ComponentOne

PDF for WPF and Silverlight

GrapeCity US

GrapeCity
201 South Highland Avenue, Suite 301
Pittsburgh, PA 15206
Tel: 1.800.858.2739 | 412.681.4343
Fax: 412.681.4384
Website: https://www.grapecity.com/en/
E-mail: us.sales@grapecity.com

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 $25 (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 WPF and Silverlight Overview

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

Getting Started

Easily create, print, and email Adobe PDF documents with **PDF for WPF and Silverlight**. Create dynamic reports or directly output your UI to a PDF format with support for security, compression, outlining, hyperlinking, 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 WPF version is that it does not support Acrobat Forms and text annotations.

Help with WPF and Silverlight Edition

Getting Started

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

Key Feature

PDF for WPF and Silverlight 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 **C1Pdf** that you may find useful:

- **Fast rendering and compression of images in Metafiles**
  Metafiles are not converted into bitmaps; they 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.

- **Familiar syntax using the 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.

- **Manage document restrictions**
  Allow users to copy and edit content, restrict users from printing the document, and more.

- **Add attachments to PDF files**
  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.

- **Owner and user password protection**
  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.

- **Add graphical elements**
  Add lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and more.

- **Create an 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 WPF, you can build this outline structure by adding outline entries (bookmarks).

- **Add 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.

- **Control 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.

- **Support for TrueType fonts**
  PDF for WPF and Silverlight now supports Acrobat Reader built-in fonts: Time, Helvetica, and Symbol. The C1PdfDocument class also supports embedded and non-embedded TrueType fonts.

- **Support for PDF/A**
  PDF for WPF and Silverlight now supports PDF/A format, which is commonly used for creating invoices, brochures, manuals or research reports. PDF/A format easily exports JPEG2000 images, supports embedded fonts and has provisions for digital signatures as well. The control also comes with PDF/A compliance to support PDF annotations.

Quick Start

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

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.

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

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

1. In Visual Studio, select **File | New | Project**.
2. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **WPF Application** or **Silverlight Application**. Enter a Name for your project and click **OK**. The New WPF or Silverlight Application dialog box will appear.
3. Click **OK** to close the **New WPF Application** or **New Silverlight Application** dialog box and create your project.
4. Right-click the project name in the Solution Explorer and select **Add Reference**.
5. In the Add Reference dialog box, locate and select the C1.WPF.Pdf.dll or the C1.Silverlight.Pdf.dll and click **OK** to add the reference to your project.
6. 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.Silverlight.Pdf;
using System.IO;
```

7. Then add the following code for the MainPage:

```
// Create the C1PdfDocument object
C1PdfDocument pdf = new C1PdfDocument();
public MainPage()
{
    InitializeComponent();
    // Make the window hit test visible.
    LayoutRoot.Background = new SolidColorBrush(Colors.Transparent);
}
```

You've successfully created a WPF or Silverlight application with a **C1PdfDocument** object.

Step 2 of 4: Adding Content to the Page

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

In the **MainPage.xaml.cs** file immediately following the new **C1PdfDocument** object you created in step 1, add the following code:

To write the code in Visual Basic:

```
Protected Overrides Sub OnMouseLeftButtonDown(e As MouseButtonEventArgs)

' Add content to the page
Dim rc As Rect = pdf.PageRectangle
rc.X = InlineAssignHelper(rc.Y, 72)
Dim font As New Font("Arial", 12)
pdf.DrawString("Hello World!", font, Colors.Black, rc)
```

End Sub

To write the code in C#:

```
protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
    // Add content to the page
    Rect rc = pdf.PageRectangle;
    rc.X = rc.Y = 72;
    Font font = new Font("Arial", 12);
    pdf.DrawString("Hello World!", font, Colors.Black, rc);
}
```

Step 3 of 4: Saving the document

In this step you'll save the document using the **Save** method.

In the **MainPage.xaml.cs** file immediately following the code you used to add content to the document in step 2, add the following code:

C#

