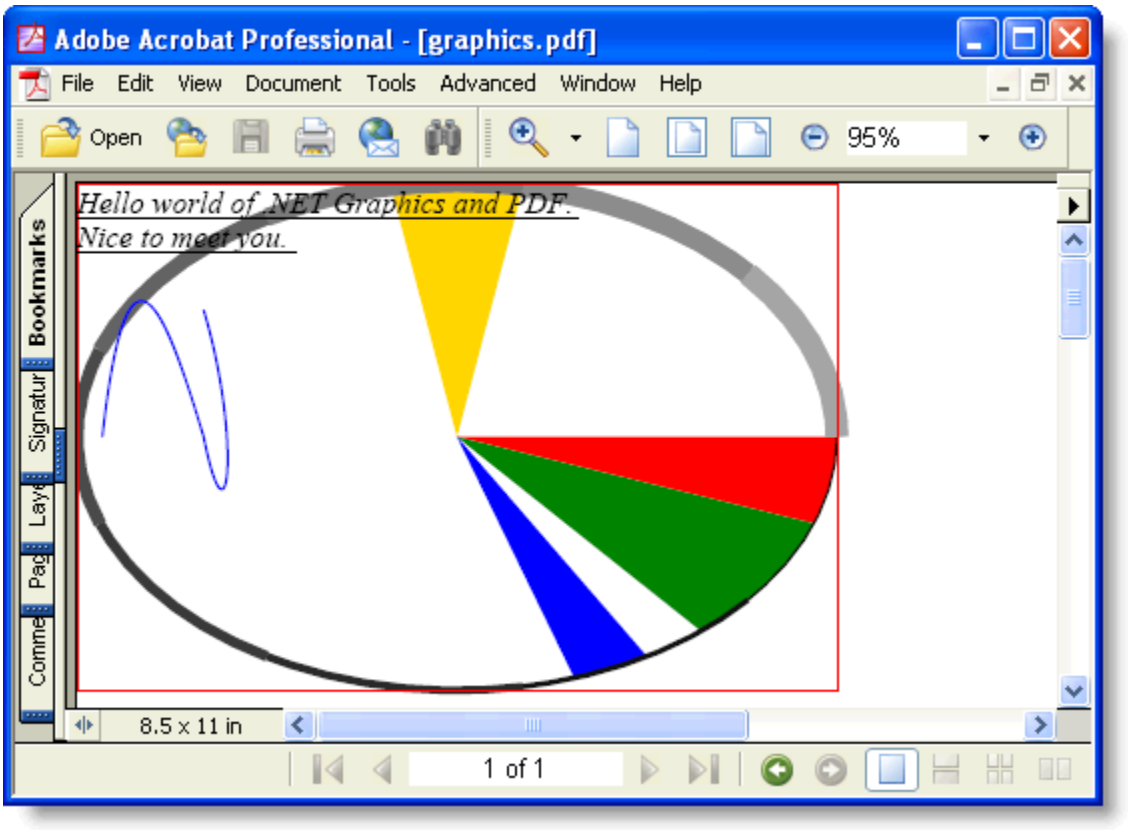
```
Dim rect As New Rect(0, 0, 300, 200)
Dim text As String = "Hello world of .NET Graphics and PDF." +
ControlChars.Cr + ControlChars.Lf + "Nice to meet you."
Dim font As New Font("Times New Roman", 12, FontStyle.Italic Or
FontStyle.Underline)
Dim bezierPoints() As PointF = {New PointF(10F, 100F), New PointF(20F,
10F), New PointF(35F, 50F), New PointF(50F, 100F), New PointF(60F, 150F),
New PointF(65F, 100F), New PointF(50F, 50F)}

' Draw some pie slices.
Dim penWidth As Integer = 0
Dim penRGB As Integer = 0
g.FillPie(Colors.Red, rect, 0, 20F)
g.FillPie(Colors.Green, rect, 20F, 30F)
g.FillPie(Colors.Blue, rect, 60F, 12F)
g.FillPie(Colors.Gold, rect, - 80F, - 20F)

' Draw some arcs.
Dim startAngle As Single
For startAngle = 0 To 360 - 40 Step 40
    Dim penColor As Color = Color.FromArgb(penRGB, penRGB, penRGB)
    penWidth = penWidth + 1
    Dim pen As New Pen(penColor, penWidth)
    penRGB = penRGB + 20
    g.DrawArc(pen, rect, startAngle, 40F)
Next

' Draw a rectangle and some bezier splines.
g.DrawRectangle(Pens.Red, rect)
g.DrawBeziers(Pens.Blue, bezierPoints)
g.DrawString(text, font, Colors.Black, rect)
```

