ComponentOne

PDF for UWP

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

Table of Contents

PDF for UWP

Easily create, print and email Adobe PDF documents with **PDF for UWP.** Create dynamic reports or directly output your UI to a PDF format with support for security, compression, outlining, hyper-linking, and attachments.

Getting Started

Help with UWP Edition

Getting Started

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

About PDF for UWP

Create Adobe PDF documents from your apps using **PDF for UWP**. The commands provided for adding content to documents are similar to the ones available in the .NET Graphics class. PDF for UWP 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 UWP 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 UWP. 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.

PDF for UWP Features

PDF for UWP supports most of the advanced features included in the PDF specification, including security, compression, outlining, hyperlinking, and attachments.

The following are some of the features of **PDF for UWP** that you may find useful:

- **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 UWP. Add text, images, lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and more.

- **Export Your UI**

  Export your UWP UI "as-is" directly to PDF with selectable text and images. This is an experimental feature which handles most common UI elements in the visual tree. Or you can simply export the UI to PDF using bitmaps. Just point the C1PdfDocument to your root visual element.

- **Familiar Syntax Using DrawImage Method**

  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**

  PDF 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. Add attachments to PDF files

- **Attachments**

  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 more. Adding an attachment to your PDF file is easy. Simply specify which file you want to attach, what area of the page should contain the attachment, and optionally, the appearance of the attachment.

- **Security and Permissions**

  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, printing, or editing the document.

- **Outline Structure**

  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 UWP, you can build this outline structure by adding outline entries (bookmarks).

- **Hyperlinks and Local links**

  PDF 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 and Viewer Preferences**

  PDF 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.

  enables export of JPEG2000 Images, provisions for digital signatures, and support for embedded fonts.

- **Support for TrueType fonts**

  PDF supports the Acrobat Reader built-in fonts: Times, Helvetica, and Symbol, as well as embedded and non-embedded TrueType fonts.

- **Support for PDF/A**

  PDF/A is commonly used by users creating invoices, brochures, manuals or research reports to store their reports to PDF/A formats. It enables export of JPEG2000 Images, provisions for digital signatures, and support for embedded fonts.

PDF for UWP Quick Start

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

In this quick start you will create a new project with a C1PdfDocument object, add content to the document, and save the document.

Step 1 of 4: Creating an Application with C1PdfDocument Object

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

1. In Visual Studio, select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select Blank App (Universal Windows).
3. Right-click the project name in the Solution Explorer and select **Add Reference**.
4. In the **Add Reference** dialog box, locate and select the **C1.UWP.Pdf.dll** and click **OK** to add the reference to your project.
5. Place your cursor between the **<Grid>** tags. Add the following markup to create a C1PdfViewer control:

   XAML

   ```
   <PdfViewer:C1PdfViewer Name="c1PdfViewer1" Grid.Row="1"/>
   ```

6. Then add the markup for a general ProgressRing control and a general Button control. This markup also adds a Click event so you can save your PDF document:

   XAML

   ```
   <ProgressRing x:Name="progressRing" Grid.Row="1"/>
   <Button x:Name="btnSave"  Content="Save PDF" Click="btnSave_Click"
   HorizontalAlignment="Right" VerticalAlignment="Top" Margin="0,70,140,0"
   Height="40" Width="106"/>
   ```

In this step, you created a Universal Windows application, added references, and added control to the application. In the next step, you'll add code to your application to create your PDF document.

Step 2 of 4: Adding Content to the Page

In this step you'll use code to add content to the document and format it.

1. Open the **MainPage.xaml.cs** file associated with your application. Add the following using statements to the top of your page:

   C#

   ```
   using C1.Xaml.Pdf;
   using Windows.UI;
   using Windows.Storage;
   using Windows.Storage.Pickers;
   using Windows.UI.Popups;
   ```

2. Create a **C1PdfDocument** by adding the following code directly below the page constructor:

```
C#
C1PdfDocument pdf;
```

3. Directly below the **InitializeComponent()** method, add a **MainPage_Loaded** method and create a new **C1PdfDocument**:

```
C#
this.Loaded += MainPage_Loaded;
pdf = new C1PdfDocument(PaperKind.Letter);
pdf.Clear();
```

4. Then add the MainPage_Loaded event:

```
C#
async void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    progressRing.IsActive = true;
    CreateDocumentText(pdf);
    await c1PdfViewer1.LoadDocumentAsync(PdfUtils.SaveToStream(pdf));
    progressRing.IsActive = false;
}
```