```
// Save the document

SaveFileDialog dlg = this.GetFileDialog();
{
        if (dlg != null)
        {
            using (Stream stream = dlg.OpenFile())
            {
                pdf.Save(stream);
            }
        }
    }

    private SaveFileDialog GetFileDialog()
    {
        SaveFileDialog dlg = new SaveFileDialog();
        dlg.DefaultExt = ".pdf";
        bool? dr = dlg.ShowDialog();
        return ((dr.HasValue && dr.Value) ? dlg : null);
    }
```

Step 4 of 4: Running the Application

In this step, you will run the application and view the hello world.pdf document.
From the **Debug** menu, select **Start Debugging** to view the new PDF.

Features: PDF for WPF and Silverlight

The following sections provide details about C1Pdf's features.

Adding Text

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

Drawing Text

Adding text to **PDF for WPF** 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 C1.WPF.Pdf.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 C1.WPF.Pdf.StringFormat()
```

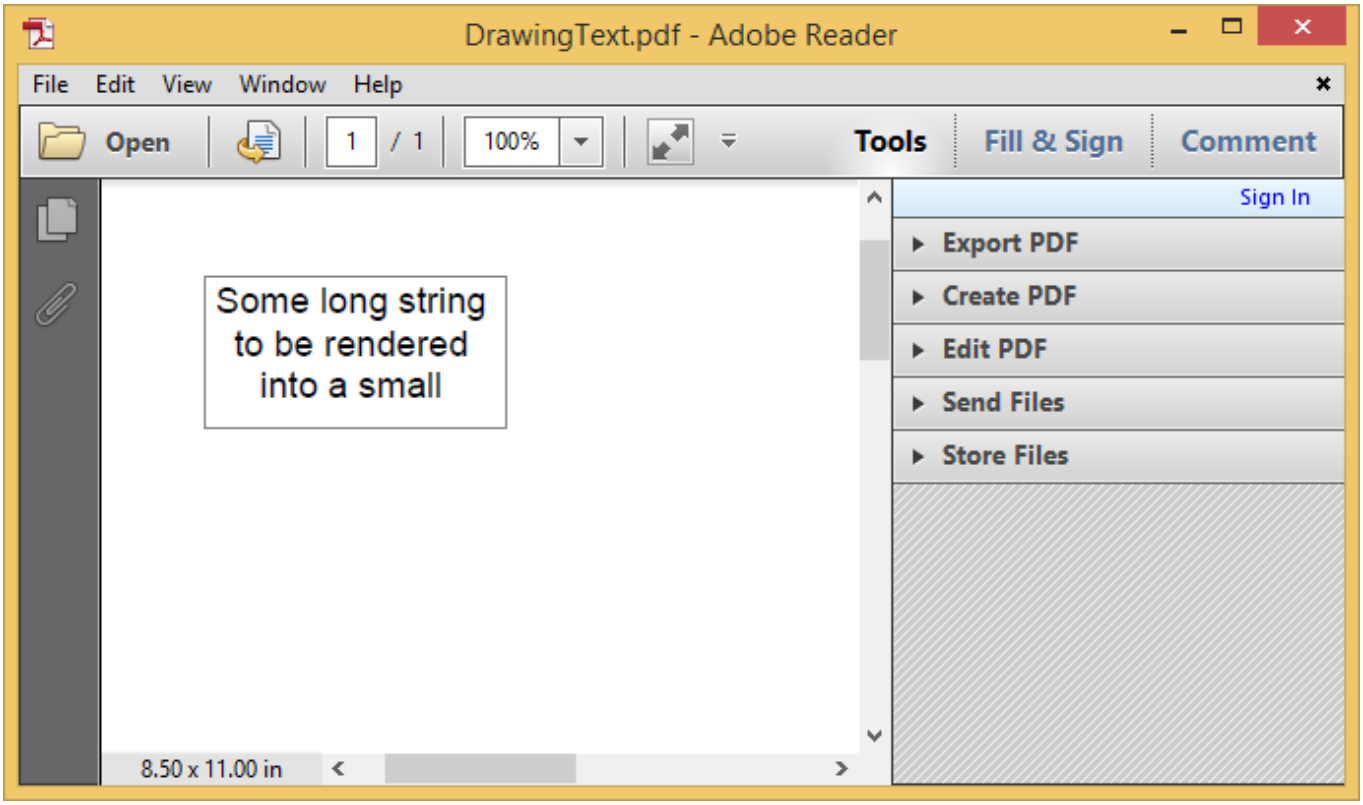
```
sf.Alignment = HorizontalAlignment.Center
sf.LineAlignment = VerticalAlignment.Center
pdf.DrawString(text, font, Colors.Black, rect, sf)
pdf.DrawRectangle(Colors.Gray, rect)
```