- C#

```
// Create PDF document.
C1.Phone.Pdf.C1PdfDocument g = new C1.Phone.Pdf.C1PdfDocument();

// Set up to draw.
Rectangle rect = new Rect(0,0,300,200);
string text = "Hello world of .NET Graphics and PDF.\r\n" + "Nice to meet
you.";
Font font = new Font("Times New Roman", 12, FontStyle.Italic |
FontStyle.Underline);
PointF[] bezierPoints = new PointF[]
{
    new PointF(10f, 100f), new PointF(20f, 10f), new PointF(35f, 50f), new
PointF(50f, 100f), new PointF(60f, 150f), new PointF(65f, 100f), new
PointF(50f, 50f)
};

// Draw some pie slices.
int penWidth = 0;
int penRGB = 0;
g.FillPie(Colors.Red, rect, 0, 20f);
g.FillPie(Colors.Green, rect, 20f, 30f);
g.FillPie(Colors.Blue, rect, 60f, 12f);
g.FillPie(Colors.Gold, rect, -80f, -20f);
```

```
// Draw some arcs.
for (float startAngle = 0; startAngle < 360; startAngle += 40)
{
    Color penColor = Color.FromArgb(penRGB, penRGB, penRGB);
    Pen pen = new Pen(penColor, penWidth++);
    penRGB = penRGB + 20;
    g.DrawArc(pen, rect, startAngle, 40f);
}

// Draw a rectangle and some bezier splines.
g.DrawRectangle(Pens.Red, rect);
g.DrawBeziers(Pens.Blue, bezierPoints);
g.DrawString(text, font, Colors.Black, rect);
```

Here is the resulting PDF document looks like in Adobe's Acrobat Reader:



Creating Pages and Overlays

You may have noticed that in the previous examples, we started adding content to the document right after creating the **C1PdfDocument** object. This is possible because when you create the **C1PdfDocument**, it automatically adds an empty page to the document, ready to receive any type of content.

When you are done filling up the first page, you can add a new one using the NewPage method.

By default, all pages in the document have the same size and orientation. These parameters can be specified in the **C1PdfDocument** constructor. You can also change the page size and orientation at any time by setting the **C1PdfDocument.PaperKind**, **C1PdfDocument.PageSize**, and **C1PdfDocument.Landscape** properties. For example, the code below creates a document with all paper sizes defined by the **PaperKind** enumeration:

- Visual Basic

```
Dim font As New Font("Arial", 9)
Dim sf As New StringFormat()
sf.Alignment = HorizontalAlignment.Center
sf.LineAlignment = VerticalAlignment.Center

' Create one page with each paper size.
Dim firstPage As Boolean = True
Dim pk As PaperKind
For Each pk In System.Enum.GetValues(GetType(PaperKind))

    ' Skip custom size.
    If pk = PaperKind.Custom Then
        GoTo ContinueForEach1
    End If

    ' Add new page for every page after the first one.
    If Not firstPage Then
        _c1pdf.NewPage()
    End If
    firstPage = False

    ' Set paper kind.
    _c1pdf.PaperKind = pk

    ' Draw some content on the page.
    _c1pdf.DrawString("PaperKind: " + pk.ToString(), font, Colors.Black,
_c1pdf.PageRectangle(), sf)

    ContinueForEach1:
Next pk
```

