

Apitron PDF Kit in Action

v.1.0.4

1. Introduction

This book is intended to give you deep knowledge and understanding of the development techniques needed to properly use the Apitron PDF Kit for .NET component for PDF manipulation tasks. It includes overview of PDF generation, editing and signing as well as various content extraction operations you may need to complete during the development of your application.

The component itself has rich API and is suitable for cross-platform development being .NET, MONO and Xamarin compatible. It was widely adopted and is being used by many desktop, server and mobile applications on the market. Novice and experienced developers could both benefit from reading this book, because even if some concepts of PDF might be known and parts of the API might look familiar there are totally new things that needs to be mastered, e.g. Flow PDF generation API.

The book is based on version 1.0.4 of the Apitron PDF Kit for .NET and all samples provided here were created and tested using this version.

1.1 PDF, what is this?

ISO 32000 specifies a digital form for representing documents called the Portable Document Format or usually referred to as PDF. PDF was developed and specified by Adobe Systems Incorporated beginning in 1993 and continuing until 2007 when this ISO standard was prepared. The Adobe Systems version PDF 1.7 is the basis for this ISO 32000 edition. Its specification can be found by this [link](#).

Previous paragraph is a quote from the spec and it gives you brief understanding of format history. But what PDF actually is and what you should know about it if you don't want to study the spec completely and implement your own component (756 pages + supplementary material e.g. font formats)?

Not considering the binary part, PDF is a vector-based format for representing documents, where each page may contain drawing operations in a form of commands e.g. draw line; fill polygon, change stroking color, draw image etc. and text drawing operators. Several well-known font formats are being supported and, since all drawing operations in PDF are vector-based (except for raster images), PDF pages usually scale well, being resolution-independent. We will consider PDF manipulations from the high level of abstraction diving into binary details only if it's absolutely necessary.

1.2. Overview of Apitron PDF Kit API

The library can be divided into two large parts, demonstrating different approaches to PDF creation and manipulation. Being independent from each other they share several common objects and can be combined together to achieve a complex effect.

These parts are:

- **FixedLayout API**, built around the `FixedDocument` class and provides “classic” approach for PDF manipulation. It has 100% mapping to spec-defined entities and can be used for every task where you need the precise control over the things which are to be put on PDF page and the way it should be done. You can get access to all drawing commands and content of the page; everything in PDF document can be manipulated using this API. See the section [3. Fixed layout API](#) for the details.
- **FlowLayout API**, built around the `FlowDocument` class and provides innovative and flexible approach to generate PDF files using styles-based, html- and css- like document model. You could use it when you need automatic layout rules applied to the content blocks. You don't set the explicit position of content elements on page, but rather control they layout behavior using built-in styling mechanism similar to CSS used with HTML. Content for the document has to be built using predefined content elements, e.g. `Textblock`, `Image`, `Grid` etc. while flow layout engine performs automatic pagination or splitting if necessary. See the section [4. Flow layout API](#) for the details.

1.3 Supported platforms

Apitron PDF Kit is compatible with any .NET framework version starting from 2.0, Mono and Xamarin. This component can be used to create applications for Windows (Windows Forms, WPF, Web, Console, Services), Windows 8/Windows 8.1/Windows Store, Windows Phone Store 8.0/8.1, Xamarin.iOS, Xamarin.Android or any OS for which a .NET/Mono implementation exists.

2. Document level API

Both **FixedDocument** and **FlowDocument** contain properties which control the document viewing behavior in conforming PDF reader. Also it's possible to change document internal structure: swap pages, append new pages, add fields etc. and it will be covered by articles in this section.

The code that demonstrates most of the techniques used here can be found in Samples folder from the download package available on our website.

2.1 Enumerate document pages and extract text

Existing document pages can be accessed by using `FixedDocument.Pages` collection; it contains a set of `Page` objects with actual content.

Consider the following code:

```
public void EnumerateAllPages()
{
    // open existing PDF file
    using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read))
    {
        FixedDocument document = new FixedDocument(inputStream);

        // enumerate through all pages and extract text from it
        foreach (Page page in document.Pages)
        {
            Console.WriteLine(string.Format(page.ExtractText()));
        }
    }
}
```

This code just opens some PDF file and prints the text from each page by enumerating document page collection.

2.2 Add new page to PDF document

Let's open the document and add a new empty page to it sized to *Letter*:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // add new page sized to Letter paper format into the document
    // and save modified document to the output stream
    document.Pages.Add(new Page(Boundaries.Letter));
    document.Save(outputStream);
}
```

The code is quite self-explaining; you can also change the desired page size by passing one of the predefined formats or creating a custom-sized page using an overload.

2.3 Remove page from the PDF document

Let's open the PDF document and delete its first page:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // remove the first page and save
    document.Pages.RemoveAt(0);
    document.Save(outputStream);
}
```

You may remove the desired page by passing its index to the `RemoveAt()` method or by simply passing the `Page` object itself to the `Remove()` method defined in page collection.

2.4 Move pages within the PDF document

Let's open the PDF document and move its third page to the 11th position in the original document:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // get the 3rd page
    Page page = document.Pages[2];

    // remove 3rd page from the document
    document.Pages.Remove(page);

    // insert 3rd page as 11th page
    document.Pages.Insert(10, page);

    document.Save(outputStream);
}
```

As you see this task can be completed by removing the page from document first and then inserting it at the desired position.

2.5 Copy pages from one PDF document to another

It's possible to copy pages between documents, see the code below:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // create new document that will get the copied page
    FixedDocument outDocument = new FixedDocument();

    // export page from source document and insert to the destination
    outDocument.Pages.Insert(0, Page.Export(outDocument, document.Pages[0]));

    outDocument.Save(outputStream);
}
```

In order to successfully copy the page from one document to another, the page should be exported first. It's being done by calling `Page.Export()` static method accepting the target document and the page to be copied.

2.6. Swap pages in PDF document

It's possible to swap pages using add and remove operations demonstrated in preceding articles combined together:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // swap pages
    Page page = document.Pages[0];
    Page page2 = document.Pages[4];

    document.Pages.Remove(page);
    document.Pages.Remove(page2);

    document.Pages.Insert(4, page);
    document.Pages.Insert(0, page2);

    // save to output stream
    document.Save(outputStream);
}
```

We get the 1st and 4th pages and then swap them by removing from their original position and adding to the right place.

2.7. Setting PDF page parameters

Page has own properties which can affect its behavior and also its content, e.g. initial transformation that changes the content placement. Using the code below we change the size of the first page, initial rotation for the second page and content transformation matrix for the third page.

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // Resize first page
    Page page = document.Pages[0];
    page.Resize(new PageBoundary(Boundaries.A3));
    page.Transform(1, 0, 0, 1, 100, 100);

    // Rotate second page
    Page page2 = document.Pages[1];
    page2.Rotate = PageRotate.Rotate90;

    // Transform content on third page
    Page page3 = document.Pages[2];
    page3.Transform(0.5, 0, 0, 0.5, 0, 0);
    page3.Transform(1, 0, 0, 1, 100, 100);

    // save to output stream
    document.Save(outputStream);
}
```

2.8 Setting initial viewer settings and page layout

PDF document may have settings which describe the preferred viewer state and page layout when the document is being opened. A conforming reader (Adobe PDF Reader for example) may respect these settings and react accordingly.

Viewer settings and initial page showing mode can be controlled by setting various flags using `FixedDocument.ViewerPreferences` property. E.g. `HideToolbar` setting controls whether reader toolbars should be shown when the document is active.

Desired page layout can be set using `FixedDocument.PageLayout` property, so you'll be able to choose which layout fits better for your document and ask the reader to apply it by default.

The code below shows how to use these settings:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // change viewer preferences
    document.PageLayout = PageLayout.TwoPageLeft;
    document.PageMode = PageMode.UseThumbs;
    document.ViewerPreferences.HideMenubar = true;
    document.ViewerPreferences.HideToolbar = true;

    // save to output stream
    document.Save(outputStream);
}
```

2.9 Working with fields

PDF defines the term field as related to the interactive forms and annotations and it often acts as backing store for objects like textbox, checkbox, button etc. Each field has its entry in document catalog `AcroForm` dictionary which is defined in our case as `AcroForm` property of the `FixedDocument` class. Section 12.7 “Interactive Forms” of the PDF specification has a complete description of fields and interactive forms.

While the subject remains quite complex, it’s easier to think about document field as of some property or attribute attached to the document. Its content can be used for further processing by automation tools as a part of some workflow or for any other purpose where such attribute is meaningful.

Fields can have visual representations placed on PDF page which are called widget annotations, but it’s not necessary. These widgets can be attached to fields, and their appearance can be affected by fields values. It’s also possible to attach multiple widgets to a single field therefore affecting them all with a single value.

The code below shows how to work with fields, it enumerates existing fields and adds new text field to the document after that; widgets will be discussed later in chapter [3.7 Interactive forms](#).

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // print names of the existing fields
    foreach (Field field in document.AcroForm.Fields)
    {
        Console.WriteLine(field.FieldName);
    }

    // add text field
    TextField textField = new TextField("tx_field", "Text field", "This is text field");
    document.AcroForm.Fields.Add(textField);

    // save to output stream
    document.Save(outputStream);
}
```

2.10 How to manage attachments in PDF document

It can be surprising but PDF document can have multiple attachments and these files can be linked to from documents content, therefore it makes PDF document a self-contained unit that can be stored or transferred as a single object. Section 7.11.4 “*Embedded File Streams*” of the PDF specification describes this functionality in details.

If you want to attach a file to PDF document you may use `FixedDocument.Attachments` property, it returns an `EmbeddedFileCollection` that belongs to the given document.

Consider the code below:

```
// open existing PDF file
using (FileStream inputStream = new FileStream("testfile.pdf", FileMode.Open, FileAccess.Read),
    outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument(inputStream);

    // get the list of attachments
    Console.WriteLine("The count of attachments is : " + document.Attachments.Count);
    Console.WriteLine("List of attachments : ");
    foreach (var attachment in document.Attachments)
    {
        Console.WriteLine(attachment.Key);
    }

    // add attachment
    document.Attachments.Add("Attachment # 1", new EmbeddedFile(@"attachment.pdf",
        "type/pdf"));

    // save to output stream
    document.Save(outputStream);
}
```

We enumerate the list of existing attachments first and then add another PDF file as new attachment. Attachments can be removed from the document by using `FixedDocument.Attachments.RemoveAttachment()` member function.

3. Fixed layout API

Fixed layout API is a “classic” API that means that it’s designed using the most common and straightforward way, being so close to the specification as possible that each PDF entity has a mapping to a Fixed layout object.

This API was built around the `FixedDocument` class which in its turn represents the PDF document model. Anything that can be done with PDF one can do using the Fixed layout API. It truly unlocks all opportunities of PDF processing to developers.

3.1 Typical PDF generation pattern

The code below produces an empty PDF file:

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument();

    // [fill doc here]

    // save to output stream
    document.Save(outputStream);
}
```

Not much can be said about that as it is pretty self-explaining. Just create a file, put some content into it and save.

3.2 Working with Text

Apitron PDF Kit for .NET implements all text features described in PDF specification. It's important to note that it also automatically handles bi-directional text entries often used in Arabic and Asian cultures.

Any text in PDF has the following key attributes:

- font, can be one of the standard fonts, externally linked or embedded
- text positioning and showing operators, describing the text transformation and state
- stroking and non-stroking colors

Subtopics below will guide you through all aspects related to these properties and will show how to use them practically.

3.2.1 Font types in PDF

Several font types are defined in PDF spec and described in terms of font file format, encodings, character maps and other usual font characteristics. But we will discuss fonts from the other point of view, because in most cases you won't be thinking whether your font is stored in TrueType, OpenType, CFF or other font file format. The most important things are however, will it be accessible to the viewer of prepared document and how it'll affect the resulting PDF file.

So far there are three font types you have to deal with:

Standard fonts - fonts defined by PDF specification as to be supported by any conforming PDF reader and therefore documents created using such fonts should be always viewable. These fonts don't require any font data to be written into resulting PDF file and don't affect its size.

These fonts are: *Times-Roman, Helvetica, Courier, Symbol, Times-Bold, Helvetica-Bold, Courier-Bold, ZapfDingbats, Times-Italic, Helvetica-Oblique, Courier-Oblique, Times-BoldItalic, Helvetica-BoldOblique, Courier-BoldOblique.*

Apitron PDF Kit defines a `StandardFonts` enum that maps to this set.

See section 9.6.2.2 "Standard Type 1 Fonts (Standard 14 Fonts)" of the PDF specification for the details.

Sample code below, shows how one could use a standard font for a text object:

```
// create text object based on standard Type1 font
TextObject text = new TextObject(StandardFonts.TimesBold, 12);
```

External fonts – fonts assumed to be installed to the default system fonts location, e.g. one of the fonts from "C:\Windows\Fonts". They could be used when document is being viewed on MS Windows. These fonts also don't affect document size because their data is not included in the resulting file.

Embedded fonts – the name speaks for itself, so these fonts are being included into the PDF file making it platform-independent and viewable on all systems where a conforming reader exists. They also affect resulting file size. It's possible to embed only the data needed to display particular text contained in PDF file and it's what Apitron PDF Kit does when it has to embed font data. This technique is called *font-subsetting* and only glyphs used along with their data are being embedded into the resulting PDF file in a form of a reduced-size font file.

3.2.2 Text operators

A PDF text object consists of operators that may show text strings, move text position, set text state and certain other parameters. This text object represents a **sequence** of text commands and therefore results depend on their **order**.

Let's look how it works in Apitron PDF kit:

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new document
    FixedDocument document = new FixedDocument();

    // add blank first page
    document.Pages.Add(new Page(Boundaries.A4));

    // create text object and append text to it
    TextObject textObject = new TextObject(StandardFonts.Helvetica,12);

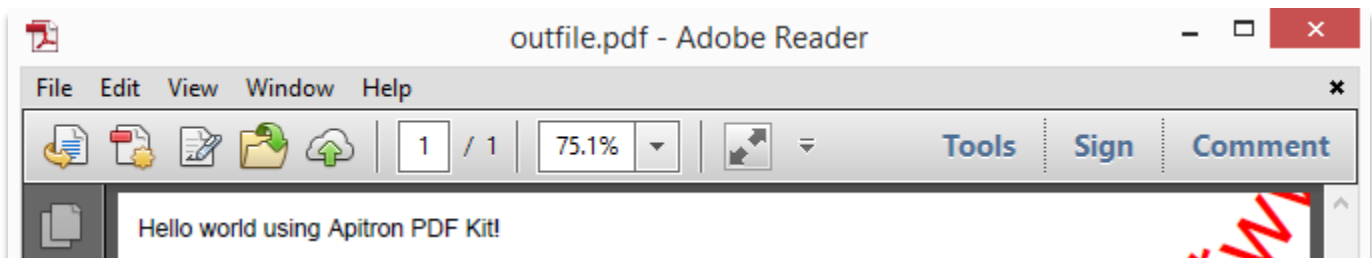
    // apply identity matrix, that doesn't change default appearance
    textObject.SetTextMatrix(1,0,0,1,0,0);
    textObject.AppendText("Hello world using Apitron PDF Kit!");

    // set current transformation matrix so text will be added to the top of the page,
    // PDF coordinate system has Y-axis directed from bottom to top.
    document.Pages[0].Content.Translate(10, 820);

    // add text object to page content, it will automatically create text showing operators
    document.Pages[0].Content.AppendText(textObject);

    // save to output stream
    document.Save(outputStream);
}
```

The result will look like this:



Pic. 1 Usage of TextObject

We created an empty document, added new page to it and put an instance of `TextObject` class inside its content. For this text object we also created a text matrix and indicated that we'd like to use one of the standard fonts. This simplistic API leaves a lot of things behind the scene e.g. creation of necessary operators which would be a boring task indeed, providing you with a clean and straightforward way to get job done. Other text options can be set using text object instance, e.g. leading, char and word spacing, text rise, rendering mode etc. See section 9.4 “*Text Objects*” of the specification for a complete list.