C#

C# copyCode

```
C1.WPF.Pdf.Font font = new C1.WPF.Pdf.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.
C1.WPF.Pdf.StringFormat sf = new C1.WPF.Pdf.StringFormat();
sf.Alignment = HorizontalAlignment.Center;
sf.LineAlignment = VerticalAlignment.Center;
pdf.DrawString(text, font, Colors.Black, rect, sf);
pdf.DrawRectangle(Colors.Gray, rect); //System.Windows.Media.Colors.Gray;
```

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:

To write the code in 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
```

To write the code in 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

VB

```
Dim font As New C1.WPF.Pdf.Font("Arial", 12)
Dim rect As New Rect(72, 72, 100, 50)
Dim text As String = "This is the random text...This is the random text...This is the random text...This is the random text..."
```

```
' Render a string spanning multiple pages.
While True
    ' Render as much as will fit into the rectangle.
    Dim nextChar As Integer = pdf.DrawString(text, font, Colors.Black, rect)
    pdf.DrawRectangle(Colors.LightGray, rect)
    ' 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.
    pdf.NewPage()
End While
```

C#

C# copyCode

```
C1.WPF.Pdf.Font font = new C1.WPF.Pdf.Font("Arial", 12);

Rect rect = new Rect(72, 72, 100, 50);
string text = "This is the random text...This is the random text...This is the random text...This is the random text...";

 // Render a string spanning multiple pages.
 while (true)
 {
      // Render as much as will fit into the rectangle.
     int nextChar = pdf.DrawString(text, font, Colors.Black, rect);

     pdf.DrawRectangle(Colors.LightGray, rect);

     // 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.
     pdf.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 WPF** 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