5. Below the added event, add the code that will add content to your Pdf document and format it:

```
C#
static void CreateDocumentText(C1PdfDocument pdf)
{
    // use landscape for more impact
    pdf.Landscape = true;
    // measure and show some text
    var text = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmood tempor incididunt ut labore magna aliqua. Ut enim ad minim veniam, quis
nostrud excecitation ullmanco laboris. nisi ut alquip ex ea commodo consequat.";
    var font = new Font("Segoe UI Light", 14, 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);
    rc = PdfUtils.Offset(rc, 110, 0);
    // draw a rounded frame
    rc = PdfUtils.Inflate(rc, 0, 0);
    pdf.FillRectangle(Windows.UI.Colors.Teal, rc, new Size(0, 0));
    //pdf.DrawRectangle(new Pen(Colors.DarkGray, 5), rc, new Size(0, 0));
    rc = PdfUtils.Inflate(rc, -10, -10);
    // draw the text
    pdf.DrawString(text, font, Windows.UI.Colors.White, rc, fmt);
```

```
    // now draw some text rotating about the center of the page
    rc = pdf.PageRectangle;
    rc = PdfUtils.Offset(rc, rc.Width / 2.2, rc.Height / 2.5);
    // build StringFormat used to rotate the text
    fmt = new StringFormat();
    // rotate the string in small increments
    var step = 6;
    text = "C1PDF works in Windows Runtime!";
    for (int i = 0; i <= 360; i += step)
    {
        fmt.Angle = i;
        font = new Font("Courier New", 8 + i / 30.0, PdfFontStyle.Bold);
        byte b = (byte)(255 * (1 - i / 360.0));
        pdf.DrawString(text, font, Windows.UI.Color.FromArgb(0xff, b, b, b), rc,
fmt);
    }
}
```

6. Add the code that will call the Save method you'll add in a separate code file:

C#

```
private void btnSave_Click(object sender, RoutedEventArgs e)
{
    PdfUtils.Save(pdf);
}
```

Step 3 of 4: Adding a Code File

In this step you'll add a code file that will contain the **Save** method, as well as code that defines both the **Offset** and the **Inflate** methods referenced in creating the PDF document's text.

1. Right-select the name of your application and select **Add | Class** from the context menu. The **Add New Item** dialog box appears.
2. Select **Class** from the list, and name it **PdfUtils.cs**. Click **OK**.
3. Add the following code in PdfUtils.cs file. The sample below adds the appropriate using statements, and all the code you'll need for the extension methods:

C#

```
using C1.Xaml.Pdf;
using System;
using System.Collections.Generic;
using Windows.Storage;
using Windows.Storage.Pickers;
using Windows.UI.Popups;
using System.IO;
using Windows.Foundation;

namespace Pdf_UWP_QS
{
    public static class PdfUtils
```

```
    {
        public static async void Save(this C1PdfDocument pdf)
        {
            FileSavePicker picker = new FileSavePicker();
            picker.FileTypeChoices.Add("PDF", new List<string>() { ".pdf" });
            picker.DefaultFileExtension = ".pdf";
            picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
            StorageFile file = await picker.PickSaveFileAsync();
            if (file != null)
            {
                await pdf.SaveAsync(file);
                MessageDialog dlg = new MessageDialog("Pdf Document saved to " +
file.Path, "PdfSamples");
                await dlg.ShowAsync();
            }
        }
        public static MemoryStream SaveToStream(this C1PdfDocument pdf)
        {
            MemoryStream ms = new MemoryStream();
            pdf.Save(ms);
            ms.Seek(0, SeekOrigin.Begin);
            return ms;
        }
        //
**************************************************************************
        // Extension methods for Rect
        //
**************************************************************************
        public static Rect Inflate(this Rect rc, double dx, double dy)
        {
            rc.X -= dx;
            rc.Y -= dy;
            rc.Width += 2 * dx;
            rc.Height += 2 * dy;
            return rc;
        }
        public static Rect Offset(this Rect rc, double dx, double dy)
        {
            rc.X += dx;
            rc.Y += dy;
            return rc;
        }
    }
}
```

In this step, you added a code file and the code for the extension methods. In the next step, you'll run your application.

Step 4 of 4: Running the Application

In this step, you will run the application and view the pdf document you created.

From the **Debug** menu, select **Start Debugging** to view the new PDF.

Congratulations!

You have completed the **PDF for UWP** quick start tutorial.

Using PDF for UWP

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 UWP** documents.

- Adding Text
- Adding Images
- Adding Graphics
- Creating Pages and Overlays
- Adding Bookmarks to a PDF Document
- Adding Links to a PDF Document
- Attaching Files to a PDF Document
- Applying Security and Permissions
- Using Metaflies

Adding Text

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

- Drawing Text
- Measuring Text
- Making Text Flow from Page to Page

Drawing Text

Adding text to **PDF for UWP** 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