- C#

```
Font font = new Font("Arial", 9);
StringFormat sf = new StringFormat();
sf.Alignment = HorizontalAlignment.Center;
sf.LineAlignment = VerticalAlignment.Center;

// Create one page with each paper size.
bool firstPage = true;
foreach (PaperKind pk in Enum.GetValues(typeof(PaperKind)))
{

    // Skip custom size.
    if (pk == PaperKind.Custom)
    {
        continue;
    }

    // Add new page for every page after the first one.
    if (!firstPage)
    {
        _c1pdf.NewPage();
    }
    firstPage = false;

    // Set paper kind.
```

```
    _c1pdf.PaperKind = pk;

    // Draw some content on the page.
    _c1pdf.DrawString("PaperKind: " + pk.ToString(), font, Colors.Black,
_c1pdf.PageRectangle(), sf);
}
```

You are not restricted to writing on the last page that was added to the document. You can use the **C1PdfDocument.CurrentPage** property to select which page you want to write to, and then use the regular drawing commands as usual. This is useful for adding content to pages after you are done rendering a document. For example, the code below adds footers to each page containing the current page number and the total of pages in the document (page n of m):

- Visual Basic

```
Private Sub AddFooters()
    Dim font As New Font("Helvetica", 7, FontStyle.Bold)
    Dim sf As New StringFormat()
    sf.Alignment = HorizontalAlignment.Center
    Dim page As Integer
    For page = 0 To  c1pdf.Pages.Count - 1

        ' Select page.
        _c1pdf.CurrentPage = page

        ' Build rectangle for rendering the footer.
        Dim rect As Rect = _c1pdf.PageRectangle()
        rect.Y = rect.Bottom

        ' Write the footer.
        Dim text As String
        text = String.Format("Page {0} of {1}", page + 1,
_c1pdf.Pages.Count)
        _c1pdf.DrawString(text, font, Colors.Gray, rect, sf)
    Next page
End Sub
```

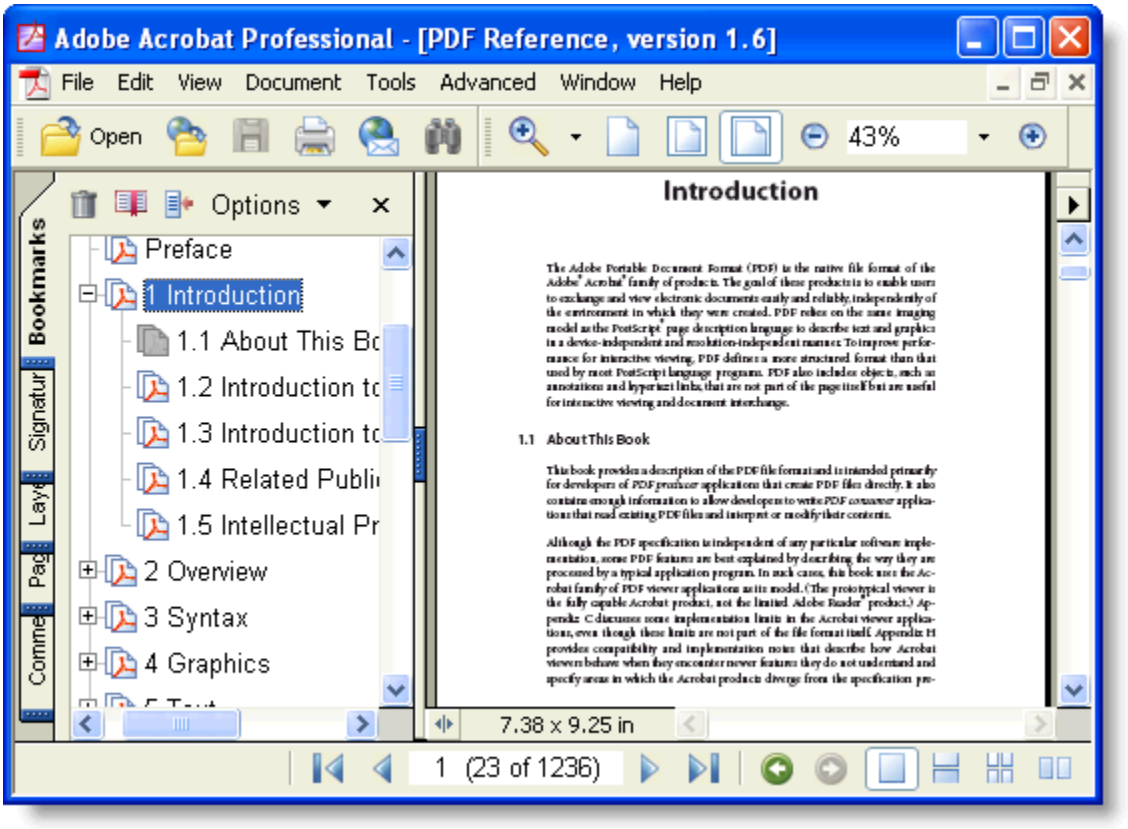
- C#

```
private void AddFooters()
{
    Font font = new Font("Helvetica", 7, FontStyle.Bold);
    StringFormat sf = new StringFormat();
    sf.Alignment = HorizontalAlignment.Center;
    for (int page = 0; page < _c1pdf.Pages.Count; page++)
    {
        // Select page.
        _c1pdf.CurrentPage = page;

        // Build rectangle for rendering the footer.
        Rect rect = _c1pdf.PageRectangle();
        rect.Y = rect.Bottom - 36;

        // Write the footer.
        string text = string.Format("Page {0} of {1}", page+1,
_c1pdf.Pages.Count);
        _c1pdf.DrawString(text, font, Colors.Gray, rect, sf);
    }
}
```