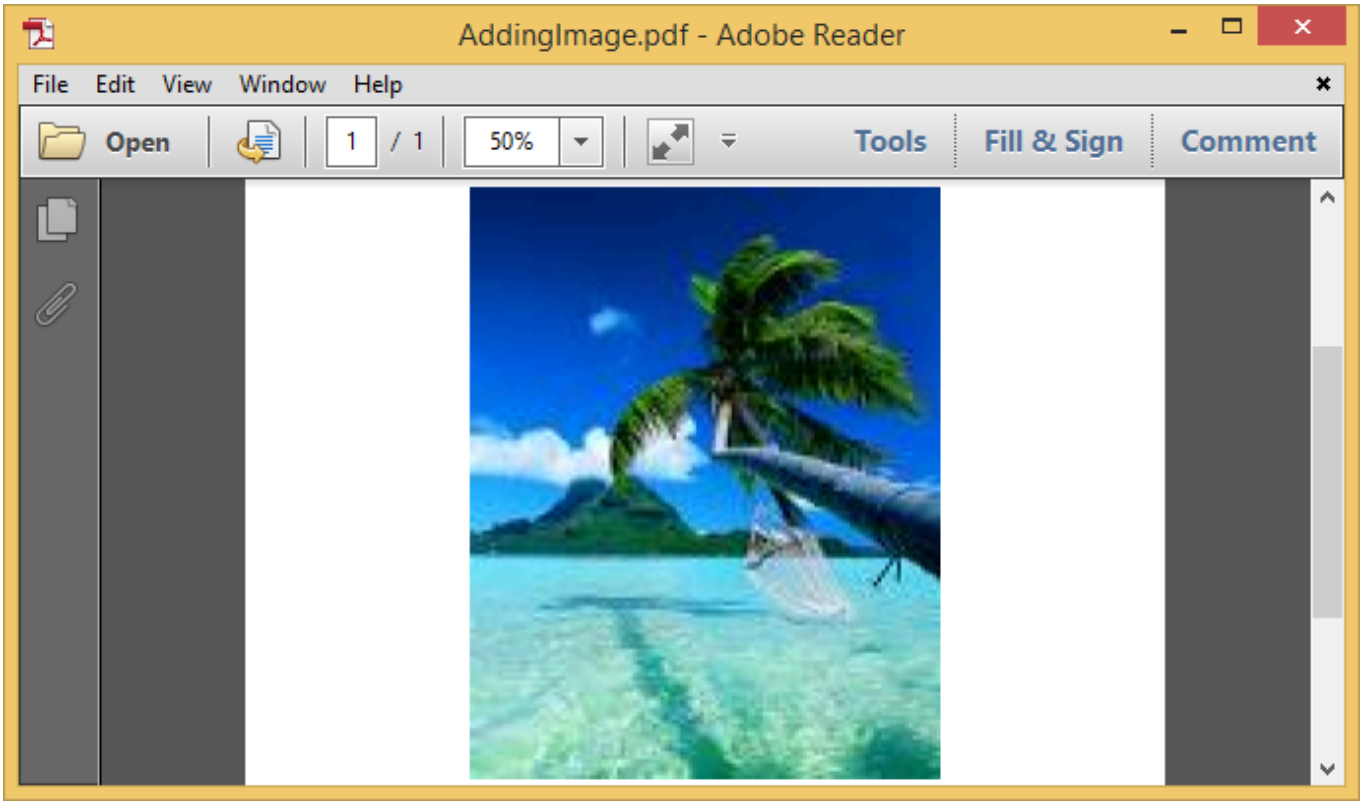
```
Dim rect As Rect = pdf.PageRectangle
rect.Inflate(-150, -150)
Dim fs As Stream = File.Open("..\..\image.jpg", FileMode.Open)
' load image into writeable bitmap
Dim bi As New BitmapImage()
bi.BeginInit()
bi.StreamSource = fs
bi.EndInit()
Dim wb = New WriteableBitmap(bi)
' align image on page preserving aspect ratio
pdf.DrawImage(wb, rect, ContentAlignment.MiddleCenter, Stretch.Uniform)
```

C#

C# copyCode

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

 Stream fs = File.Open(@"..\..\image.jpg", FileMode.Open);

 // load image into writeable bitmap
 BitmapImage bi = new BitmapImage();
 bi.BeginInit();
 bi.StreamSource = fs;
 bi.EndInit();
 var wb = new WriteableBitmap(bi);