VB

```
pdf.DrawString("Hello World!", font, Colors.Black, rect)
```

C#

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

VB

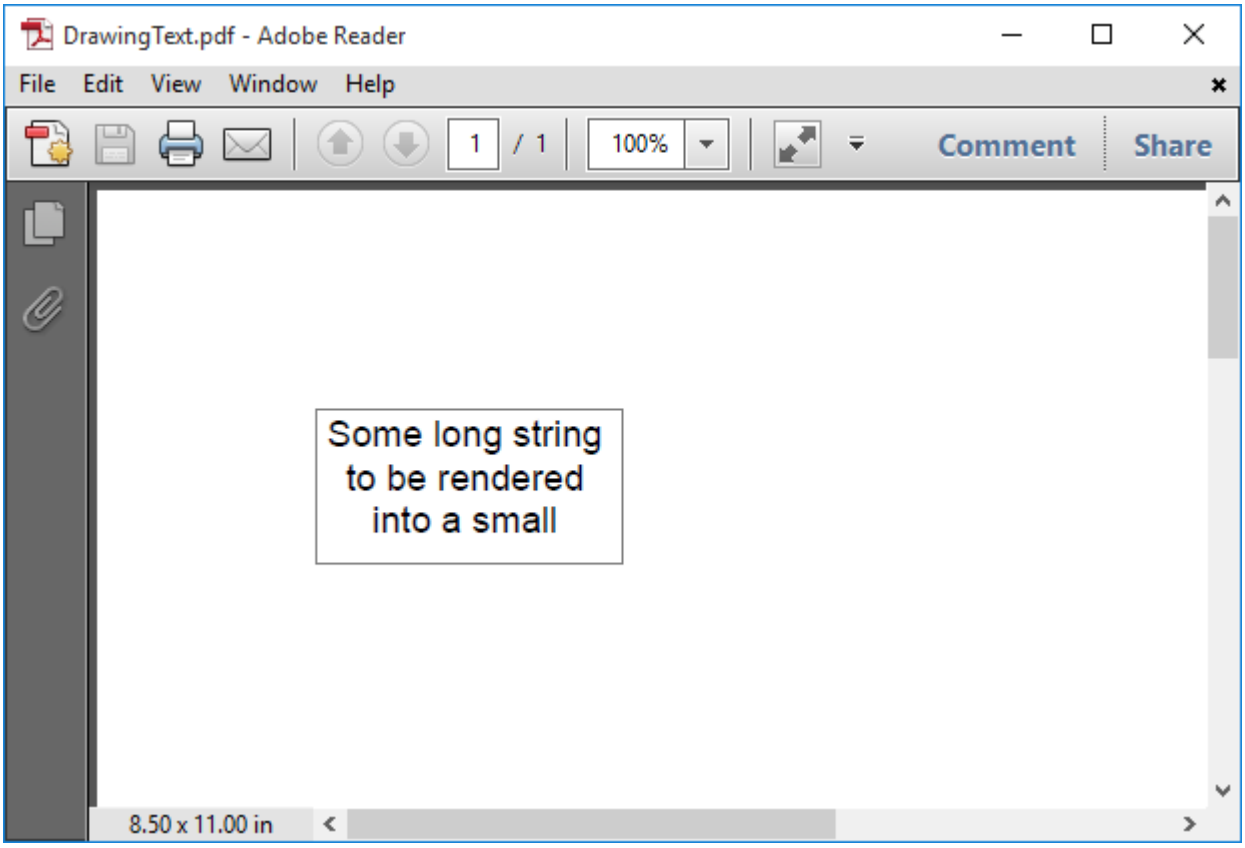
```
 Dim font As New Font("Arial", 12)
 Dim rect As New Rect(72, 72, 100, 50)
 Dim text As String = "Some long string to be rendered into a small rectangle. "
 text = Convert.ToString(Convert.ToString(Convert.ToString(Convert.ToString(text &
text) & text) & text) & text) & text
    ' Center align string.
 Dim sf As New StringFormat()
 sf.Alignment = HorizontalAlignment.Center
 sf.LineAlignment = VerticalAlignment.Center
 pdf.DrawString(text, font, Windows.UI.Colors.Black, rect, sf)
 pdf.DrawRectangle(Windows.UI.Colors.Gray, rect)
```