Note that the code uses the Pages property to get the page count. **Pages** is a collection based on the **ArrayList** class, and has methods that allow you to count and enumerate pages, as well as add and remove pages at specific positions. You can use the **Pages** collection to remove pages from certain locations in the document and re-insert them elsewhere.

Adding Bookmarks to a PDF Document

When you open a PDF document using Adobe's Acrobat Reader application, you will notice that most long documents contain an outline structure that is displayed on a pane on the left of the reader. The outline makes it easy to browse through a document's structure and find specific topics. The picture below shows a PDF document with an outline:



The outline entries are called Bookmarks, and you can add them to your **PDF for Windows Phone** documents using the **C1PdfDocument.AddBookmark** method. The **C1PdfDocument.AddBookmark** method takes three parameters: the title of the outline entry, the outline level, and the 'y' position of the entry on the current page (measured in points from the top of the page).

For example, the routine below adds a paragraph to a document and optionally marks it as a level-zero outline entry:

- Visual Basic

```
Private Function RenderParagraph(text As String, font As Font, rect As
Rect, rectPage As Rect, outline As Boolean) As Rect

    ' If it doesn't fit on this page, add a page break.
    rect.Height = _c1pdf.MeasureString(text, font, rect.Width).Height
    If rect.Bottom > rectPage.Bottom Then
        _c1pdf.NewPage()
```

```
        rect.Y = rectPage.Top
    End If

    ' Draw the string.
    _c1pdf.DrawString(text, font, Colors.Black, rect)

    ' Add headings to outline.
    If outline Then
        _c1pdf.DrawLine(Pens.Black, rect.X, rect.Y, rect.Right, rect.Y)
        _c1pdf.AddBookmark(text, 0, rect.Y)
    End If

    ' Update rectangle for next time.
    rect.Offset(0, rect.Height)
    Return rect
End Function
```

- C#

```
private Rect RenderParagraph(string text, Font font, Rect rect, Rect
rectPage, bool outline)
{
    // If it doesn't fit on this page, add a page break.
    rect.Height = _c1pdf.MeasureString(text, font, rect.Width).Height;
    if (rect.Bottom > rectPage.Bottom)
    {
        _c1pdf.NewPage();
        rect.Y = rectPage.Top;
    }

    // Draw the string.
    _c1pdf.DrawString(text, font, Colors.Black, rect);

    // Add headings to outline.
    if (outline)
    {
        _c1pdf.DrawLine(Pens.Black, rect.X, rect.Y, rect.Right, rect.Y);
        _c1pdf.AddBookmark(text, 0, rect.Y);
    }

    // Update rectangle for next time.
    rect.Offset(0, rect.Height);
    return rect;
}
```

Adding Links to a PDF Document

The PDF specification allows you to add several types of annotations to your documents. Annotations are often added by hand, as highlights and notes. But they can also be added programmatically. C1PdfDocument provides methods for adding hyperlinks, hyperlink targets, and file attachments to your PDF documents.