Other thing to notice is how we positioned the text on page. It was achieved by setting current transformation object for the page content to which our textobject was subsequently added. This transformation set the initial position of the text object coordinate space.

```
// set current transformation matrix so text will be added to the top of the page,  
// PDF coordinate system has Y-axis directed from bottom to top.  
document.Pages[0].Content.Translate(10, 820);
```

So we've added 10pt offset to the X-coordinate and 820pt to the Y coordinate of the page initial transformation, moving our objects from left to right and from bottom to top. This way we got our text object placed on top of the page. If there was another text object added after the first one, it would have to have an additional transformation applied in order to not overlap.

3.2.3 Add text to PDF file

In previous paragraph we showed how to add simple text into PDF file using standard font. Here we'll show to add text using non-standard font and set its color and other properties.

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new document
    FixedDocument document = new FixedDocument();

    // add blank first page
    document.Pages.Add(new Page(Boundaries.A4));

    // create text object and append text to it
    TextObject textObject = new TextObject("Arial", 14);

    // apply identity matrix, that doesn't change default appearance
    textObject.SetTextMatrix(1, 0, 0, 1, 0, 0);
    textObject.AppendText("Hello world using Apitron PDF Kit!");

    textObject.SetFont("ArialItalic", 14);
    // apply vertical scaling and offset
    textObject.SetTextMatrix(1, 0, 0, 2.5, 0, -40);

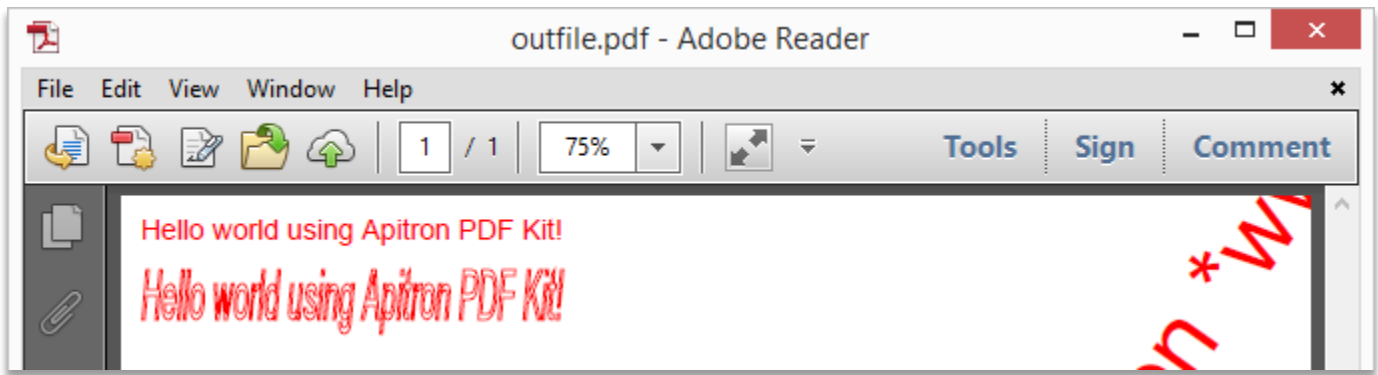
    // set mode to stroke only
    textObject.SetTextRenderingMode(RenderingMode.StrokeText);
    textObject.AppendText("Hello world using Apitron PDF Kit!");

    // set current stroking and non-stroking color
    document.Pages[0].Content.SetDeviceStrokingColor(new double[]{1,0,0});
    document.Pages[0].Content.SetDeviceNonStrokingColor(new double[]{1,0,0});

    // set current transformation
    document.Pages[0].Content.Translate(10, 820);
    // add text object to page content, it will automatically create text showing operators
    document.Pages[0].Content.AppendText(textObject);

    // save to output stream
    document.Save(outputStream);
}
```

This code produces the following results:



Pic. 2 Add text to PDF file

These two lines of text were added using single text object. As it's been said in [3.2.2 Text operators](#), text object is a **sequence** of text commands; therefore it's possible to create several textual sequences with different appearance contained in one text object.

You may also note that we changed the text color, using this code:

```
// set current stroking and non-stroking color
document.Pages[0].Content.SetDeviceStrokingColor(new double[]{1,0,0});
document.Pages[0].Content.SetDeviceNonStrokingColor(new double[]{1,0,0});
```

It set both stroking and non-stroking colors to **red** by specifying its RGB value in so-called device color space. It was automatically detected from number of arguments and was set to DeviceRGB (see section 8.6.4.3 "DeviceRGB Colour Space" of the specification). All **subsequent** drawing commands added after these calls and involving filling or stroking would have this color applied to them. In our example we added text after these calls so it became red.

3.2.4 Right to left and bidirectional text

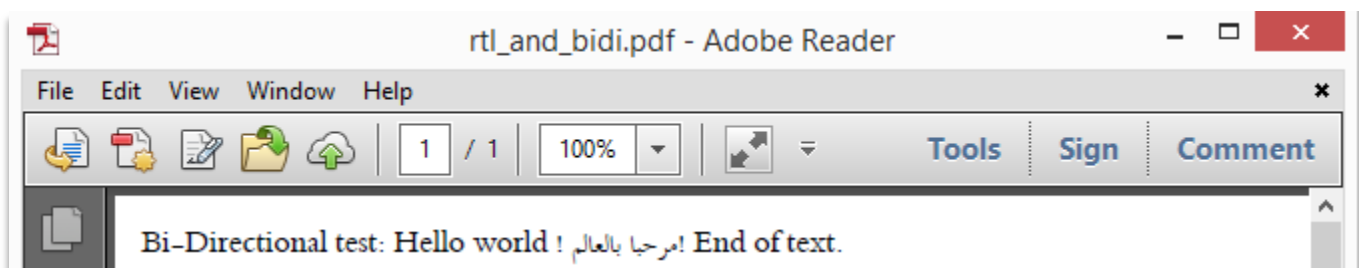
Apitron PDF Kit provides automatic support for adding right to left and bidirectional text, the sample below demonstrates how it could be added.

```
// create output file
using (Stream outputStream = File.Create("rtl_and_bidi.pdf"))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument();
    fixedDocument.Pages.Add(new Page());

    ClippedContent content = fixedDocument.Pages[0].Content;
    content.Translate(10, 820);

    // add text using regular textobject, using system font
    TextObject textObject = new TextObject("Traditional Arabic", 12);
    textObject.AppendText("Bi-Directional test: Hello world ! عالم مرحبا ! End of text.");
    content.AppendText(textObject);
    // save document
    fixedDocument.Save(outputStream);
}
```

Image below demonstrates how the resulting file looks like:



Pic. 3 Right to left and bi-directional text

It only requires you to set the proper font containing glyphs for the characters you used, and that's it. It could be either the system one or one set using an explicit font path.

3.3. Graphics

PDF graphics system is built around a few concepts:

Coordinate space, which defines the canvas on which all painting occurs. It determines the position, orientation, and scale of the text, graphics, and images that appear on a page.

Paths and positions shall be defined in terms of pairs of coordinates on the Cartesian plane. A coordinate pair is a pair of real numbers x and y that locate a point horizontally and vertically within a two-dimensional coordinate space. A coordinate space is determined by the following properties with respect to the current page: the location of the origin, the orientation of the x and y axes, the lengths of the units along each axis

Graphics state, which defines the current state of the drawing system: current stroking and non-stroking colors, masks, transformations, clipping, alpha etc. See section 8.4 “Graphics State” of the PDF specification.

Drawing commands, which use current coordinate space and graphics state and may affect them if designed to do so. They often use *graphics paths* as an input, which in turn consist of atomic operations for combining lines and curves into a single figure. `ClippedContent` class instances are being used as command containers and can be nested into each other providing a way to create combined clippings and other visual effects. `Page` class is also inherited from `ClippedContent` thus providing the same API.

Next few paragraphs will guide you through all these concepts and show how to use them in real-life application.

3.3.1 Basics

3.3.1.1 Colorspaces and colors

Colors in PDF are defined using colorspaces and color values mapped to these spaces. All available colorspaces are described in section 8.6 “Colour Spaces” of the PDF specification. The colorspace in short, is a way to represent a particular color using specific “coordinates” in this space. As an example, for well-known sRGB colorspace these values are R, G and B components.

In order to stroke or fill using a particular color you’ll have to select the corresponding colorspace and set this color as current stroking or non-stroking color. Consider the sample code below:

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    Page page = new Page();

    // register RGB and CMYK color spaces.
    // it's also possible to use Lab, Gray, Indexed, ICC based color spaces.
    document.ResourceManager.RegisterResource(new RgbColorSpace("CS_RGB"));
    document.ResourceManager.RegisterResource(new CmykColorSpace("CS_CMYK"));

    // use RGB color space and set non stroking color
    page.Content.SetNonStrokingColorSpace("CS_RGB");
    page.Content.SetNonStrokingColor(0.33, 0.66, 0.33);
    // use RGB color space and set stroking color
    page.Content.SetStrokingColorSpace("CS_RGB");
    page.Content.SetStrokingColor(0.77, 0.2, 0.33);

    // create path and fill it using the color created above
    Path path = new Path();
    path.AppendRectangle(10, 720, 300, 80);
    page.Content.FillAndStrokePath(path);

    // use CMYK color space, and set non-stroking color
    page.Content.SetNonStrokingColorSpace("CS_CMYK");
    page.Content.SetNonStrokingColor(0.1, 0.3, 0.2, 0);

    page.Content.ModifyCurrentTransformationMatrix(1, 0, 0, -1, 150, 850);
}
```

...code continues on next page

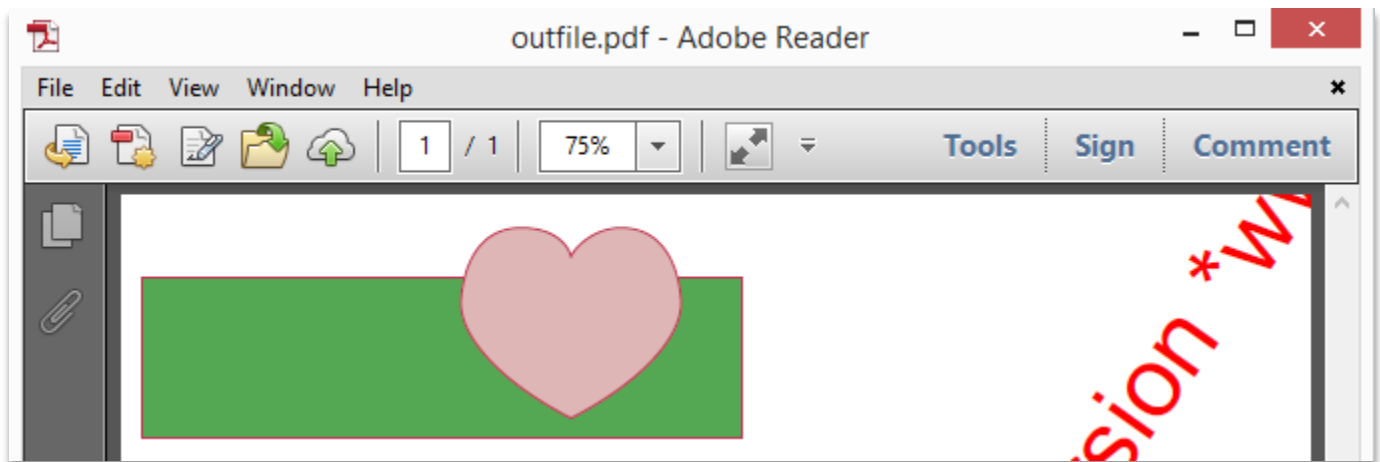
```

// create path, fill it using CMYK color and stroke it using RGB color set above
Path path2 = new Path(70, 80);
path2.MoveTo(75, 40);
path2.AppendCubicBezier(75, 37, 70, 25, 50, 25);
path2.AppendCubicBezier(20, 25, 20, 62.5, 20, 62.5);
path2.AppendCubicBezier(20, 80, 40, 102, 75, 120);
path2.AppendCubicBezier(110, 102, 130, 80, 130, 62.5);
path2.AppendCubicBezier(130, 62.5, 130, 25, 100, 25);
path2.AppendCubicBezier(85, 25, 75, 37, 75, 40);
path2.ClosePath();
page.Content.FillAndStrokePath(path2);

// add created page and save document
document.Pages.Add(page);
document.Save(outputStream);
}

```

This code shows how to set current colors using various colorspaces for filling and stroking of graphics paths. The image below demonstrates the result:



Pic. 4 Colorspaces and colors

We used simple device colorspaces in this example, but it's possible to load colorspaces described by ICC profiles provided with some reproduction devices or use complex colorspaces e.g. Lab or Indexed.

3.3.1.2 Drawing a line on PDF page

Here goes the code for drawing a line. We create a `Path` object and add line to it by calling the self-explaining method `Path.AppendLine`. After that we set line width and color and stroke this path object by calling `ClippedContent.StrokePath` method.