C#

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, Windows.UI.Colors.Black, rect, sf);
pdf.DrawRectangle(Windows.UI.Colors.Gray, rect);
```

Here is the resulting PDF document:



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 UWP** 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, the image is center-aligned within the rectangle and scaled to keep the aspect ratio. The sample declares a C1PdfDocument class called 'pdf' and calls method for drawing image.

This code is used to draw the same image as follows:

Visual Basic

VB

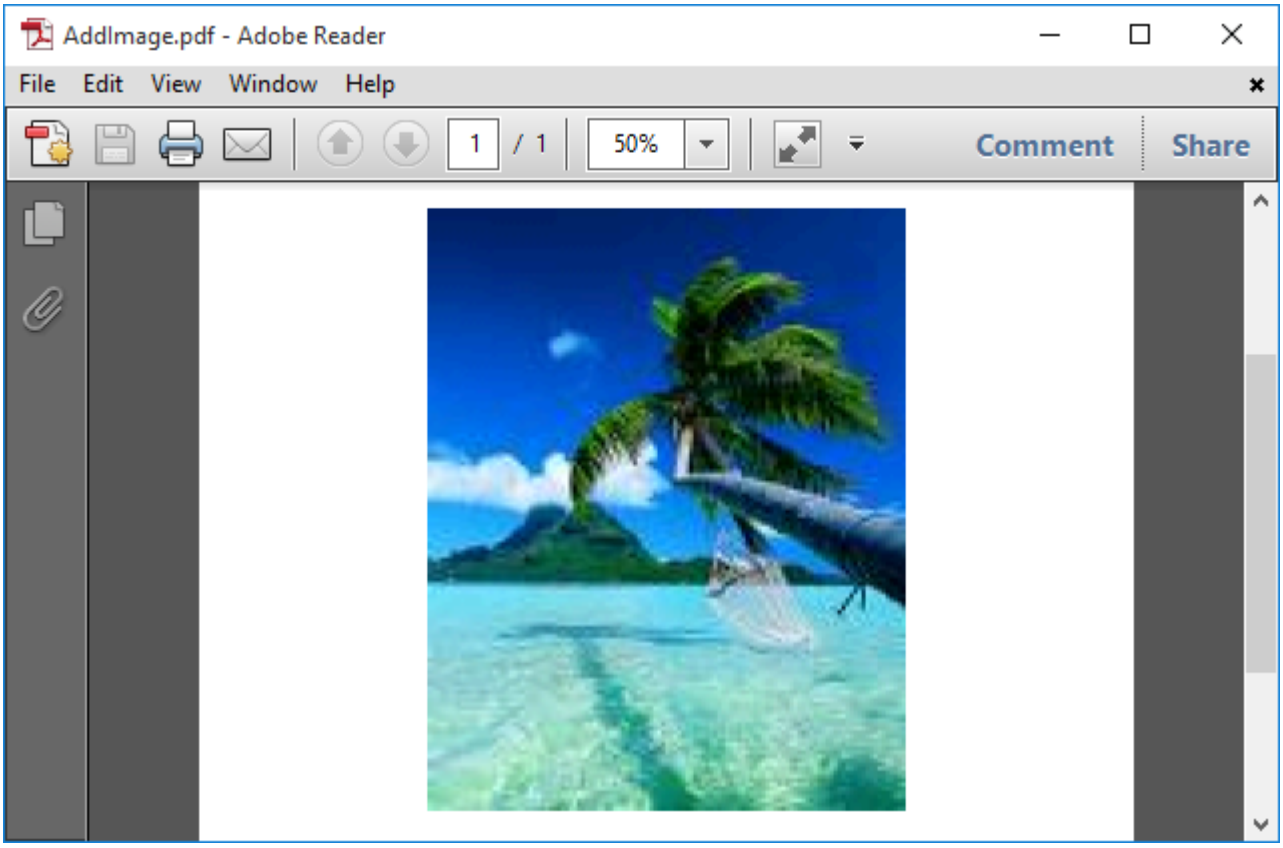
```
Dim rect As Rect = pdf.PageRectangle
rect = PdfUtils.Inflate(rect, -150, -150)
Dim ras As New InMemoryRandomAccessStream()
' load image into writeable bitmap
Dim wb As New WriteableBitmap(880, 660)
Dim ProjectFolder = Windows.ApplicationModel.Package.Current.InstalledLocation
Dim file As StorageFile = Await ProjectFolder.GetFileAsync("image.jpg")
wb.SetSource(Await file.OpenReadAsync())
Dim rcPic As New Rect(New Point(0, 0), New Point(pdf.PageSize.Width,
pdf.PageSize.Height))
' draw on page preserving aspect ratio
pdf.DrawImage(wb, rect, ContentAlignment.MiddleCenter, Stretch.Uniform)
```

C#

C#

```
Rect rect = pdf.PageRectangle;
rect = PdfUtils.Inflate(rect, -150, -150);

InMemoryRandomAccessStream ras = new InMemoryRandomAccessStream();

// load image into writeable bitmap
WriteableBitmap wb = new WriteableBitmap(880, 660);

var ProjectFolder = Windows.ApplicationModel.Package.Current.InstalledLocation;
StorageFile file = await ProjectFolder.GetFileAsync("image.jpg");

wb.SetSource(await file.OpenReadAsync());
Rect rcPic = new Rect(new Point(0, 0), new Point(pdf.PageSize.Width,
pdf.PageSize.Height));

// draw on page preserving aspect ratio
pdf.DrawImage(wb, rect, ContentAlignment.MiddleCenter, Stretch.Uniform);
```

The PDF document will look similar to this:



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 UWP**. 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 UWP** 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 UWP** and the .NET **Graphics** class. The sample declares a **C1PdfDocument** class called 'pdf' and calls methods to draw pies, Bezier curve, 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

VB