To add a hyperlink to your document, use the AddLink method. AddLink method takes two parameters: a string that specifies a *url* and a *Rect* that specifies the area on the current page that should behave as a link.

Note that the AddLink method does not add any visible content to the page, so you will usually need another command along with AddLink to specify some text or an image that the user can see. For example, the code below adds a string that says "Visit ComponentOne" and a link that takes the user to the ComponentOne home page:

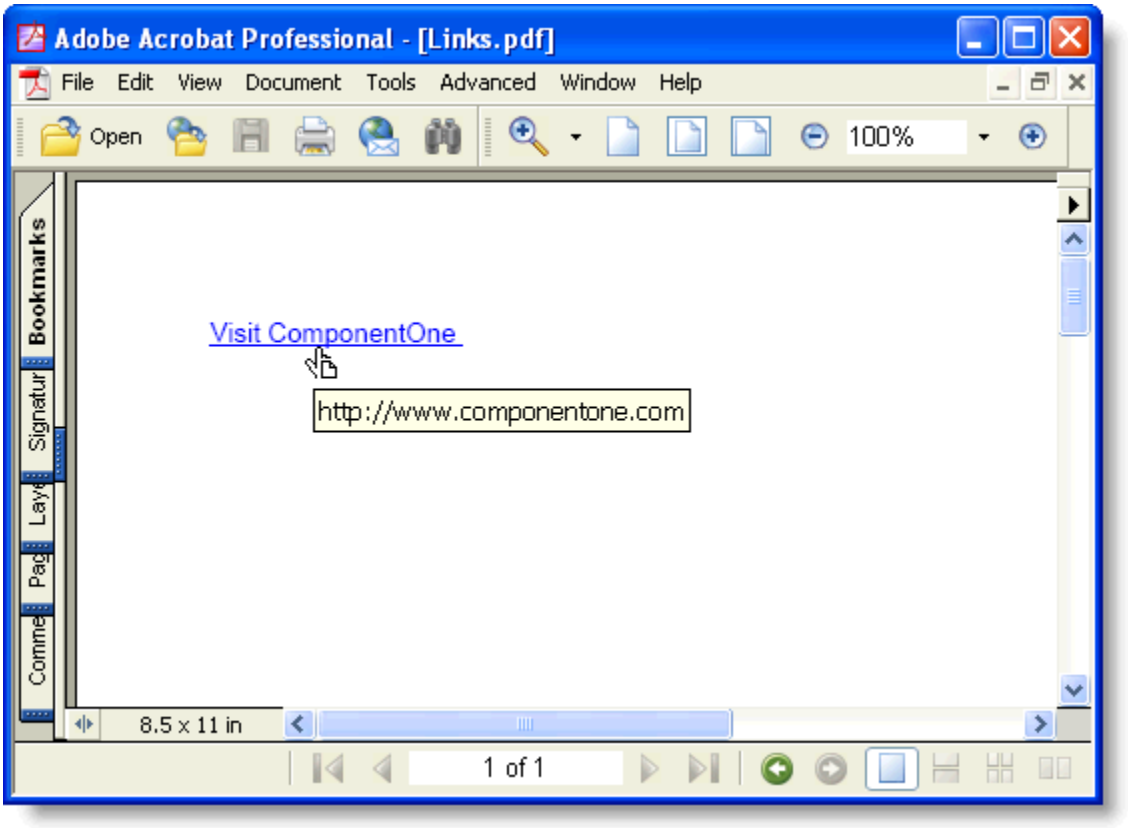
- Visual Basic

```
Dim rect As New Rect(50, 50, 100, 15)
Dim font As New Font("Arial", 10, FontStyle.Underline)
_c1pdf.AddLink("http://www.componentone.com", rect)
_c1pdf.DrawString("Visit ComponentOne", font, Colors.Blue, rect)
```

- C#

```
Rect rect = new Rect(50, 50, 100, 15);
Font font = new Font("Arial", 10, FontStyle.Underline);
_c1pdf.AddLink("http://www.componentone.com", rect);
_c1pdf.DrawString("Visit ComponentOne", font, Colors.Blue, rect);
```

Here is the resulting resulting PDF document looks like in Adobe's Acrobat Reader:



You can also add local links, which when clicked take the user to another location within the same PDF document. This type of link is useful when you want to implement some type of cross-referencing within the document, such as a table of contents or an index.