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new document
    FixedDocument document = new FixedDocument();
    // create new page
    Page page = new Page(Boundaries.A4);

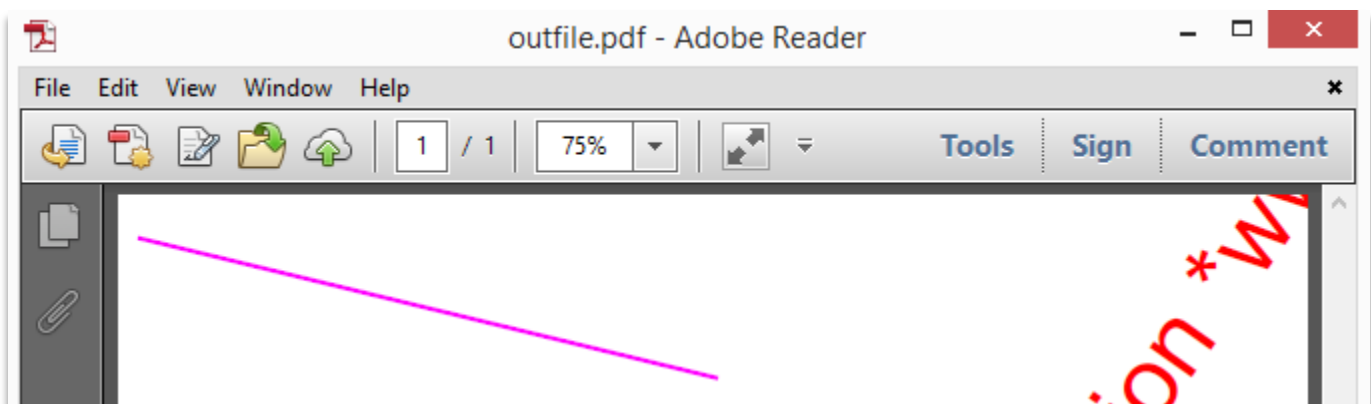
    // create new path representing a line
    Path path = new Path(10, 820);
    path.AppendLine(300, 750);

    // set current non-stroking color
    page.Content.SetDeviceStrokingColor(new double[] { 1, 0, 1 });
    // set line width
    page.Content.SetLineWidth(2.0);
    // stroke path
    page.Content.StrokePath(path);

    // add page to the document
    document.Pages.Add(page);

    // save to output stream
    document.Save(outputStream);
}
```

The results produced by this code are below:



Pic. 5 Drawing a line

3.3.1.3 Drawing a Bézier curve on PDF page

Drawing a curve has lots in common with drawing a line, except we have to define control points for it as described in section 8.5.2.2 “Cubic Bézier Curves” of the specification. The code is almost the same as for drawing a line except that `AppendCubicBezier` method was used instead of `AppendLine` to define a Bézier curve.

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new document and add empty page
    FixedDocument document = new FixedDocument();
    Page page = new Page(Boundaries.A4);

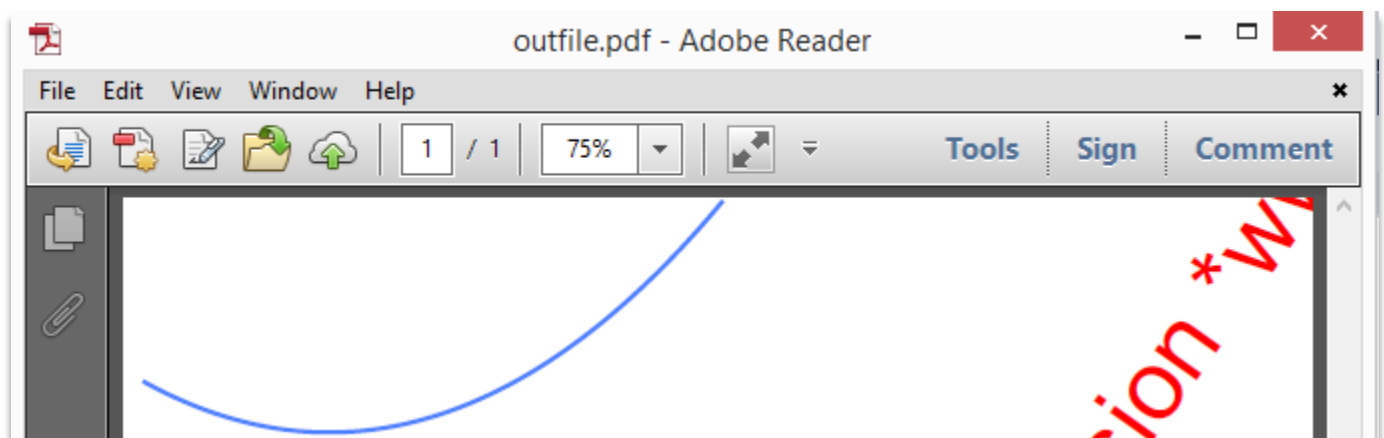
    // create new path representing a line
    Path path = new Path(10, 750);
    path.AppendCubicBezier(100,700,200,720,300,840);
    // set current non-stroking color
    page.Content.SetDeviceStrokingColor(new double[] {0.3, 0.5, 1 });

    // set line width
    page.Content.SetLineWidth(2.0);
    // stroke path
    page.Content.StrokePath(path);

    document.Pages.Add(page);

    // save to output stream
    document.Save(outputStream);
}
```

Here are the results:



Pic. 6 Drawing a Bézier curve

3.3.1.4 Drawing a circle on PDF page using Bézier curves

Drawing a line or a curve is not so entertaining, so we are going to draw a complex but very common figure, a circle. In order to draw it we will combine four Bézier curves to one path and stroke it.

The code for drawing a circle is as follows:

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // radius of the circle
    double radius = 100;
    // circle constant
    double r_c = 0.5522847498 * radius;

    // create new document
    FixedDocument document = new FixedDocument();

    // create new page
    Page page = new Page(Boundaries.A4);

    // create new path representing a circle
    Path path = new Path(0, radius);
    path.AppendCubicBezier(r_c, radius, radius, r_c, radius, 0);
    path.AppendCubicBezier(radius, -r_c, r_c, -radius, 0, -radius);
    path.AppendCubicBezier(-r_c, -radius, -radius, -r_c, -radius, 0);
    path.AppendCubicBezier(-radius, r_c, -r_c, radius, 0, radius);

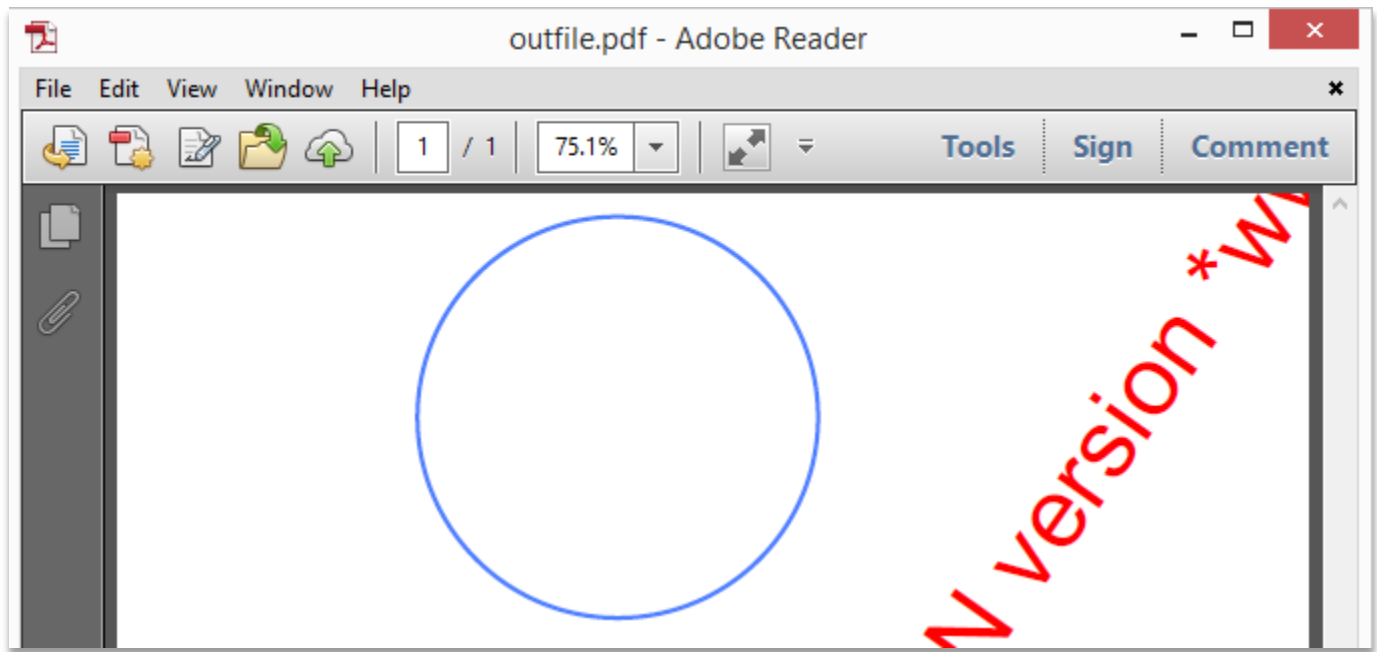
    // set current non-stroking color
    page.Content.SetDeviceStrokingColor(new double[] { 0.3, 0.5, 1 });
    // set line width
    page.Content.SetLineWidth(2.0);

    page.Content.ModifyCurrentTransformationMatrix(1,0,0,1,250,730);
    // stroke path
    page.Content.StrokePath(path);

    // add page to the document
    document.Pages.Add(page);

    // save to output stream
    document.Save(outputStream);
}
```

The resulting circle it creates is shown below:



Pic. 7 Drawing a circle

This sample demonstrated how to create complex paths and combine several drawing commands together. Next sample will show how to draw filled shapes and apply clipping to achieve various effects.

3.3.1.5 Drawing a filled shape with clipping

In order to draw a clipped and filled shape on PDF page you have to set clipping path and draw through it. PDF specification defines several filling and clipping rules described in sections 8.5.3.3 "Filling" and 8.5.4 "Clipping Path Operators" respectively. These sections contain very detailed description of what areas of the path should be considered "fillable" or "clippable", it's important to understand what *EvenOdd* and *NonZeroWinding* rules are and reading of these sections is highly recommended.

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // radius of the circle
    double radius = 100;
    // circle constant
    double r_c = 0.5522847498*radius;

    // create new document
    FixedDocument document = new FixedDocument();

    // create new page
    Page page = new Page(Boundaries.A4);

    // create and construct new clipping path
    Path clippingPath = new Path(radius,-radius);

    // outer rect
    clippingPath.AppendLine(radius, radius);
    clippingPath.AppendLine(-radius,radius);
    clippingPath.AppendLine(-radius, -radius);
    clippingPath.AppendLine(radius, -radius);

    //inner small rect
    clippingPath.MoveTo(-20, 20);
    clippingPath.AppendLine(20, 20);
    clippingPath.AppendLine(20, -20);
    clippingPath.AppendLine(-20, -20);
    clippingPath.AppendLine(-20, 20);

    // create clipped content using non-zero winding rule and our clipping path,
    // it will clip everything that hits the inner small rect and we will get a "hole"
    // because rectangles in clipping path are drawn in opposite direction
    ClippedContent clippedContent = new ClippedContent(clippingPath, FillRule.Nonzero);
}
```

...code continues on next page


```

// create new path representing a circle
Path path = new Path(0, radius);
path.AppendCubicBezier(r_c, radius, radius, r_c, radius, 0);
path.AppendCubicBezier(radius, -r_c, r_c, -radius, 0, -radius);
path.AppendCubicBezier(-r_c, -radius, -radius, -r_c, -radius, 0);
path.AppendCubicBezier(-radius, r_c, -r_c, radius, 0, radius);

// set current non-stroking color
clippedContent.SetDeviceStrokingColor (new double[] { 0.3, 0.5, 1 });
clippedContent.SetDeviceNonStrokingColor(new double[] { 0.3, 0.5, 1 });
// set line width
clippedContent.SetLineWidth(2.0);
// stroke path
clippedContent.FillAndStrokePath(path);

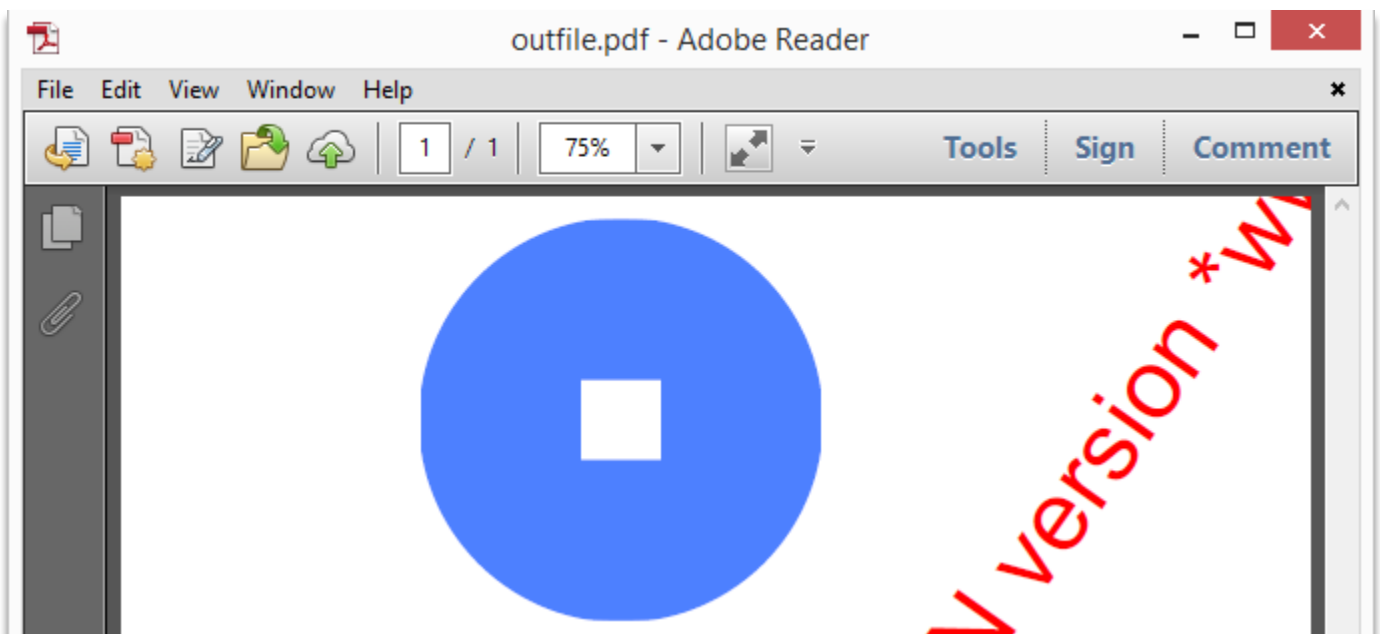
// set current tranform
page.Content.ModifyCurrentTransformationMatrix(1, 0, 0, 1, 250, 730);
// append clipped content
page.Content.AppendContent(clippedContent);

// add page to the document
document.Pages.Add(page);

// save to output stream
document.Save(outputStream);
}

```

The results are shown below:



Pic. 8 Filled shape with clipping applied

3.3.1.6 Drawing standard primitives on PDF page

A number of the static methods provided by `Path` class can be used to draw standard primitives like circle, ellipse, rect etc.