```
' set up to draw
Dim rc As New Rect(0, 0, 300, 200)
Dim text As String = "Hello world of .NET Graphics and PDF." & vbCr & vbLf & "Nice to
meet you."
Dim font As New Font("Times New Roman", 12, PdfFontStyle.Italic Or
PdfFontStyle.Underline)
' draw to pdf document
Dim penWidth As Integer = 0
Dim penRGB As Byte = 0
pdf.FillPie(Colors.Red, rc, 0, 20F)
pdf.FillPie(Colors.Green, rc, 20F, 30F)
pdf.FillPie(Colors.Blue, rc, 60F, 12F)
pdf.FillPie(Colors.Orange, rc, -80F, -20F)
For startAngle As Single = 0 To 359 Step 40
    Dim penColor As Color = Color.FromArgb(&Hff, penRGB, penRGB, penRGB)
    Dim pen As New Pen(penColor,
System.Math.Max(System.Threading.Interlocked.Increment(penWidth),penWidth - 1))
    penRGB = CByte(penRGB + 20)
    pdf.DrawArc(pen, rc, startAngle, 40F)
Next
pdf.DrawRectangle(Colors.Red, rc)
pdf.DrawString(text, font, Colors.Black, rc)
' show a Bezier curve
Dim pts = New Point() {New Point(400, 100), New Point(420, 30), New Point(500, 140),
New Point(530, 20)}
' draw Bezier
pdf.DrawBezier(New Pen(Colors.Blue, 4), pts(0), pts(1), pts(2), pts(3))
' show Bezier control points
pdf.DrawLines(Colors.Gray, pts)
For Each pt As Point In pts
    pdf.FillRectangle(Colors.Red, pt.X - 2, pt.Y - 2, 4, 4)
Next
```

C#

C#

```
// set up to draw
Rect rc = new Rect(0, 0, 300, 200);
string text = "Hello world of .NET Graphics and PDF.\r\nNice to meet you.";
Font font = new Font("Times New Roman", 12, PdfFontStyle.Italic |
PdfFontStyle.Underline);

// draw to pdf document
int penWidth = 0;
byte penRGB = 0;
pdf.FillPie(Colors.Red, rc, 0, 20f);
pdf.FillPie(Colors.Green, rc, 20f, 30f);
pdf.FillPie(Colors.Blue, rc, 60f, 12f);
pdf.FillPie(Colors.Orange, rc, -80f, -20f);
```

```
for (float startAngle = 0; startAngle < 360; startAngle += 40)
{
    Color penColor = Color.FromArgb(0xff, penRGB, penRGB, penRGB);
    Pen pen = new Pen(penColor, penWidth++);
    penRGB = (byte)(penRGB + 20);
    pdf.DrawArc(pen, rc, startAngle, 40f);
}
pdf.DrawRectangle(Colors.Red, rc);
pdf.DrawString(text, font, Colors.Black, rc);

// show a Bezier curve
var pts = new Point[]
{
    new Point(400, 100), new Point(420,  30),
    new Point(500, 140), new Point(530,  20),
};

// draw Bezier
pdf.DrawBezier(new Pen(Colors.Blue, 4), pts[0], pts[1], pts[2], pts[3]);

// show Bezier control points
pdf.DrawLines(Colors.Gray, pts);
foreach (Point pt in pts)
{
    pdf.FillRectangle(Colors.Red, pt.X - 2, pt.Y - 2, 4, 4);
}
```

Here is the resulting PDF document:

Specifying Page Sizes and Orientation

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 C1PdfDocumentBase.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

VB

```
Dim rect As New Rect(72, 72, 100, 50)
' create constant font and StringFormat objects
Dim font As New Font("Tahoma", 18)
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
For Each fi As var In GetType(PaperKind).GetFields(System.Reflection.BindingFlags.
[Static] Or System.Reflection.BindingFlags.[Public])
    ' Silverlight/Phone doesn't have Enum.GetValues
    Dim pk As PaperKind = DirectCast(fi.GetValue(Nothing), PaperKind)
    ' skip custom size
    If pk = PaperKind.[Custom] Then
        Continue For
    End If
    ' add new page for every page after the first one
    If Not firstPage Then
        pdf.NewPage()
    End If
    firstPage = False
    ' set paper kind and orientation
    pdf.PaperKind = pk
    pdf.Landscape = Not pdf.Landscape
    ' draw some content on the page
    Dim text As String = String.Format("PaperKind: [{0}];" & vbCr & vbLf &
"Landscape: [{1}];" & vbCr & vbLf & "Font: [Tahoma 18pt]", pdf.PaperKind,
pdf.Landscape)
    pdf.DrawString(text, font, Colors.Black, rect, sf)
    pdf.DrawRectangle(Colors.Black, rect)
Next
```

C#

C#

```
Rect rect = new Rect(72, 72, 100, 50);
// create constant font and StringFormat objects
Font font = new Font("Tahoma", 18);
StringFormat sf = new StringFormat();
sf.Alignment = HorizontalAlignment.Center;
sf.LineAlignment = VerticalAlignment.Center;

// create one page with each paper size
bool firstPage = true;
foreach (var fi in typeof(PaperKind).GetFields(System.Reflection.BindingFlags.Static
| System.Reflection.BindingFlags.Public))
{
    // Silverlight/Phone doesn't have Enum.GetValues
    PaperKind pk = (PaperKind)fi.GetValue(null);

    // skip custom size
    if (pk == PaperKind.Custom) continue;