Local links are identical to regular hyperlinks, except for two things:

- The *url* parameter must start with a "#".
- You must specify the target location for the link using the AddTarget method. The AddTarget method takes the same parameters as AddLink, a string that specifies the name of the target and a rectangle that marks the area on the page that will be displayed when the user selects the link.

Attaching Files to a PDF Document

Adding file attachments to PDF files is often a useful feature. Attachments can contain any kind of file, including spreadsheets with detailed information that would clutter the main document, multimedia files with movies and sound, sample code, and so on.

Adding file attachments to your **PDF for Windows Phone** documents is easy. All you have to do is call the **C1PdfDocument.AddAttachment** method and specify which file you want to attach, what area of the page should contain the attachment, and optionally, the appearance of the attachment.

For example, the following code attaches all files in the application directory to the PDF document:

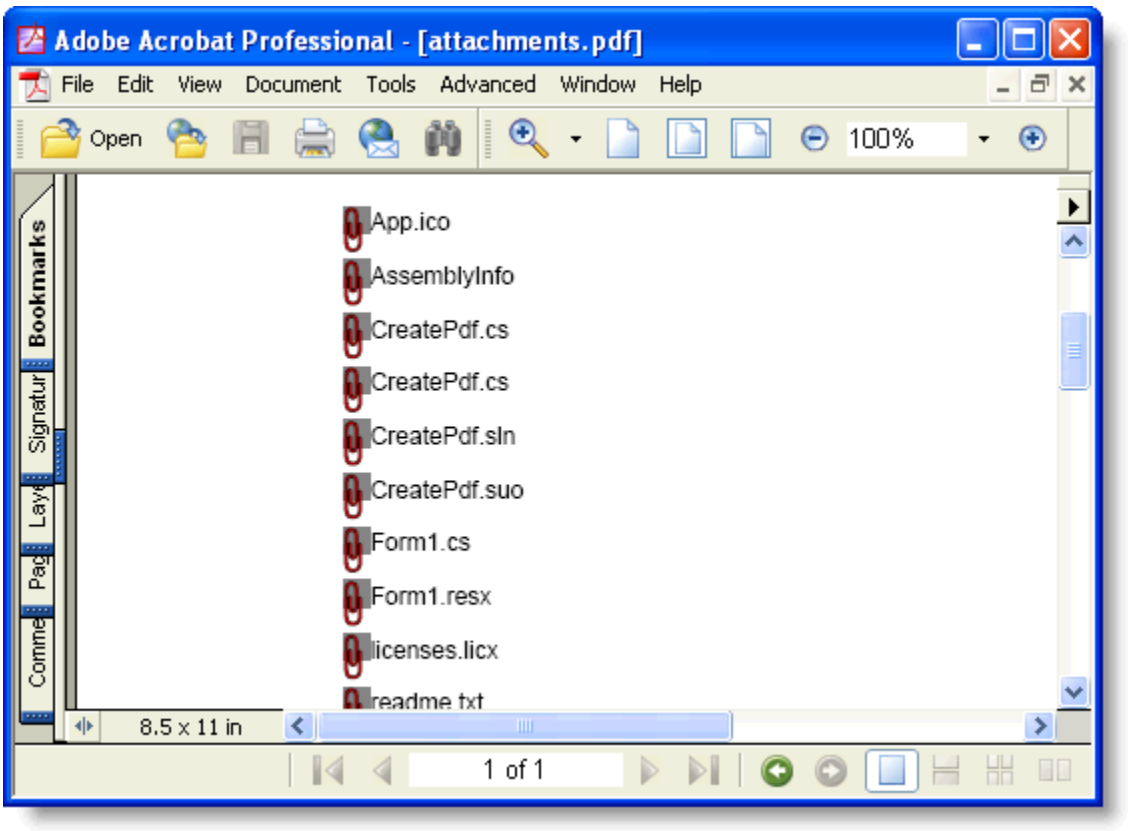
- Visual Basic

```
Dim rect As New Rect(100, 100, 60, 10)
Dim font As New Font("Arial", 9)

' Attach some files.
Dim path As String = "c:\temp\files"
Dim file As String
For Each file In Directory.GetFiles(path)
    Dim width As Single = rect.Width
    rect.Width = rect.Height
    _c1pdf.FillRectangle(Colors.Gray, rect)
    _c1pdf.AddAttachment(file, rect)
    rect.Width = width
    rect.X += rect.Height
    _c1pdf.DrawString(Path.GetFileName(file), font, Colors.Black, rect)
    rect.X -= rect.Height
    rect.Y += 2 * rect.Height
Next file
```