 // align image 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 WPF**. 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 WPF** 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 WPF** and the .NET **Graphics** class. The sample declares a **C1PdfDocument** class called 'pdf' and calls method 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 on 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 C1.WPF.Pdf.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 C1.WPF.Pdf.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
```

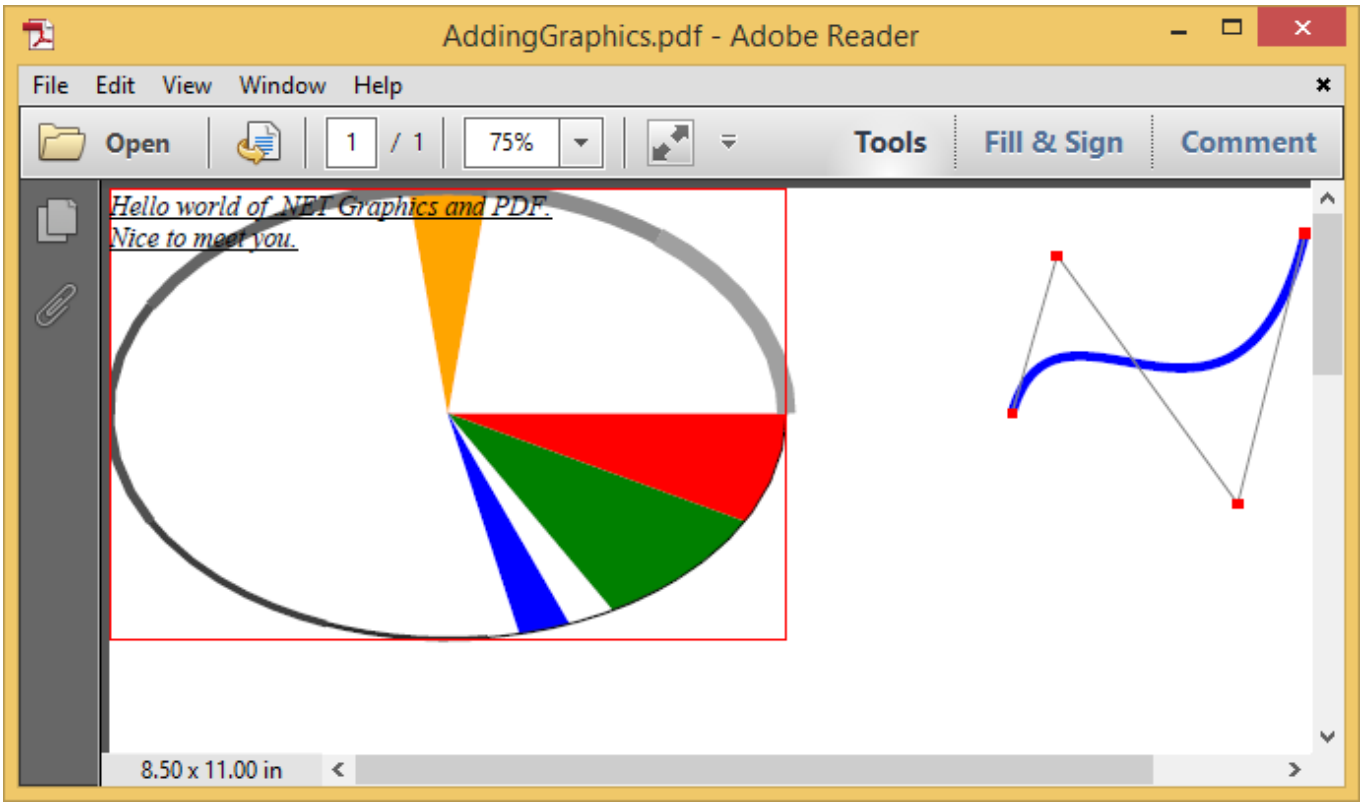
C#

C# copyCode

```
// 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 on 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);
    C1.WPF.Pdf.Pen pen = new C1.WPF.Pdf.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 C1.WPF.Pdf.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 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# copyCode

```
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
```