See the code below:

```
using (Stream outputStream = File.Create("primitives.pdf"))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument();
    fixedDocument.Pages.Add(new Page());

    ClippedContent pageContent = fixedDocument.Pages[0].Content;
    pageContent.Translate(50,790);

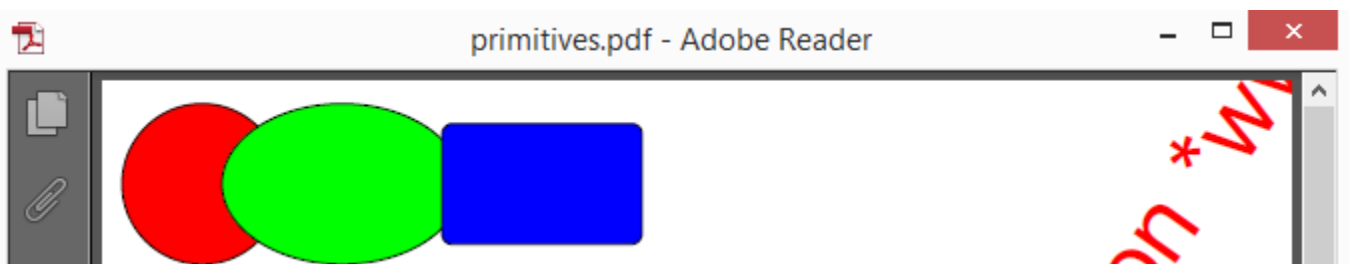
    // create circle
    Path circle = Path.CreateCircle(0, 0, 40);
    pageContent.SetDeviceNonStrokingColor(new double[]{1,0,0});
    pageContent.FillAndStrokePath(circle);

    // create ellipse
    Path ellipse = Path.CreateEllipse(0, 0, 60, 40);
    pageContent.Translate(70,0);
    pageContent.SetDeviceNonStrokingColor(new double[] { 0, 1, 0 });
    pageContent.FillAndStrokePath(ellipse);

    // create roundrect
    Path roundRect = Path.CreateRoundRect(0, 0, 100, 60, 5);
    pageContent.Translate(50, -30);
    pageContent.SetDeviceNonStrokingColor(new double[] { 0, 0, 1 });
    pageContent.FillAndStrokePath(roundRect);

    // save document
    fixedDocument.Save(outputStream);
}
```

Resulting PDF file looks as follows:



Pic. 9 Drawing standard primitives on PDF page

3.3.1.7 Graphics state

One may wonder how we'd be getting color and transformation values back to its original state if we would need to. So far we built our samples in a way that they didn't require any rollbacks, being fairly straightforward. But what if we would have to draw multiple objects on PDF page each having its own transforms and other settings? As you remember from previous articles, drawing commands are being added in sequences and commands that change the state of drawing context often affect subsequent drawings.

In order to avoid this you may perform drawings as isolated sequences and do a quick rollback when it's done. Here the concept of graphics state arises. PDF specification defines it in section 8.4 "Graphics State". In a few words, a graphics state is an internal object that is being used to maintain the state of current graphics context: its colors, clipping, transformations etc. These objects can be enclosed into each other thus making inner state saving possible. Saving and restoring of the *current* graphics state requires a command to be added to the drawing commands sequence. Below is the code that shows it in action.

```
// create clipped content object for drawing
ClippedContent page = new ClippedContent();

// save current state
page.SaveGraphicsState();

... apply transforms, set colors, draw paths

// restore initial state
page.RestoreGraphicsState();

...continue drawing using restored state
```

The sample above shows how to work with *current* graphics state, but it's also possible to set graphics states that you have created in advance and designed for multiple usages. E.g. a graphics state describing some alpha blending operations that you'd like to share and use for many independent drawings.

There is a special type called `GraphicsState` that can be used to achieve this goal, it's located under `Apitron.PDF.Kit.FixedLayout.Resources.GraphicsStates` namespace. These state objects can be set using their identifiers and should be registered as resources first in order to be used.

See the code sample below showing how to set a named graphics state for drawing:

```
// create output PDF file
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // register font resource
    Font font = new Font("F1", StandardFonts.TimesItalic);
    document.ResourceManager.RegisterResource(font);

    // register graphics state resource
    GraphicsState gs = new GraphicsState("gs01") {FontResourceID = "F1", FontSize = 14};

    // register graphics state resource
    document.ResourceManager.RegisterResource(gs);

    Content content = document.Pages[0].Content;

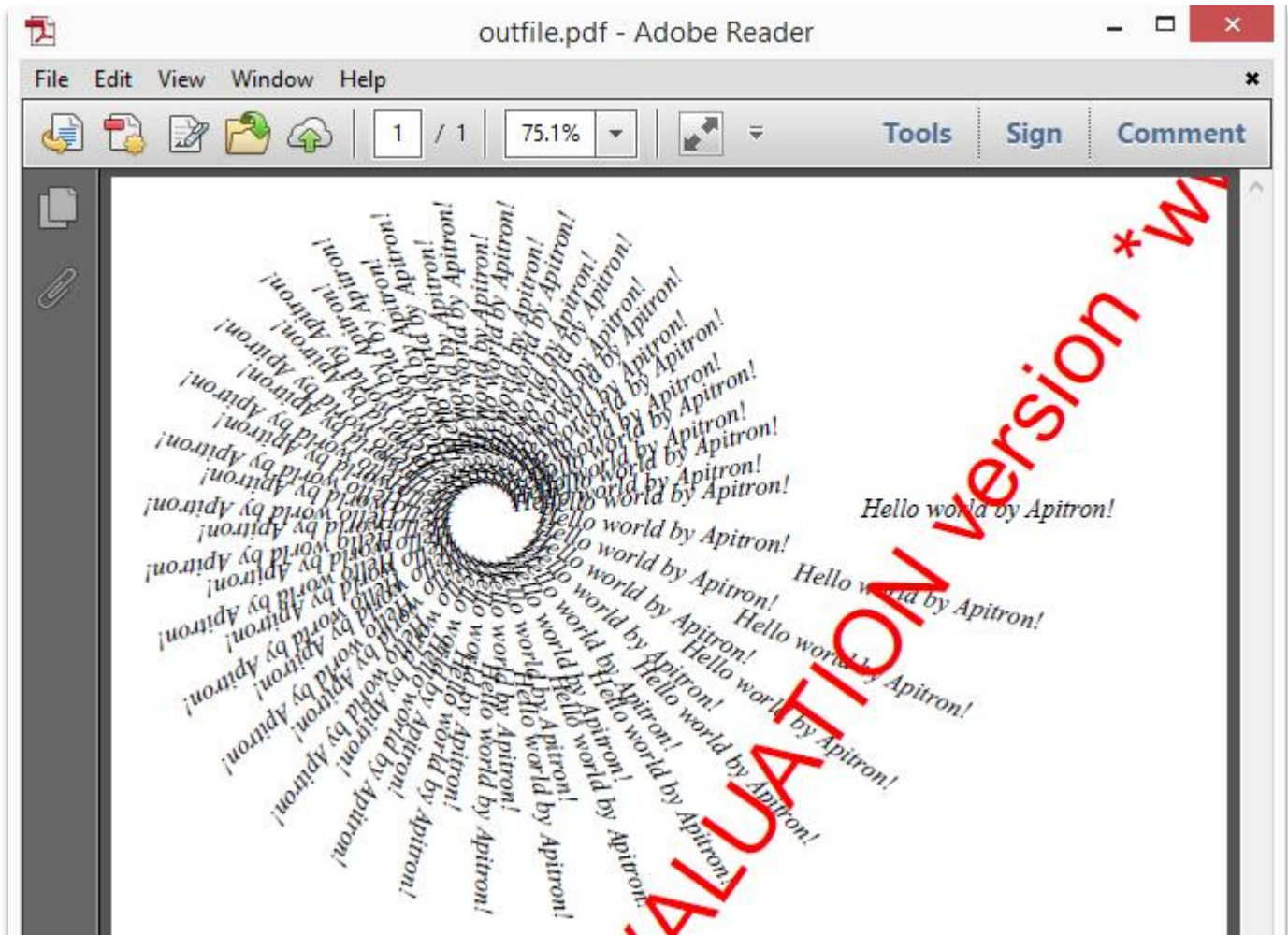
    // font will be set using the graphics state resource
    content.SetGraphicsState("gs01");

    // draw text spiral
    for (double angle = 0; angle <= 4*3.14; angle += 0.2)
    {
        content.SaveGraphicsState();
        content.ModifyCurrentTransformationMatrix(Math.Cos(angle), -Math.Sin(angle), Math.Sin(angle),
            Math.Cos(angle), 200, 660);
        TextObject text = new TextObject();
        text.SetTextMatrix(1, 0, 0, 1, 200/(angle + 1), 0);
        text.SetTextRenderingMode(RenderingMode.FillText);
        text.AppendText("Hello world by Apitron!");
        content.AppendText(text);
        content.RestoreGraphicsState();
    }

    // save document
    document.Save(outputStream);
}
```

This sample draws text along the spiral and uses named graphics state resource object to set current text font. It also uses `SaveGraphicsState` and `RestoreGraphicsState` commands for saving and restoring the state changed during the drawing of each line of text.

Here is the resulting image showing the spiral text generated by the sample code above. You may notice that text in this PDF file uses standard *TimesItalic* font set by applying a named graphics state object.



Pic. 10 Graphics state usage

3.3.1.8 Drawing transparent objects in PDF

Transparent objects can be drawn using the current alpha setting stored in the graphics state. There are two values of type double ranging from 0.0 to 1.0 and indicating the degree of transparency that should be applied to filled or stroked content.

See the following code:

```
using (FileStream outputStream = new FileStream("outfile.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // register graphics state resource
    GraphicsState gs = new GraphicsState("gs01") {CurrentStrokingAlpha = 0.3,
CurrentNonStrokingAlpha = 0.3};
    // register graphics state resource
    document.ResourceManager.RegisterResource(gs);

    Content content = document.Pages[0].Content;

    // select current colors
    content.SetDeviceStrokingColor (new double[] {1,0,0});
    content.SetDeviceNonStrokingColor(new double[] { 0, 1, 0 });

    // create rect path
    Path rect = new Path();
    rect.AppendRectangle(0,0,100,100);

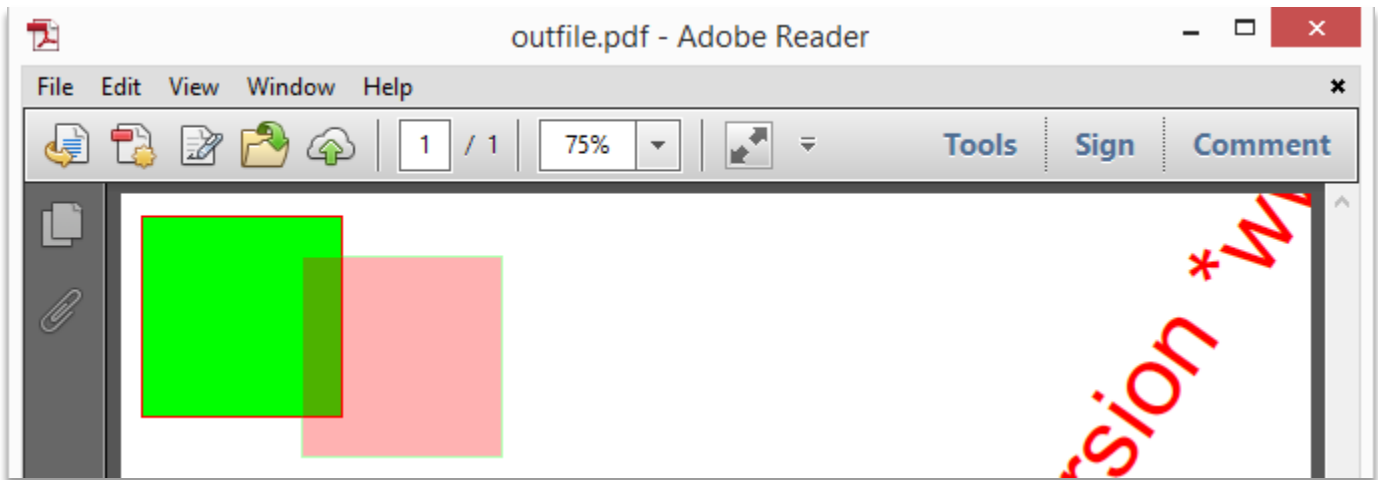
    content.Translate(10,730);
    // fill and stroke this path, it creates an opaque rect
    content.FillAndStrokePath(rect);

    // transparency will be set using the graphics state resource
    content.SetGraphicsState("gs01");
    // select current colors
    content.SetDeviceStrokingColor(new double[] { 0, 1, 0 });
    content.SetDeviceNonStrokingColor(new double[] { 1, 0, 0 });

    // draw a transparent rect
    content.Translate(80, -20);
    content.FillAndStrokePath(rect);

    // save document
    document.Save(outputStream);
}
```

It produces the following results, see image below:



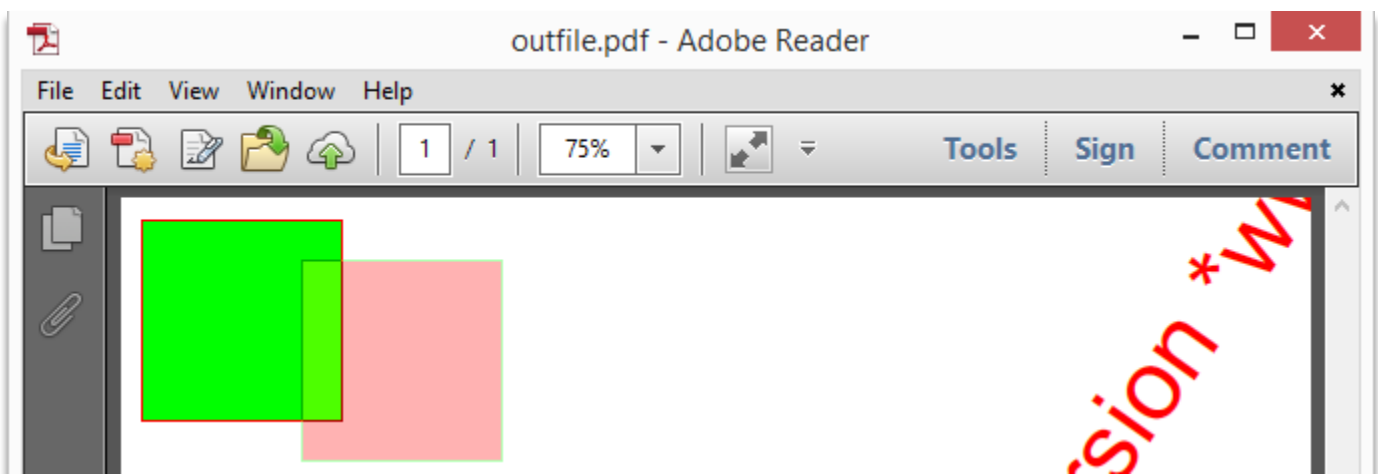
Pic. 11 Drawing transparent object

Transparency in PDF is outlined in section 11 *"Transparency"* of the specification and describes many other things to consider except basic alpha settings. As an example, current alpha could be combined with any blending mode described in section 11.3.5 *"Blend Mode"* to achieve various visual effects. There are also transparency groups and soft mask objects which will be discussed in chapter [3.3.6 Soft Masks](#).