- C#

```
Rect rect = new Rect(100, 100, 60, 10);
Font font = new Font("Arial", 9);

// Attach some files.
string path = @"c:\temp\files";
string[] files = Directory.GetFiles(path);
foreach (string file in files)
{
    float width = rect.Width;
    rect.Width = rect.Height;
    _c1pdf.FillRectangle(Colors.Gray, rect);
    _c1pdf.AddAttachment(file, rect);
    rect.Width = width;
    rect.X += rect.Height;
    _c1pdf.DrawString(Path.GetFileName(file), font, Colors.Black, rect);
    rect.X -= rect.Height;
    rect.Y += 2 * rect.Height;
}
```

Here's what the resulting PDF document looks like in Adobe's Acrobat Reader:



The attachments are displayed as icons (you can select from four predefined icons in the **AttachmentIconEnum** enumeration and you can also select the icon color). When the user moves the mouse over the attachment, the file name is displayed and the cursor changes to indicate there is an attachment available. The user can then right-click the attachment name to open the attachment or save it.

Rendering Custom Content

The DrawingElement event allows you to customize the output generated by the DrawElement method. The event fires once for each FrameworkElement rendered after calls to the DrawElementmethod. It allows you to override the built-in rendering behavior and to render custom content for specific elements.

The code below shows how you can use the DrawingElement event to provide custom rendering for elements of type 'RichTextBox':

```
 // create C1PdfDocument
var pdf = new C1PdfDocument(PaperKind.Letter);

// use DrawingElement event to render RichTextBox elements
pdf.DrawingElement += (s, e) =>
{
if (e.Element is RichTextBox)
{
// get element image
#if SILVERLIGHT
var bmp = new WriteableBitmap(e.Element, e.DocumentTransform);
#else // WPF
var sz = e.Element.RenderSize;
var rtBmp = new RenderTargetBitmap((int)sz.Width, (int)sz.Height, 96,
```

```
96, PixelFormats.Pbgra32);
rtBmp.Render(e.Element);
var bmp = new WriteableBitmap(rtBmp);
#endif

// render it into the document
pdf.DrawImage(bmp, e.Bounds);

// done rendering this element
e.Handled = true;
}
};

// render LayoutRoot element into C1PdfDocument
var rc = pdf.PageRectangle;
rc.Inflate(-20, -20);
pdf.DrawElement(LayoutRoot, rc, ContentAlignment.TopLeft,
Stretch.None);
```

This example is quite general. You can use it to render any elements that do not expose their child elements as primitives such as TextBlock, Border, Rectangle, etc. The drawback associated with rendering elements as images is that the output will contain raster images, which can be quite large and may look rough when you zoom in on the PDF.

Applying Security and Permissions

By default, anyone can open, copy, print, and edit PDF files. If your PDF documents contain sensitive information, however, you can encrypt them so that only authorized users can access it.

There is a separate password for the owner of the document and for all other users. The user's access can be selectively restricted to allow only certain operations, such as viewing, printing, or editing the document.

PDF for Windows Phone provides a Security property that returns a **PdfSecurity** object. This object has properties that allow you to specify the **owner password** (required to change passwords and permissions for the document) and the **user password** (required to open the document). Additionally, the **PdfSecurity** object allows you to specify what permissions a regular user should have. For example you may allow users to see the document but not to print or edit it.