    // add new page for every page after the first one
    if (!firstPage) pdf.NewPage();
```

```
    firstPage = false;

    // set paper kind and orientation
    pdf.PaperKind = pk;
    pdf.Landscape = !pdf.Landscape;

    // draw some content on the page

    string text = string.Format("PaperKind: [{0}];\r\nLandscape: [{1}];\r\nFont:
[Tahoma 18pt]",
        pdf.PaperKind, pdf.Landscape);
    pdf.DrawString(text, font, Colors.Black, rect, sf);
    pdf.DrawRectangle(Colors.Black, rect);
}
```

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

VB

```
Dim font As New Font("Tahoma", 7, PdfFontStyle.Bold)
Dim sf As New StringFormat()
sf.Alignment = HorizontalAlignment.Center
For page As Integer = 0 To pdf.Pages.Count - 1
    ' Select page.
    pdf.CurrentPage = page
    ' Build rectangle for rendering the footer.
    Dim rect As Rect = pdf.PageRectangle
    rect.Y = rect.Bottom - 36
    ' Write the footer.
    Dim text As String = String.Format("Page {0} of {1}", page + 1, pdf.Pages.Count)
    pdf.DrawString(text, font, Colors.Gray, rect, sf)
Next
```

C#

C#

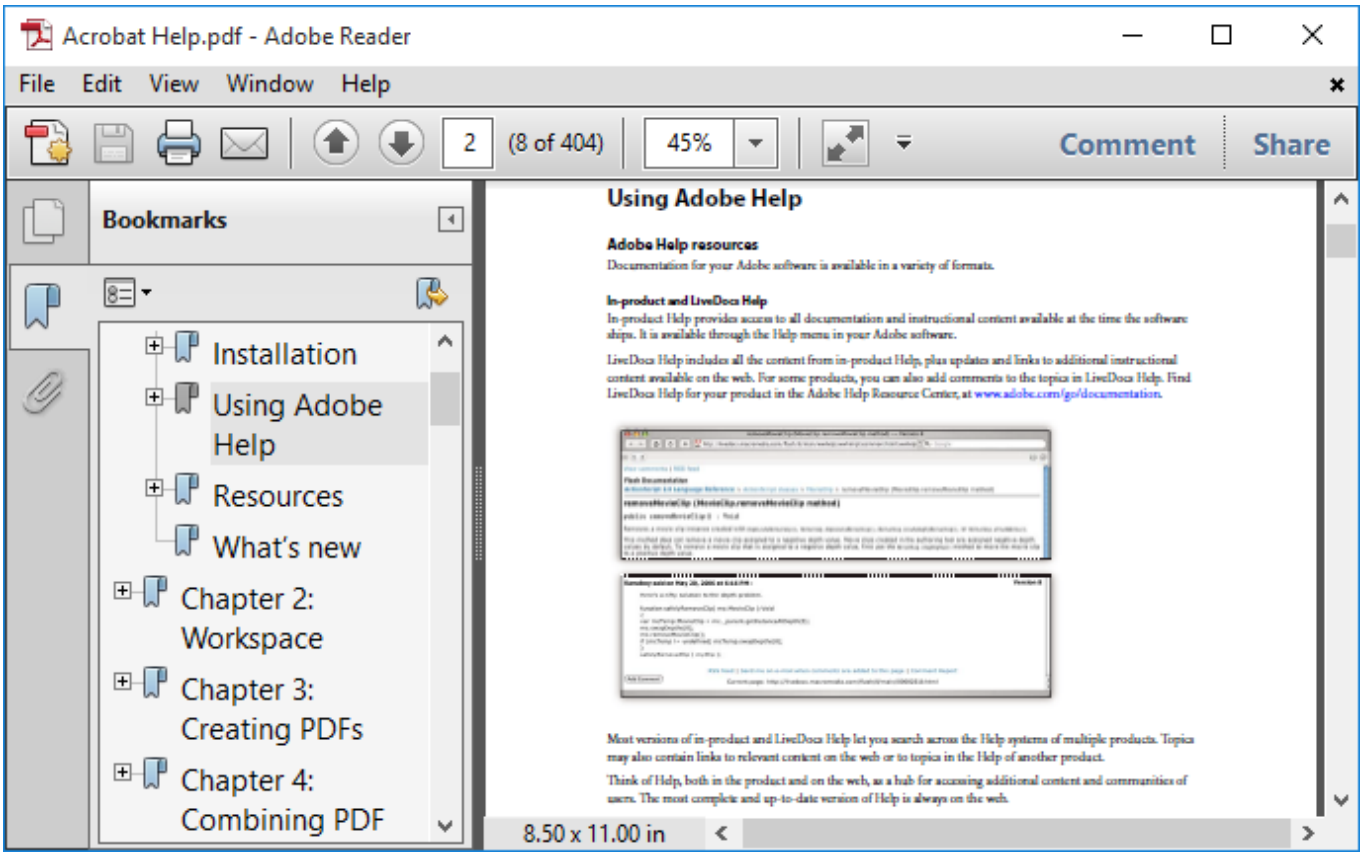
```
Font font = new Font("Tahoma", 7, PdfFontStyle.Bold);

StringFormat sf = new StringFormat();
sf.Alignment = HorizontalAlignment.Center;
for (int page = 0; page < pdf.Pages.Count; page++)
{
    // Select page.
    pdf.CurrentPage = page;
```

```
    // Build rectangle for rendering the footer.