If we would change the blending mode in code sample created the image above from Normal to Exclusion like this:

```
// register graphics state resource
GraphicsState gs = new GraphicsState("gs01") {CurrentStrokingAlpha = 0.3, CurrentNonStrokingAlpha = 0.3,
BlendMode = BlendMode.Exclusion};
```

Then results would be as follows:



Pic. 12 Drawing transparent object with BlendMode=Exclusion

3.3.2 Tiling patterns and pattern colorspaces

While everyone can imagine what a pattern is, a pattern colorspace looks very strange from the first sight. In PDF, it's possible to draw repeating content using objects called tiling patterns, see section 8.7.3 “*Tiling Patterns*” of the specification. There are also other patterns, called shading patterns but they will be discussed later.

A tiling pattern is basically a drawing that is being repeatedly drawn whenever something needs to be filled or stroked with it, therefore *tilled*. PDF treats patterns as a regular *colors* defined in so-called *Pattern Colorspace*. Therefore if something needs to be filled or stroked with the tiling pattern, this pattern is being selected as either non-stroking or stroking color and current colorspace is being set to special type `Pattern`.

Tiling patterns can be of two types, *colored* and *uncolored*. The first one contains all color information needed to draw itself and is a self-contained entity while the second accepts external color settings for its stroking and non-stroking operations making it context dependent.

The code below draws objects using both types of tiling patterns on PDF page one by one:

```
// open and load the file
using (FileStream outputStream = new FileStream("patterns.pdf", FileMode.Create))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    Page page = new Page(new PageBoundary(Boundaries.A4));
    document.Pages.Add(page);

    // set stroking settings
    page.Content.SetLineWidth(2);
    page.Content.SetLineDashPattern(new float[] { 2, 2, 4, 2 }, 3);
    page.Content.SetLineCapStyle(LineCapStyle.Round);
    page.Content.SetLineJoinStyle(LineJoinStyle.Bevel);

    // draw colored pattern
    DrawColoredTilingPattern(page, document);

    // draw uncolored pattern
    DrawUncoloredTilingPattern(page, document);

    document.Save(outputStream);
}
```

Corresponding drawing functions for each pattern are defined in next subsections.

3.3.2.1 Colored tiling patterns

This pattern contains all color information needed to draw itself and doesn't require external parameters to be set for drawing. See the code below:

```
private static void DrawColoredTilingPattern(Page page, FixedDocument document)
{
    // Create and register colored tiling pattern object
    string colorPatternId = "ColoredTilingPattern";

    TilingPattern coloredPattern = new TilingPattern(colorPatternId, new Boundary(0, 0, 30, 20), 30, 20);

    // path defining pattern content
    Path patternPath = new Path();
    patternPath.AppendRectangle(-15, -5, 25, 7);
    patternPath.AppendRectangle(15, -5, 25, 7);
    patternPath.AppendRectangle(-15, 15, 25, 7);
    patternPath.AppendRectangle(15, 15, 25, 7);
    patternPath.AppendRectangle(0, 5, 25, 7);

    // set gray color as fill color for the pattern content
    coloredPattern.Content.SetDeviceNonStrokingColor(0.7);
    coloredPattern.Content.FillAndStrokePath(patternPath);

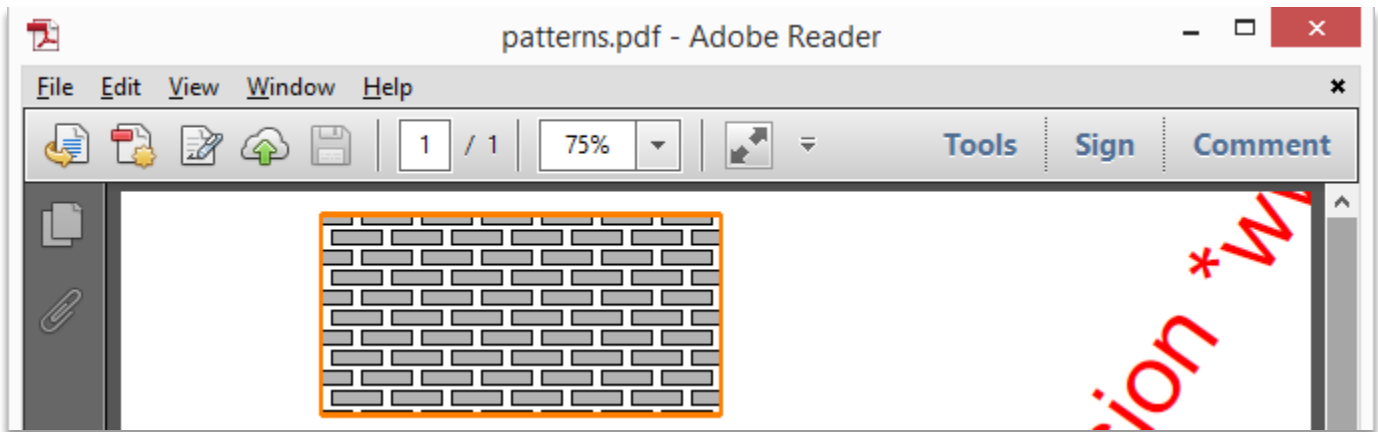
    // register pattern
    document.ResourceManager.RegisterResource(coloredPattern);

    // Set colored Pattern as fill color for the rect (!)
    page.Content.SetNonStrokingColorSpace(PredefinedColorSpaces.Pattern);
    page.Content.SetNonStrokingColor(colorPatternId);

    // Use RGB color to stroke the rect
    page.Content.SetStrokingColorSpace(PredefinedColorSpaces.RGB);
    page.Content.SetStrokingColor(1, 0.5, 0);

    // Draw rectangle
    Path path = new Path();
    path.AppendRectangle(100, 730, 200, 100);
    page.Content.FillAndStrokePath(path);
}
```

The resulting image showing the rectangle filled using the colored tiling pattern is below.



Pic. 13 Colored tiling pattern

3.3.2.2 Uncolored tiling patterns

Uncolored tiling pattern differs from the colored one in way of specifying its stroking and non-stroking colors. It requires the colorspace to be set that will define the actual colors used inside this pattern. In Apitron PDF Kit it can be achieved by using one of the predefined pattern colorspace, e.g. `RGBPattern`.

The code below draws uncolored pattern and fills the rect using it.

```
private static void DrawUncoloredTilingPattern(Page page, FixedDocument document)
{
    // Create and register uncolored tiling pattern object
    string uncoloredPatternId = "UnColoredTilingPattern";

    TilingPattern uncoloredPattern = new TilingPattern(uncoloredPatternId, new Boundary(0, 0, 30, 20),
    30, 20, false);

    // path defining pattern content
    Path patternPath = new Path();
    patternPath.AppendRectangle(-15, -5, 25, 7);
    patternPath.AppendRectangle(15, -5, 25, 7);
    patternPath.AppendRectangle(-15, 15, 25, 7);
    patternPath.AppendRectangle(15, 15, 25, 7);
    patternPath.AppendRectangle(0, 5, 25, 7);

    uncoloredPattern.Content.FillAndStrokePath(patternPath);

    ...code continues on next page
```

```

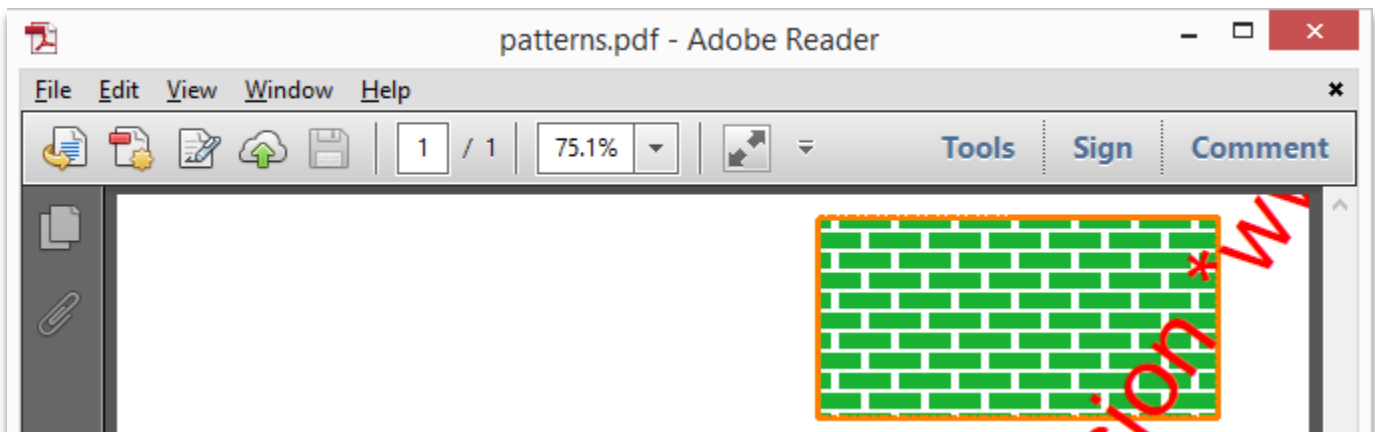
// register pattern
document.ResourceManager.RegisterResource(uncoloredPattern);

// Set uncolored Pattern as fill color space for the rect and RGB colorspace for its internal colors (!)
page.Content.SetNonStrokingColorSpace(PrefdefinedColorSpaces.RGBPattern);
// set pattern as fill color using its id and also set its internal fill color
page.Content.SetNonStrokingColor(uncoloredPatternId, 0.1, 0.7, 0.2);
// set stroking color as device RGB
page.Content.SetDeviceStrokingColor(1,0.5,0);

// Draw rectangle
Path path = new Path();
path.AppendRectangle(350, 730, 200, 100);
page.Content.FillAndStrokePath(path);
}

```

You see that after setting of the RGBPattern colorspace as non-stroking we also set our pattern as current non-stroking color along with specifying internal fill color for the pattern. The resulting image that shows our rectangle filled with uncolored pattern is below.



Pic. 14 Uncolored tiling pattern

Uncolored patterns provide a good way for sharing various repeating drawings and make them using filling and stroking colors assigned on demand.

3.3.2.3 Shading patterns, complex fills in PDF

Shading patterns provide a smooth transition between colors across an area to be painted, independent of the resolution of any particular output media and without specifying the number of steps in the color transition. Patterns of this type can be used to create complex fills and have several subtypes described in section 8.7.4.5 “*Shading Types*” of the specification .

Separate article is being prepared to cover this area providing samples for all kinds of shading patterns and creation techniques.

3.3.3 Form XObjects

Form XObject is a self-contained description of any sequence of graphics objects (including path objects, text objects, and sampled images). A *form XObject* may be painted multiple times either on several pages or at several locations on the same page-and produces the same results each time, subject only to the graphics state at the time it is invoked.

Not only is this shared definition economical to represent in the PDF file, but under suitable circumstances the conforming reader can optimize execution by caching the results of rendering the *form XObject* for repeated reuse. See section 8.10 “*Form XObjects*” of the PDF specification for the detailed description.

So, basically *form XObject* is a group of drawing commands which can be repeatedly played at some place but unlike the tiling pattern for example, it requires to be called explicitly each time one needs to draw its content at some place.

Apitron PDF Kit for .NET provides a `FixedContent` class that can be used to create form XObjects and fill them with drawing commands.