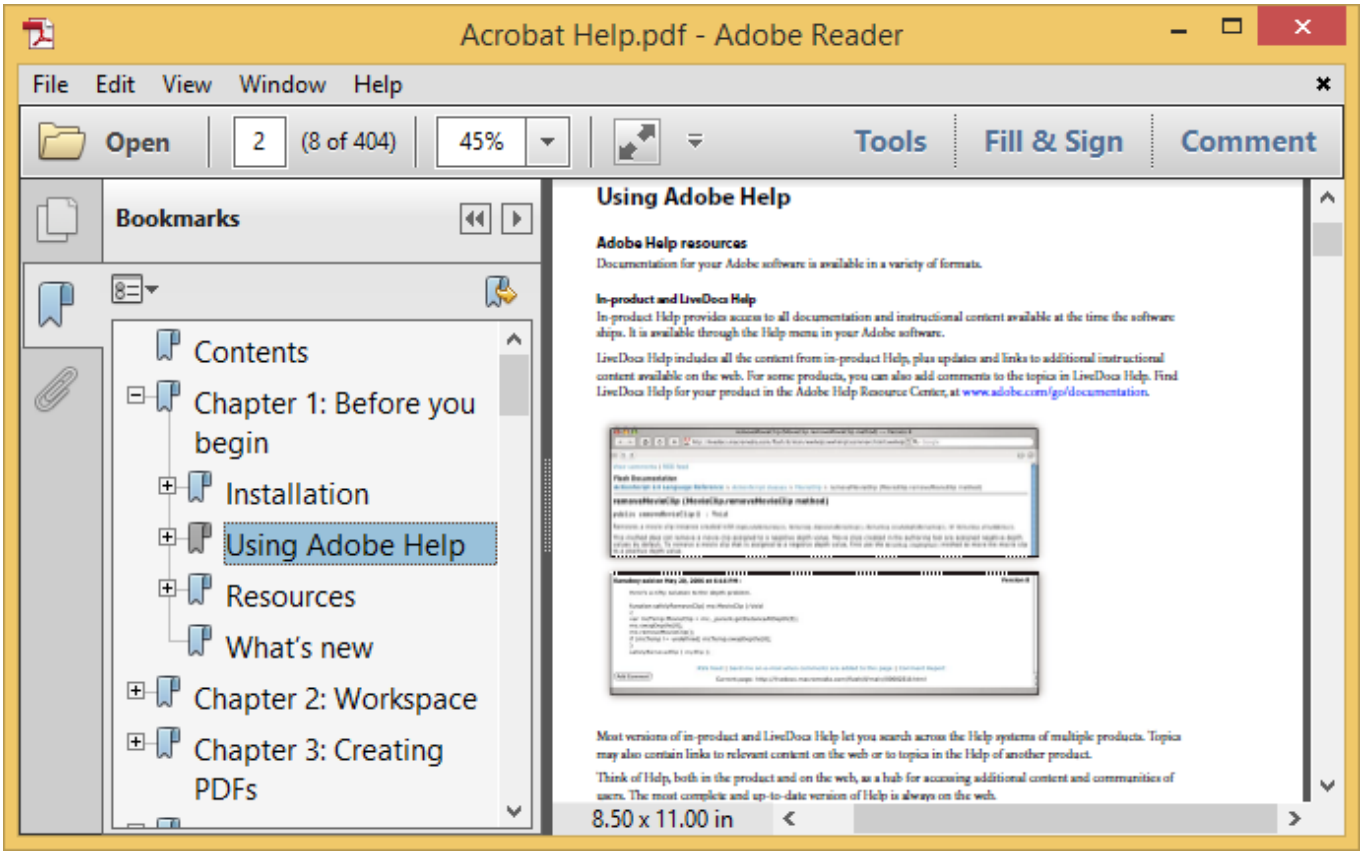
C#

C# copyCode

```
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);
}
```

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 WPF** 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 copyCode

```
   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, Colors.Black, rect);

    // Add headings to outline
    if (outline)
    {
        pdf.DrawLine(System.Windows.Media.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;
}
```

You can also use more overloads for AddBookmark() method - such as to pass the Boolean arguments; to specify whether the children of bookmark are initially visible; or to pass the target name of the document.

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:

To write the code in Visual Basic:

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

To write the code in C#:

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

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 WPF 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 WPF** security features, simply set the passwords before you save the document. For example:

To write the code in 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
```

To write the code in 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;
```

Attaching Files to 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 WPF** 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:

To write the code in 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
```

To write the code in 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;
}
```

Loading Documents

To open an existing PDF file you can use the LoadDocument or LoadDocumentAsync method by passing a stream to the file. To open a file selected by the user, complete the following code:

C#

```
FileOpenPicker openPicker = new FileOpenPicker();

openPicker.FileTypeFilter.Add(".pdf");
StorageFile file = await openPicker.PickSingleFileAsync();
  if (file != null)
    {
       Stream stream = await file.OpenStreamForReadAsync();
       pdfViewer.LoadDocument(stream);
    }
```