To use the **PDF for Windows Phone** security features, simply set the passwords before you save the document. For example:

- Visual Basic

```
' Create the document as usual.
CreateDoc()

' Set passwords.
_c1pdf.Security.OwnerPassword = "2mds%dffgd"
_c1pdf.Security.UserPassword = "anyone"
_c1pdf.Security.AllowEditAnnotations = False
_c1pdf.Security.AllowEditContent = False
_c1pdf.Security.AllowPrint = False
```

- C#

```
// Create the document as usual.
CreateDoc();

// Set passwords.
```

```
_c1pdf.Security.OwnerPassword = "2mds%dffgd";
_c1pdf.Security.UserPassword = "anyone";
_c1pdf.Security.AllowEditAnnotations = false;
_c1pdf.Security.AllowEditContent = false;
_c1pdf.Security.AllowPrint = false;
```

Save the document using the **Save** method as explained in the quick start topic, Step 3 of 4: Saving and Loading the Document.

Note that you can specify permissions and set only the owner password, leaving the user password empty. In this case, anyone will be allowed to open the document, but only the owner will be allowed to change the permissions.

Note also that the encryption scheme used by **PDF for Windows Phone** is public and is not 100% secure. There are ways to crack encrypted PDF documents. The security provided is adequate to protect your documents from most casual attacks, but if your data is truly sensitive you should not rely on PDF encryption alone.

Using Metafiles

PDF for Windows Phone makes it very easy to create documents, mainly because the object model mimics the well-known .NET **Graphics** model. However, not all methods available in the **Graphics** class are available in **PDF for Windows Phone**. Plus, you may have existing code that draws to a **Graphics** object and that you do not want to rewrite even if most methods are very similar.

In these cases, you can reuse your existing .NET code by sending the drawing commands to a **Metafile**, then rendering the **Metafile** into **PDF for Windows Phone** using the **C1PdfDocument.DrawImage** command. This method allows you to expose any graphics you create as images or as PDF documents.

For example, suppose you have an application that generates documents using the **PrintDocument** pattern of drawing each page into a **Graphics** object. You could then use the same methods to create a collection of metafiles, one per page, and then convert the list into a PDF document using the following code:

- Visual Basic

```
' Get the document as a list of Metafiles, one per page.
Dim pages As ArrayList = GetMetafiles()

' Loop through the pages and create a PDF document.
_c1pdf.Clear()
Dim i As Integer
for i = 0 i <= pages.Count

    ' Get ith page.
    Dim page As Metafile = CType(Metafile.FromFile(pages[i]), Metafile)
    If Not (page Is Nothing) Then

        ' Calculate the page size.
        Dim sz As SizeF = page.PhysicalDimension
        sz.Width = Math.Round(sz.Width * 72.0F / 2540.0F, 2)
        sz.Height = Math.Round(sz.Height * 72.0F / 2540.0F, 2)

        ' Add a page and set the size.
        If i > 0 Then
            _c1pdf.NewPage()
        End If
        _c1pdf.PageSize = sz

        ' Draw the page into the PDF document.
        _c1pdf.DrawImage(page, _c1pdf.PageRectangle())
    End If
Next
```

- *C#*

```
// Get the document as a list of Metafiles, one per page.
ArrayList pages = GetMetafiles();

// Loop through the pages and create a PDF document.
_c1pdf.Clear();
for (int i = 0; i < pages.Count; i++)
{

    // Get ith page.
    Metafile page = (Metafile)Metafile.FromFile(pages[i]);
    if (page == null)
    {
        continue;
    }

    // Calculate the page size.
    SizeF sz = page.PhysicalDimension;
    sz.Width  = (float)Math.Round(sz.Width  * 72f / 2540f, 2);
    sz.Height = (float)Math.Round(sz.Height * 72f / 2540f, 2);

    // Add a page and set the size.
    if (i > 0) _c1pdf.NewPage();
    _c1pdf.PageSize = sz;

    // Draw the page into the PDF document.
    _c1pdf.DrawImage(page, _c1pdf.PageRectangle());
}
```

Save the document using the **Save** method as explained in the quick start topic, Step 3 of 4: Saving and Loading the Document.

The code gets each metafile on the list, calculates its size in points (each page could have a different size), then draws the metafile into the page. The metafiles could be generated by a reporting engine, drawing or charting program, or any application that can create metafile images.