3.3.3.1 Drawing a simple formXObject

Consider the following code demonstrating how to create and use simple form XObject:

```
// create output PDF file
using (FileStream outputStream = new FileStream("xobjects.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // create XObject
    FixedContent xObject = new FixedContent("myXObject", new Boundary(0, 0, 200, 100));

    // register XObject
    document.ResourceManager.RegisterResource(xObject);

    // select fill and stroke colors
    xObject.Content.SetDeviceNonStrokingColor(new double[] {1,0,0});
    xObject.Content.SetDeviceStrokingColor(new double[] {0});
    // create path and fill it
    Path path = new Path();
    path.AppendRectangle(10,10,150,80);
    xObject.Content.FillAndStrokePath(path);

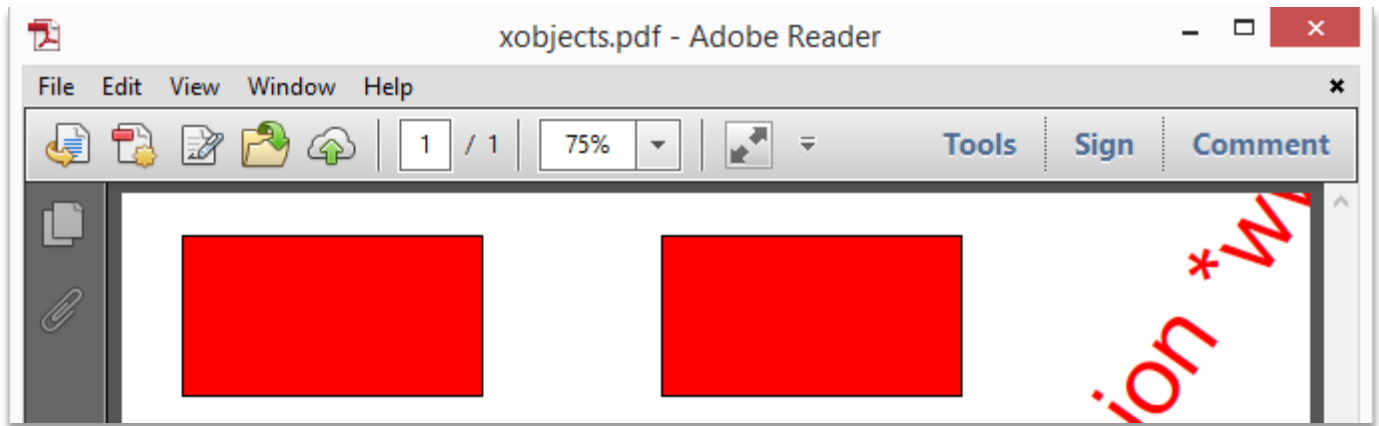
    // set offset
    document.Pages[0].Content.Translate(20,730);
    // append xObject to page content
    document.Pages[0].Content.AppendXObject("myXObject");

    // set offset
    document.Pages[0].Content.Translate(240,0);
    // append xObject to page content
    document.Pages[0].Content.AppendXObject("myXObject");

    // save document
    document.Save(outputStream);
}
```

You see that we create new XObject, define its boundaries and *register* it. It's important to not forget to register the XObject otherwise you won't be able to place it on page. Later we draw this XObject twice by calling `AppendXObject()` on `Page.Content` instance.

The resulting image demonstrating two identical rectangles drawn as XObjects:



Pic. 15 Drawing a simple form XObject

Using this technique you may reduce file size and optimize your PDF generation routines. Commands needed to paint an object, would be created only once when you define it and this definition would be reused every time it needs to be painted.

3.3.4 Transparency Group XObjects

A *Transparency Group XObject* is a special type of *Group Form XObject* (see section 8.10.3 “*Group XObjects*” in PDF specification) that can be used to group graphical elements together and have extended support for transparency-related operation. Transparency groups and their properties are described in section 11.6.6 “*Transparency Group XObjects*” of the PDF specification. Any group XObject is a [form XObject](#) with an additional attribute describing its type. Since there is only single type of *Group XObjects* defined in PDF specification so far, it only makes sense to discuss *transparency groups*.

Apitron PDF Kit supports creation of transparency groups which can be used for various reasons. As an example you may check chapter [3.3.6.1 Create soft mask from arbitrary drawing](#) which shows how to use them to create advanced masking effects. Transparency groups can be created using `TransparencyGroup` class and their properties can be changed via `Attributes` property.

Most important properties of a transparency group define whether the group should be treated as “isolated” and “knockout”. These terms describe an interaction between group as a whole with its background or other elements on page and also interaction between objects within a group. These topics are rather complicated and are explained in details in sections 11.2 “*Overview of Transparency*”, 11.4.5 “*Isolated Groups*”, 11.4.6 “*Knockout Groups*” of PDF specification. See also figures L.16 and L.17 on page 741 of PDF specification.

3.3.4.1 Draw objects using Transparency Group

The code sample below demonstrates how to use transparency group for drawing several objects at once and its difference from “normal” drawing when transparency becomes involved. It draws two sets of objects over the same image and sets current alpha state:

- a transparency group containing colored rectangles
- a set of ungrouped colored rectangles

```
// create output file
using (Stream outputStream = File.Create("transparency_group.pdf"))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument();
    fixedDocument.Pages.Add(new Page());

    fixedDocument.ResourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.XObjects.Image("logo", "apitron.png"));
    // create graphics state, it will be used to define group transparency
    GraphicsState graphicsState = new GraphicsState("semiTransparent"){CurrentNonStrokingAlpha =
    0.7};
    fixedDocument.ResourceManager.RegisterResource(graphicsState);

    // create transparency group and draw several rects inside it
    TransparencyGroup transparencyGroup = new TransparencyGroup("myGroup", new
    Boundary(0,0,150,150));
    fixedDocument.ResourceManager.RegisterResource(transparencyGroup);

    // draw colored rects inside the transparency group
    Path path = new Path();
    path.AppendRectangle(20, 20, 50, 50);

    Action<ClippedContent,double, double, double[]> drawRect = (content,tx, ty, color) =>
    {
        content.Translate(tx, ty);
        content.SetDeviceNonStrokingColor(color);
        content.FillAndStrokePath(path);
    };

    drawRect(transparencyGroup.Content,0, 0, new double[] {1, 0, 0});
    drawRect(transparencyGroup.Content,30, 30, new double[] { 0, 1, 0 });
    drawRect(transparencyGroup.Content,30, 30, new double[] { 0, 0, 1 });

    ClippedContent pageContent = fixedDocument.Pages[0].Content;
```

...code continues on next page


```

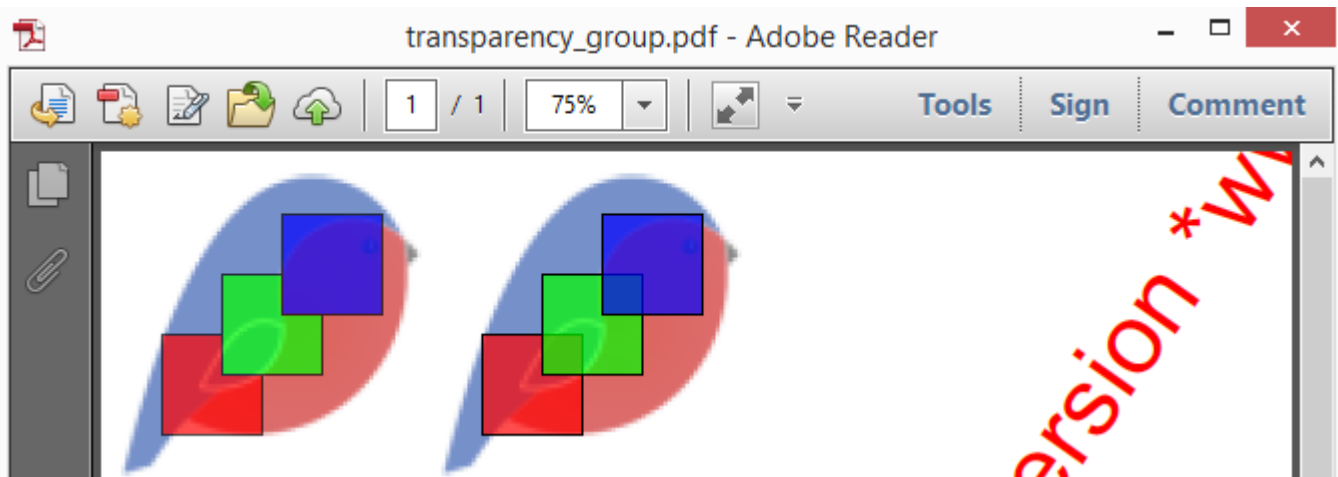
// draw group over an image (1)
pageContent.SaveGraphicsState();
// draw bg image
pageContent.Translate(10, 680);
pageContent.AppendImage("logo",0,0,150,150);
// set state and draw transparency group
pageContent.SetGraphicsState("semiTransparent");
pageContent.AppendXObject("myGroup");
pageContent.RestoreGraphicsState();

// draw rects over an image (2)
pageContent.SaveGraphicsState();
// draw bg image
pageContent.Translate(170, 680);
pageContent.AppendImage("logo", 0, 0, 150, 150);

// set state and draw colored rects directly on page
pageContent.SetGraphicsState("semiTransparent");
drawRect(pageContent, 0, 0, new double[] { 1, 0, 0 });
drawRect(pageContent, 30, 30, new double[] { 0, 1, 0 });
drawRect(pageContent, 30, 30, new double[] { 0, 0, 1 });
pageContent.RestoreGraphicsState();
// save document
fixedDocument.Save(outputStream);
}

```

This code produces the following results:



Pic. 14 Draw objects using Transparency Group

Note that rectangles on the left, which were included into the transparency group, were treated as a whole transparent thing, while rectangles on the right were treated as separate transparent objects. This sample demonstrates one of the possible usages of transparency groups.

3.3.5 Images

Images in PDF are represented as special type of XObject called *image XObject*. Like the *form XObject* described in section [3.3.3 Form XObjects](#), this type of XObject can be used to place its content repeatedly when it's needed, using the single registered instance.

Image coordinate system in PDF differs from the default and has its Y-axis inverted making it similar to Windows GDI coordinate system. It's important to take this into account for proper positioning of the image using regular drawing commands. We also designed several helper methods handling this difference and providing a more convenient way to draw an image.

Apitron PDF Kit for .NET supports the following source image formats for creation of image XObjects: *BMP*, *PNG*, *JPEG*, *TIFF*, *JP2*. It has a special `Image` resource class defined in `Apitron.PDF.Kit.FixedLayout.Resources.XObjects` namespace that can be used to create such XObjects. Image settings can be changed using properties such as `Width`, `Height`, `Interpolate` etc.

Images and their attributes are discussed in depth in section 8.9 “*Images*” of the PDF specification.

3.3.5.1 Drawing an image on PDF page

The code below demonstrates how to use image XObject for drawing images:

```
// create output PDF file
using (FileStream outputStream = new FileStream("xobjects.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // create XObject
    Image xObject = new Image("myImageXObject", "apitron.png");

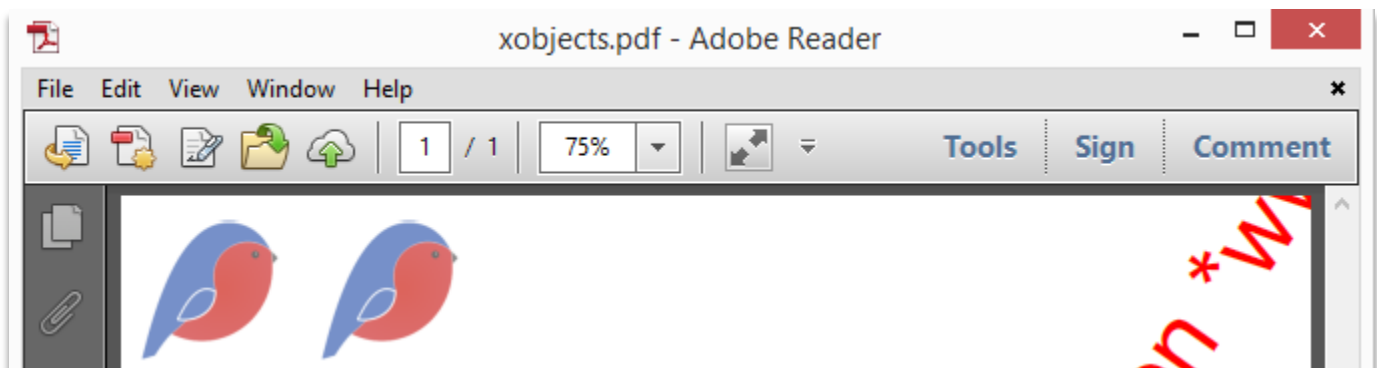
    // register XObject
    document.ResourceManager.RegisterResource(xObject);

    // append two identical images using image XObject
    document.Pages[0].Content.AppendImage("myImageXObject", 10, 760,
xObject.Width, xObject.Height);

    document.Pages[0].Content.AppendImage("myImageXObject", 100, 760,
xObject.Width, xObject.Height);

    // save document
    document.Save(outputStream);
}
```

The result is shown below; please note that we used a special method `AppendImage()` which performs all transformations needed to draw an image. Usually, drawing an image requires us to set current scale transform because image is considered to be 1x1 by default. The invoked member `AppendImage()` is simply a wrapper that does necessary scaling for us and invokes `AppendXObject()`.



Pic. 15 Drawing an image XObject

3.3.5.2 Applying color key mask to an image

It's possible to define a mask for the image using special color that will be used as a clipping source for it. One may use `MaskColorRanges` property for it. See the code:

```
using (FileStream outputStream = new FileStream("xobjects.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

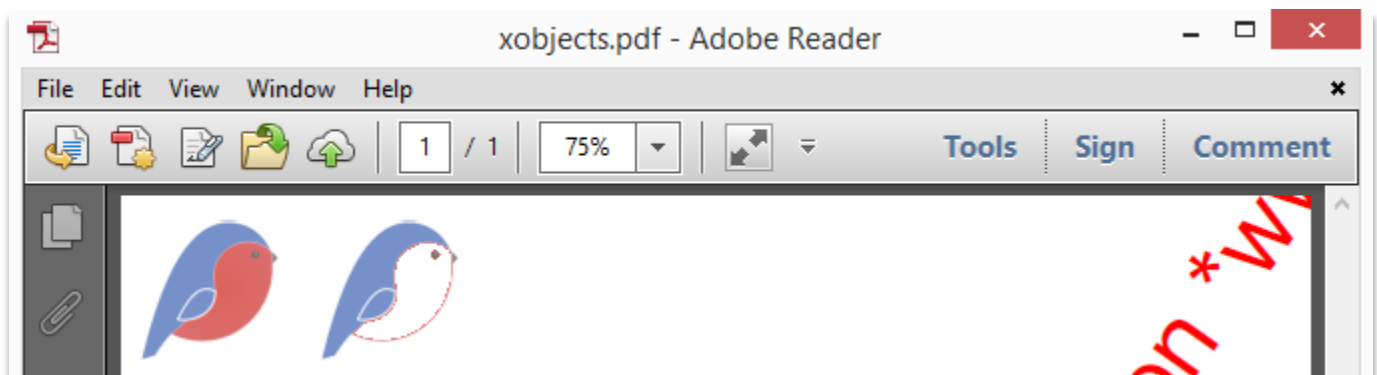
    // createXObject
    Image normalImage = new Image("myImageXObject", "apitron.png");
    Image maskedImage = new Image("myImageXObjectMasked", "apitron.png");
    // masking colors should be specified in image colorspace and include min and max for each channel
    maskedImage.MaskColorRanges = new double[] {210, 220, 90, 115, 85, 115 };

    // register XObjects
    document.ResourceManager.RegisterResource(normalImage);
    document.ResourceManager.RegisterResource(maskedImage);

    // append normal and masked images one by one
    document.Pages[0].Content.AppendImage("myImageXObject", 10, 760, normalImage.Width,
normalImage.Height);
    document.Pages[0].Content.AppendImage("myImageXObjectMasked", 100, 760,
maskedImage.Width, maskedImage.Height);

    // save document
    document.Save(outputStream);
}
```

We specify that RGB colors falling in range $[(210, 90, 85), (220, 115, 115)]$ should be masked out. The result can be seen on the image below:



Pic. 16 Applying color key mask to an image

3.3.5.3 Using image as stencil mask for drawing

An image itself can be used as a source of mask data marking areas where *current non-stroking color* can be painted over and where it's not allowed. Ideally, to create such a mask, this image should be *monochrome* but we can also accept color images converting them to black and white if needed. It creates a binary mask which can be used to paint over.