    Rect rect = pdf.PageRectangle;

    rect.Y = rect.Bottom - 36;
    // Write the footer.
    string text = string.Format("Page {0} of {1}", page + 1, pdf.Pages.Count);
    pdf.DrawString(text, font, Colors.Gray, rect, sf);
}
```

Note that the code uses the C1PdfDocumentBase.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 UWP** 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

VB

```
 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 = pdf.MeasureString(text, font, rect.Width).Height
      If rect.Bottom > rectPage.Bottom Then
        pdf.NewPage()
        rect.Y = rectPage.Top
     End If

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

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

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

C#

Example Title

```
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 = new Rect(72, 72, 100, 50);
    rect.Height = pdf.MeasureString(text, font, rect.Width).Height;

    if (rect.Bottom > rectPage.Bottom)
    {
        pdf.NewPage();
        rect.Y = rectPage.Top;
    }

    // Draw the string
```

```
    pdf.DrawString(text, font, Windows.UI.Colors.Black, rect);

    // Add headings to outline
    if (outline)
    {
        pdf.DrawLine(Windows.UI.Colors.Black, rect.X, rect.Y, rect.Right, rect.Y);
        pdf.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 C1PdfDocument.AddLink method. **C1PdfDocument.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.

> The **C1PdfDocument.AddLink** method does not add any visible content to the page, so you will usually need another command along with **C1PdfDocument.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

VB

```
Dim rect As New Rect(50, 50, 100, 15)

Dim font As New Font("Arial", 10)

pdf.AddLink("http://www.componentone.com", rect)

pdf.DrawString("Visit ComponentOne", font, Colors.Blue, rect)
```

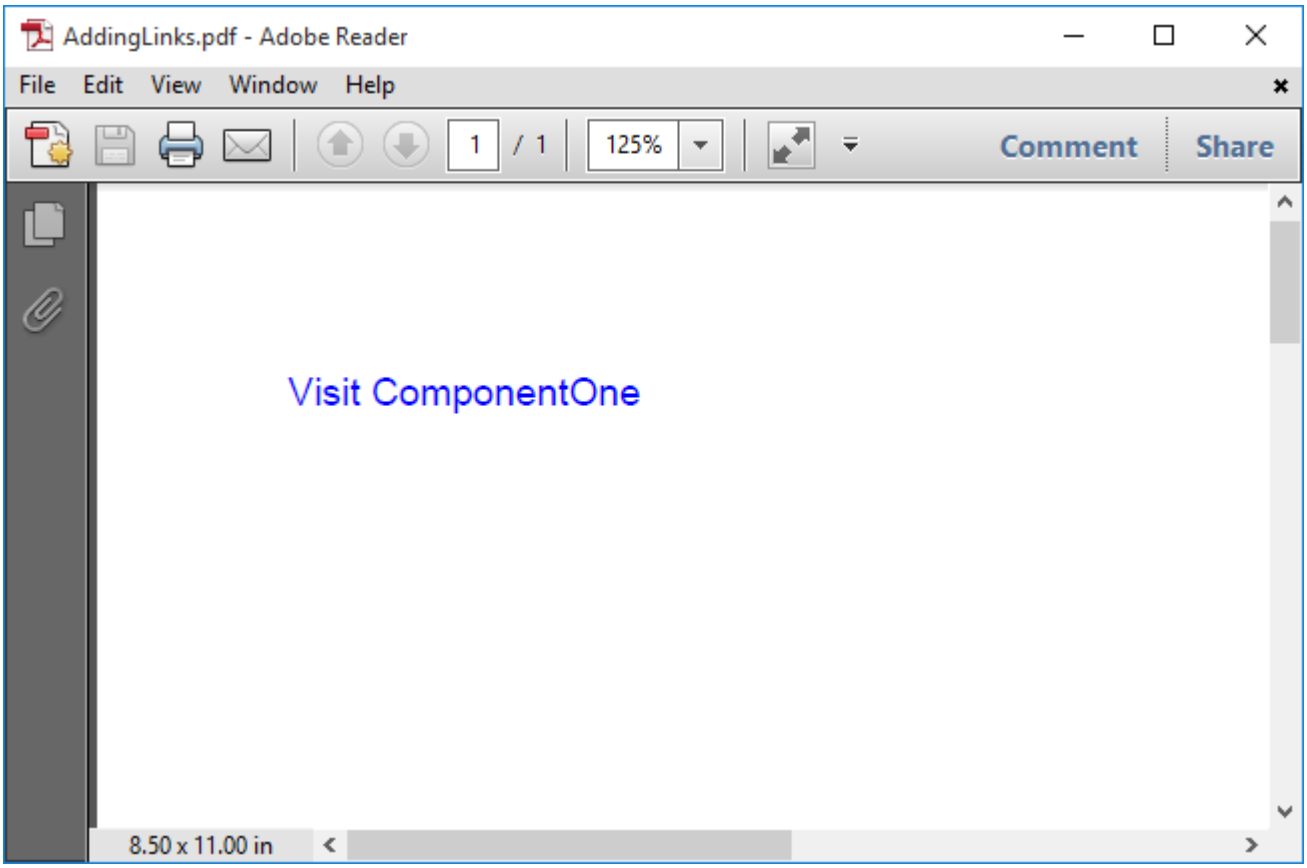
C#

Example Title