Aynchronous Loading

For better performance you can have the C1PdfViewer control load documents in the background asynchronously. Using the .NET await keyword, you can easily call asynchronous methods. To open a file selected by the user asynchronously, complete the following code:

Visual Basic

Visual Basic

```
Dim openPicker As New FileOpenPicker()

openPicker.FileTypeFilter.Add(".pdf")

Dim file As StorageFile = Await openPicker.PickSingleFileAsync()

If file IsNot Nothing Then

       Dim stream As Stream = Await file.OpenStreamForReadAsync()

       Await pdfViewer.LoadDocumentAsync(stream)
End If
```

C#

C#

```
FileOpenPicker openPicker = new FileOpenPicker();

openPicker.FileTypeFilter.Add(".pdf");

StorageFile file = await openPicker.PickSingleFileAsync();

if (file != null)

{

    Stream stream = await file.OpenStreamForReadAsync();

    await pdfViewer.LoadDocumentAsync(stream);

}
```

Loading Encrypted Files

You can open encrypted files using C1Pdf so long as you have the password that the file was encrypted with. To load password protected PDF documents use the LoadDocument or LoadDocumentAsync methods with the password as a parameter.

Visual Basic

Visual Basic

```
Dim password As String = "password"

Await pdfViewer.LoadDocumentAsync(stream, password)
```

C#

C#

```
string password = "password";
await pdfViewer.LoadDocumentAsync(stream, password);
```

Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio and know how to use the **C1PDF** control in general. If you are unfamiliar with the **PDF for WPF and Silverlight** product, please see the Quick Start first.

Each topic in this section provides a solution for specific tasks using the **C1PDF** product.

Each task-based help topic also assumes that you have created a new project.

Drawing Framework Element Objects

You can use the C1PDFDocument.DrawElement method to draw the framework element objects like the following:

To write the code in Visual Basic:

```
Private Sub CreateVisualTree(pdf As C1PdfDocument, targetElement As FrameworkElement)

' set up to render
Dim font = New Font("Courier", 14)

' go render
Dim firstPage = True
For Each stretch__1 As Stretch In New Stretch() {Stretch.Fill, Stretch.None, Stretch.Uniform, Stretch.UniformToFill}

 ' add pagebreak
 If Not firstPage Then

  pdf.NewPage()
End If
firstPage = False

' set up to render
Dim alignment = ContentAlignment.TopLeft
Dim rc = PdfUtils.Inflate(pdf.PageRectangle, -72, -72)

' render element
pdf.DrawString("Element as VisualTree, Stretch: " + stretch__1.ToString(), font, Colors.Black, rc)
rc = PdfUtils.Inflate(rc, -20, -20)
```

```
 pdf.DrawElement(targetElement, rc, alignment, stretch__1)
Next

End Sub
```

To write the code in C#:

```
 void CreateVisualTree(C1PdfDocument pdf, FrameworkElement targetElement)
        {
            // set up to render
            var font = new Font("Courier", 14);
            // go render
            var firstPage = true;
             foreach (Stretch stretch in new Stretch[] { Stretch.Fill, Stretch.None,
Stretch.Uniform, Stretch.UniformToFill })
            {
                // add pagebreak
                if (!firstPage)
                {
                    pdf.NewPage();
                }
                firstPage = false;
                // set up to render
                var alignment = ContentAlignment.TopLeft;
                var rc = PdfUtils.Inflate(pdf.PageRectangle, -72, -72);
                // render element
                 pdf.DrawString("Element as VisualTree, Stretch: " + stretch.ToString(), font,
Colors.Black, rc);
                 rc = PdfUtils.Inflate(rc, -20, -20);
                 pdf.DrawElement(targetElement, rc, alignment, stretch);
            }
        }
```