```
Rect rect = new Rect(50, 50, 100, 15);
Font font = new Font("Arial", 10);
pdf.AddLink("http://www.componentone.com", rect);
```

```
pdf.DrawString("Visit ComponentOne", font, Windows.UI.Colors.Blue, rect);
```

Here is the resulting PDF document:



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 C1PdfDocument.AddTarget method. The C1PdfDocument.AddTarget method takes the same parameters as C1PdfDocument.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 UWP** 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:

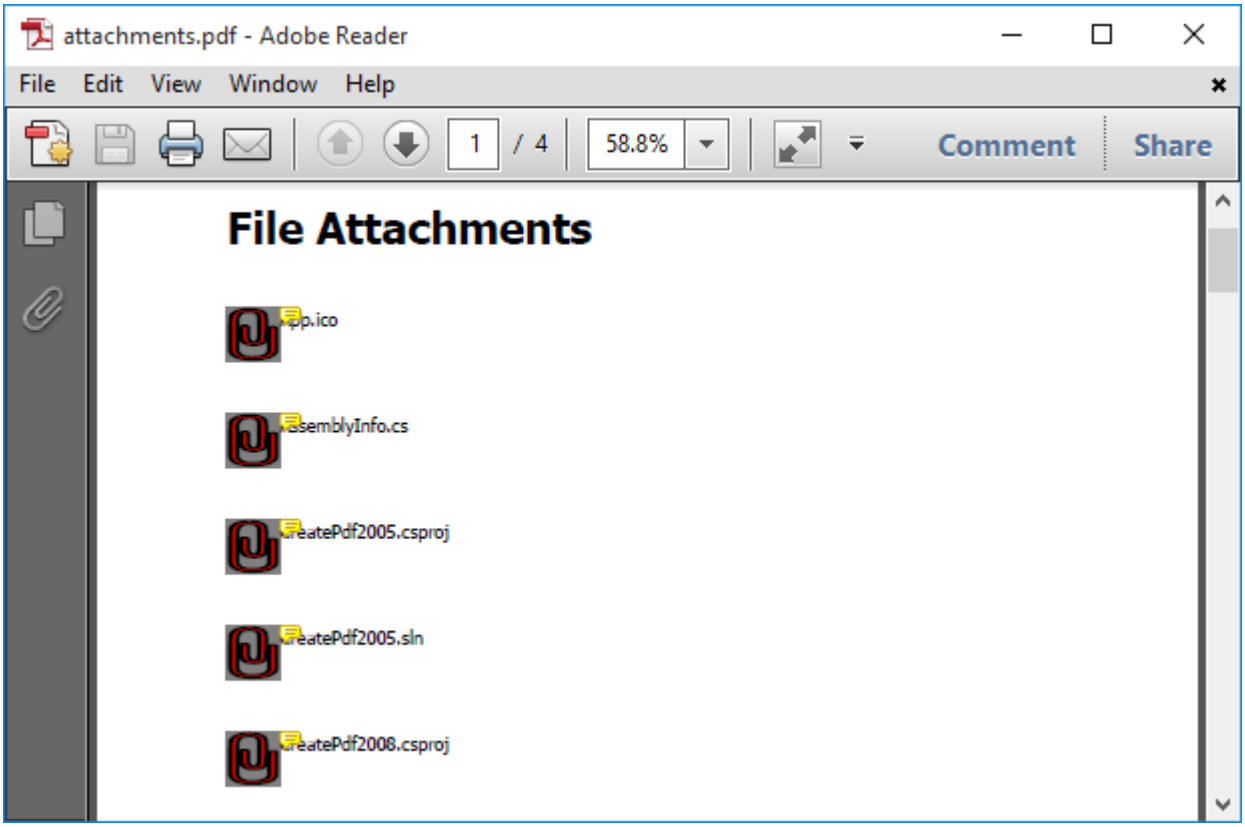
Visual Basic

```
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 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.

Applying Security and Permissions

> The encryption scheme used by **PDF for UWP** 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.

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 UWP provides a C1PdfDocumentBase.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 UWP** 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 the document.

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.

Using Metafiles

PDF for UWP 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 UWP**. 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 UWP** 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 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.