Loading Documents from the Web

To load a file from the Web you must first download it to your application using an asynchronous request object such as HttpClient. Then you simply pass the resulting stream to the LoadDocument method or LoadDocumentAsync method. The following code snippet example uses an HTTP request:

Visual Basic

Visual Basic

```
Private Sub LoadProtectedDocument(stream As System.IO.MemoryStream, password As
String)
        Try
            stream.Position = 0
            _viewer.LoadDocument(stream, password)
        Catch x As Exception
            'if (x.Message.IndexOf("password") > -1)
            '{
            Dim msg = "This file seems to be password-protected." & vbCr & vbLf &
```

```
"Please provide the password and try again."
            C1.Silverlight.C1PromptBox.Show(msg, "Enter Password", Function(text,
result)
                  If result = MessageBoxResult.OK Then

                    ' try again using the password provided by the user
                    LoadProtectedDocument(stream, text)
                  End If
                '}
                   'else
                '{
                   'throw;
                '}
            End Function)
        End Try
    End Sub
```

C#

C#

```
void LoadProtectedDocument(System.IO.MemoryStream stream, string password)
        {
            try
            {
                stream.Position = 0;
                _viewer.LoadDocument(stream, password);
            }
            catch (Exception x)
            {
                //if (x.Message.IndexOf("password") > -1)
                //{
                    var msg = "This file seems to be password-protected.\r\nPlease
provide the password and try again.";
                    C1.Silverlight.C1PromptBox.Show(msg, "Enter Password", (text,
result) =>
                        {
                            if (result == MessageBoxResult.OK)
                            {
                                // try again using the password provided by the user
                                LoadProtectedDocument(stream, text);
                            }
                        });
                //}
                //else
                //{
                //    throw;
                //}
            }
```

Opening Potentially Protected Files

When giving the end-user the ability to open a PDF file, sometimes you can't predict whether or not the file will be password protected or not. The following sample method demonstrates how to perform this check and open the document accordingly:

Visual Basic

Visual Basic

```
Private Sub _btnOpen_Click(sender As Object, e As RoutedEventArgs)

        Dim dlg = New OpenFileDialog()

        dlg.Filter = "Pdf files (*.pdf)|*.pdf"

        If dlg.ShowDialog().Value Then

            Dim ms = New System.IO.MemoryStream()

            Using stream = dlg.File.OpenRead()

              stream.CopyTo(ms)

            End Using

            LoadProtectedDocument(ms, Nothing)

            End If

    End Sub
```

C#

C#

```
void _btnOpen_Click(object sender, RoutedEventArgs e)

        {
            var dlg = new OpenFileDialog();

            dlg.Filter = "Pdf files (*.pdf)|*.pdf";

            if (dlg.ShowDialog().Value)

            {
                var ms = new System.IO.MemoryStream();
```

```
                using (var stream = dlg.File.OpenRead())
                {

                    stream.CopyTo(ms);
                }

                LoadProtectedDocument(ms, null);

            }
        }
```

When giving the end-user the ability to open a PDF file, sometimes you can't predict whether or not the file will be password protected or not. The following sample method demonstrates how to perform this check and open the document accordingly:

Creating and Writing Encrypted Files

To create and write an encrypted Pdf file using owner and user passwords you can first use the LoadDocument method with password parameter and then create and write the encrypted file. For more information on how to use the LoadDocument method seeLoading Documents from the Web.

Visual Basic

Visual Basic

```
 Using stream = dlg.OpenFile()
    Dim pdf = New C1PdfDocument()
       pdf.Security.OwnerPassword = "ownerPassword"
       pdf.Security.UserPassword = "userPassword"
    ' added others content of the pdf file
       pdf.Save(stream)
End Using
```

C#

C#

```
using (var stream = dlg.OpenFile())
                {
                    var pdf = new C1PdfDocument();
                    pdf.Security.OwnerPassword = "ownerPassword";
                    pdf.Security.UserPassword = "userPassword";

                     …       // added others content of the pdf file

                    pdf.Save(stream);
        }
```