Consider the code below:

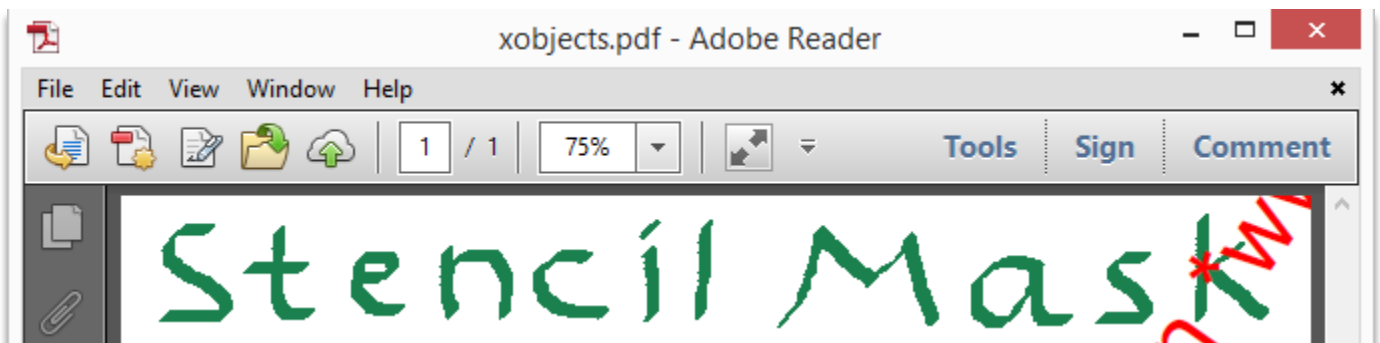
```
// create output PDF file
using (FileStream outputStream = new FileStream("xobjects.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // create XObject defining a stencil mask
    Image stencilmask = new Image("maskXObject", "stencil_mask.png");
    stencilmask.IsStencilMask = true;
    // register xObject
    document.ResourceManager.RegisterResource(stencilmask);

    // set the non-stroking color
    document.Pages[0].Content.SetDeviceNonStrokingColor(new double[] {0.1,0.5,0.3});
    // draw using current non-stroking color and the mask set
    document.Pages[0].Content.AppendImage("maskXObject", 10, 770,
    stencilmask.Width,stencilmask.Height);

    document.Save(outputStream);
}
```

Note that `Image.IsStencilMask` property was used to define that the image should be processed as a stencil mask; this code produces the following results:



Pic. 17 Using an image as a stencil mask for drawing

3.3.5.4 Applying explicit mask to an image

Explicit masking assumes that you would like to draw an image using a custom mask and it combines stencil masking with regular image drawing by setting stencil mask directly to an image XObject. Let's see the code:

```
// create output PDF file
using (FileStream outputStream = new FileStream("xobjects.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

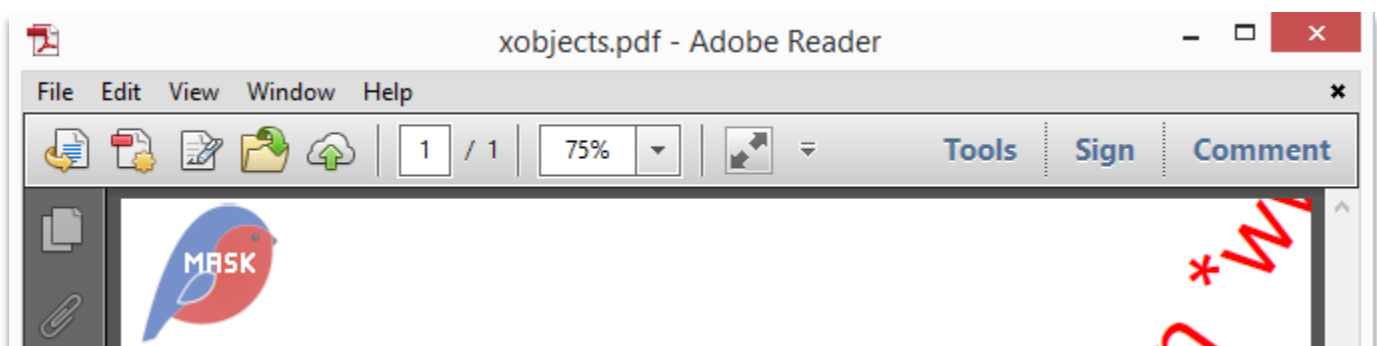
    // create XObject defining a mask
    Image stencilmask = new Image("maskXObject", "apitron_mask.png");
    // register mask xObject
    document.ResourceManager.RegisterResource(stencilmask);

    // create XObject for the image and assign an explicit mask to it
    Image normalImage = new Image("imageXObject", "apitron.png");
    normalImage.MaskResourceID = "maskXObject";

    // register mask XObject
    document.ResourceManager.RegisterResource(normalImage);

    // draw masked image and save file
    document.Pages[0].Content.AppendImage("imageXObject", 10, 770, normalImage.Width,
    normalImage.Height);
    document.Save(outputStream);
}
```

Here we simply create two image objects and assign the first one, representing the explicit mask, to the image we want to be drawn masked using its `MaskResourceID` property. Results are as follows, note the “MASK” word cut in the middle of the image:



Pic. 18 Applying explicit mask to an image

3.3.6 Soft masks

In chapter [3.3.5.3 Using image as stencil mask for drawing](#), we have discussed how to create a stencil mask using image as a source for mask values and draw through it with current non-stroking color. Chapter [3.3.5.4 Applying explicit mask to an image](#), showed how to do the same for an image itself. But stencil masks are quite limited in terms of defining transparency. You can only set whether the particular location within the mask is fully transparent or not, making it impossible to define a mask containing semi-transparent areas.

Soft masks discussed in this chapter, are designed to create complex masking effects which are impossible to achieve with any other type of masking. The word soft emphasizes that the mask value at a given point is not limited to just 0.0 or 1.0 but can take on intermediate fractional values as well. Such a mask is typically the only means of providing position-dependent opacity values, since elementary objects do not have intrinsic opacity of their own. It is fully described in section 11.5 “*Soft Masks*” of the PDF specification.

A special class was designed to work with such masks and it's called `SoftMask`, it's a resource object that in its turn uses `TransparencyGroup` instance as a source for soft mask values. `TransparencyGroup` is special kind of a *form XObject* which is being discussed in chapter [3.3.4 Transparency Group XObjects](#) and, among other things, can provide soft mask values derived either from its alpha or luminosity calculated for any given point. It implements `ClippedContent` and therefore can contain same drawing commands as other implementers.

To apply a `SoftMask` while drawing, one have to create a [GraphicsState](#) object and apply it by calling `SetGraphicsState()` function on the desired `ClippedContent` instance before any further drawing occurs.

3.3.6.1 Create soft mask from arbitrary drawing

Consider the following code that draws two rectangles. The red one on the left is drawn without any masking and the blue on the right using the soft mask shaped as circle. Luminosity values are used to derive soft mask from the transparency group containing the drawn circle.

```
// create output PDF file
using (FileStream outputStream = new FileStream("softmask.pdf", FileMode.Create, FileAccess.Write))
{
    // radius of the circle
    double radius = 100;
    // circle constant
    double r_c = 0.5522847498*radius;
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // an object to be used as source of soft mask values,
    // these values will be provided by calculating its luminosity at the given point
    SoftMask softMask = new SoftMask("softMask", SoftMaskSubtype.Luminosity, new Boundary(200,
200));

    // set the background color so areas where nothing will be painted will have max luminosity
    softMask.BackgroundColor = new double[]{1};

    // draw to transparency group XObject that soft mask will use to derive values from
    Content maskContent = softMask.Group.Content;

    // create new path representing a circle
    Path path = new Path(0, radius);
    path.AppendCubicBezier(r_c, radius, radius, r_c, radius, 0);
    path.AppendCubicBezier(radius, -r_c, r_c, -radius, 0, -radius);
    path.AppendCubicBezier(-r_c, -radius, -radius, -r_c, -radius, 0);
    path.AppendCubicBezier(-radius, r_c, -r_c, radius, 0, radius);

    // move it to the center of the mask area
    maskContent.Translate(100,100);
    // set current non-stroking color
    maskContent.SetDeviceNonStrokingColor(new double[] { 0.6 });
    // stroke path
    maskContent.FillPath(path);

    // register soft mask resource
    document.ResourceManager.RegisterResource(softMask);

    // create new graphics state and prepat it to use soft mask
    GraphicsState maskingState = new GraphicsState("gsMask");
    maskingState.SoftMaskResourceID = softMask.ResourceID.ID;
```



```

// register mask state
document.ResourceManager.RegisterResource(maskingState);

ClippedContent pageContent = document.Pages[0].Content;
// select current color and drawing origin
pageContent.SetDeviceNonStrokingColor(new double[] { 1, 0, 0 });
pageContent.Translate(10, 630);

// draw non-clipped RED rect
Path rect1 = new Path();
rect1.AppendRectangle(0, 0, 200, 200);
pageContent.FillAndStrokePath(rect1);

pageContent.Translate(100,-20);

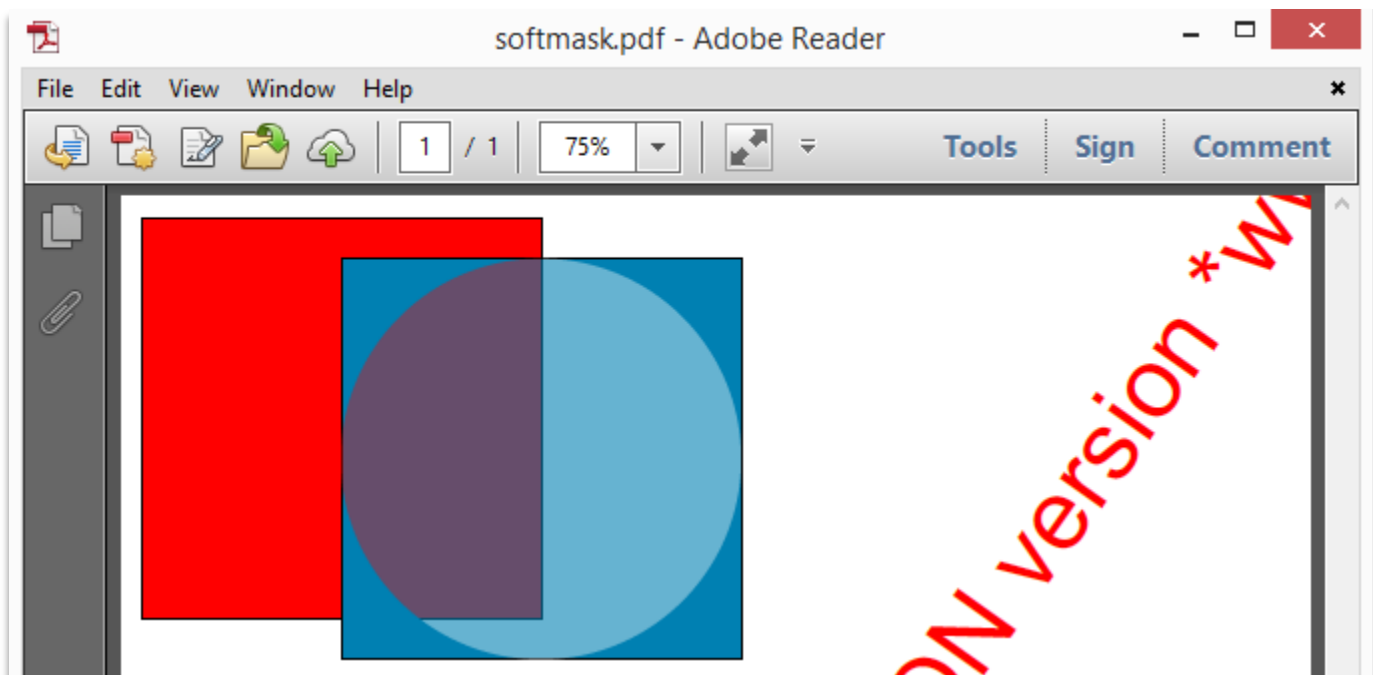
// select current color
pageContent.SetDeviceNonStrokingColor(new double[] { 0, 0.5, 0.7 });

// apply masking graphics state that sets the soft mask
pageContent.SetGraphicsState(maskingState.ResourceID.ID);

// draw BLUE rect using the soft mask set
Path rect2 = new Path();
rect2.AppendRectangle(0, 0, 200, 200);
pageContent.FillAndStrokePath(rect2);
document.Save(outputStream);
}

```

The resulting PDF file looks as follows:



Pic. 19 Create soft mask from arbitrary drawing

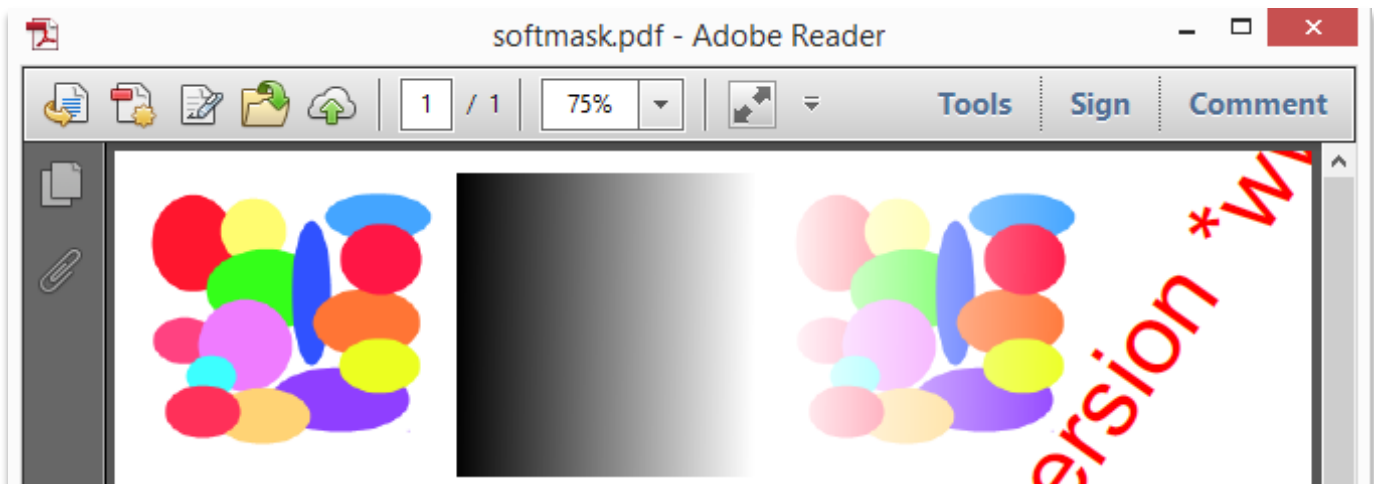
3.3.6.2 Specify a soft mask for an image

It's possible to assign a soft mask to image XObject and it can be done by associating a *soft-mask image* with it. *Soft-mask image* is a subsidiary image XObject which resource ID is being assigned to the `SoftMaskResourceID` property of the parent image XObject. See the code below:

```
using (FileStream outputStream = new FileStream("softmask.pdf", FileMode.Create, FileAccess.Write))
{
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());
    // create image XObject to be used as a mask and register it
    Image softMaskImage = new Image("softMaskImage", "softmask.png");
    document.ResourceManager.RegisterResource(softMaskImage);
    // create image XObject and register it
    Image image = new Image("image", "image.png");
    document.ResourceManager.RegisterResource(image);

    // create masked image XObject, register it and set the mask
    Image maskedImage = new Image("maskedImage", "image.png");
    // set soft mask resource id to the id of the masking image
    maskedImage.SoftMaskResourceID = softMaskImage.ResourceID.ID;
    document.ResourceManager.RegisterResource(maskedImage);
    // draw images and save file
    document.Pages[0].Content.AppendImage(image.ResourceID.ID, 10, 680, 150, 150);
    document.Pages[0].Content.AppendImage(softMaskImage.ResourceID.ID, 170, 680, 150, 150);
    document.Pages[0].Content.AppendImage(maskedImage.ResourceID.ID, 330, 680, 150, 150);
    document.Save(outputStream);
}
```

The screenshot below shows three images: original, soft mask and the masked result.



Pic. 20 Specify a soft mask for an image

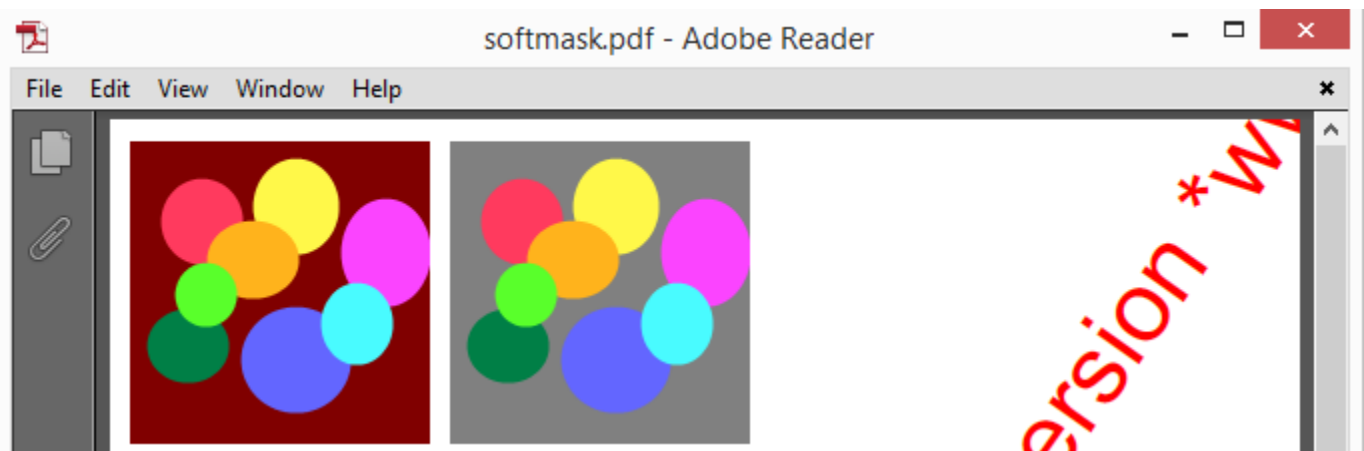
3.3.6.3 Support for transparent images

Apitron PDF Kit provides support for images which have transparency data included along with the color data. Transparent PNGs or BMPs are a good example. It can be enabled on creation of the image XObject by setting `useTransparency` parameter to *true*. See the code below:

```
using (FileStream outputStream = new FileStream("softmask.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());
    // create image XObject and set it to use embedded transparency data
    Image maskedImage = new Image("maskedImage", "colored_bubbles.png", true);
    document.ResourceManager.RegisterResource(maskedImage);
    // create rect
    Path path = new Path();
    path.AppendRectangle(0,0,150,150);

    ClippedContent pageContent = document.Pages[0].Content;
    // draw dark red rect and transparent image over it
    pageContent.Translate(10,680);
    pageContent.SetDeviceNonStrokingColor(new double[]{0.5,0,0});
    pageContent.FillPath(path);
    pageContent.AppendImage(maskedImage.ResourceID.ID,0,0,150,150);
    // draw gray rect and transparent image over it
    pageContent.Translate(160,0);
    pageContent.SetDeviceNonStrokingColor(new double[]{0.5,0.5,0.5});
    pageContent.FillPath(path);
    pageContent.AppendImage(maskedImage.ResourceID.ID,0,0,150,150);
    document.Save(outputStream);
}
```

Image below demonstrates a transparent PNG drawn over two different rectangles:



Pic. 21 Support for transparent images

3.4 Document-level navigation

PDF standard describes several ways which can be used to implement document-level navigation. For these purposes a few PDF objects were defined, namely *Destinations* and *Document Outlines* (sometimes referred as “*Bookmarks*”).

A destination defines a particular view of a document, consisting of the following items:

- The page of the document that shall be displayed
- The location of the document window on that page
- The magnification (zoom) factor

Destinations may be associated with outline items (see section 12.3.3 “*Document Outline*” of the specification), annotations (see chapter [3.5.2 Use link annotations for quick navigation](#)) or actions (see chapter [3.6.1 Navigate to the parts of PDF document using Go-To action](#)). In each case, the destination specifies the view of the document that shall be presented when the outline item or annotation is being opened or the action is being performed. A destination may be specified either explicitly, by creating an instance of the `Destination` class and defining its properties or indirectly, using its name.

A PDF document may contain a document outline that the conforming reader may display on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of outline items (sometimes called bookmarks), which serve as a visual table of contents to display the document structure to the user.

The user may interactively open and close individual items by clicking on them with the mouse. When an item is open, its immediate children in the hierarchy shall become visible on the screen; each child may in turn be opened or closed, selectively revealing or hiding further parts of the hierarchy. When an item is closed, all of its descendants in the hierarchy shall be hidden. Clicking the text of any visible item activates the item, causing the conforming reader to jump to a destination or trigger an action associated with the item.

3.4.1 Add links using explicit and named destinations

The code below demonstrates how to add two link annotations onto the page of PDF document and set their targets using explicit and named `Destination` objects.

```
// create output PDF file
using (FileStream outputStream = new FileStream("navigation.pdf", FileMode.Create,
FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();

    document.Pages.Add(new Page());
    // second page we are going to add a link to
    Page page2 = new Page();
    document.Pages.Add(page2);

    // create and register named destination object
    document.Destinations.Add("myDestination", new Destination(page2, new DestinationTypeFit()));

    // add and place text object containing text for our links
    TextObject textObject = new TextObject(StandardFonts.Helvetica, 14);
    textObject.AppendText("Navigate to page 2 using explicit destination");
    textObject.MoveToNextLine(0, -30);
    textObject.AppendText("Navigate to page 2 using named destination");

    document.Pages[0].Content.Translate(10, 815);
    document.Pages[0].Content.AppendText(textObject);

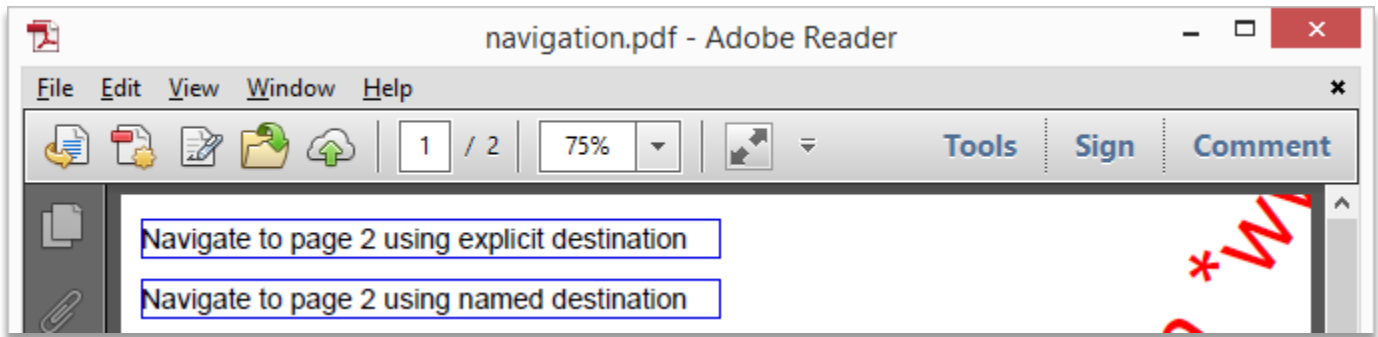
    // create link annotation using an explicit destination
    LinkAnnotation linkAnnotation1 = new LinkAnnotation(new Boundary(10, 810, 300, 830));
    linkAnnotation1.BorderStyle = new AnnotationBorderStyle(1);
    linkAnnotation1.Color = new double[] { 0, 0, 0.9 };
    linkAnnotation1.Destination = new Destination(page2, new DestinationTypeFit());

    // create link annotation using named destination
    LinkAnnotation linkAnnotation2 = new LinkAnnotation(new Boundary(10, 780, 300, 800));
    linkAnnotation2.BorderStyle = new AnnotationBorderStyle(1);
    linkAnnotation2.Color = new double[] { 0, 0, 0.9 };
    linkAnnotation2.Destination = new Destination("myDestination");

    // add annotations to the PDF page
    document.Pages[0].Annotations.Add(linkAnnotation1);
    document.Pages[0].Annotations.Add(linkAnnotation2);
    document.Save(outputStream);
}
```

Usage of explicit destinations is pretty straightforward, while named destinations require registration in their parent document. Using named destination it becomes possible to separate reference generation from actual content generation. You may generate a list of links using destinations names first and later register these destinations pointing to correct locations.

Resulting PDF file is shown on the image below:



Pic. 22 Add links using explicit and named destinations

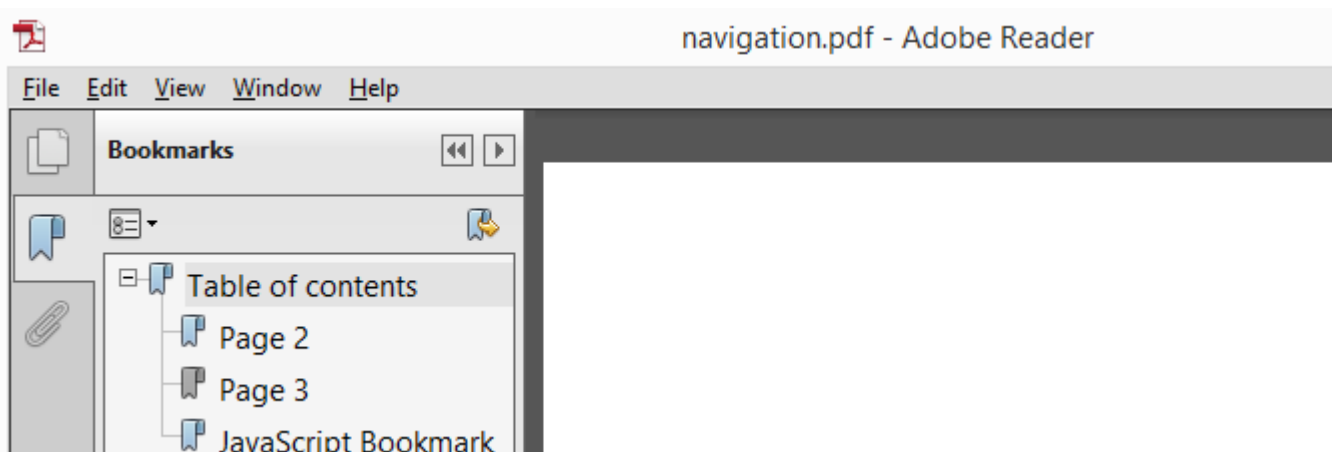
3.4.2 Working with PDF Bookmarks

Document Bookmarks or *Outlines*, as they are named in PDF specification, can be enumerated, added or edited using `Bookmark` class provided by Apitron PDF Kit. The `Bookmarks` property of the `FixedDocument` class is designed for this purpose. Each bookmark element in its turn has `Bookmarks` property holding its child bookmarks and so on. A bookmark can have either destination or action assigned. Consider the code below:

```
// create output PDF file
using (FileStream outputStream = new FileStream("navigation.pdf", FileMode.Create,
FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    // add three pages
    document.Pages.Add(new Page());
    document.Pages.Add(new Page());
    document.Pages.Add(new Page());

    // add root bookmark and set it open by default
    Bookmark rootBookmark = new Bookmark(document.Pages[0], "Table of contents");
    rootBookmark.IsOpen = true;
    document.Bookmarks.AddFirst(rootBookmark);
    // add nested bookmarks
    rootBookmark.AddLast(new Bookmark(new Destination(document.Pages[1]), "Page 2"));
    rootBookmark.AddLast(new Bookmark(new GoToAction(document.Pages[2]), "Page 3"));
    rootBookmark.AddLast(new Bookmark(new JavaScriptAction("app.alert('Hello World!');"),
"JavaScript Bookmark"));
    document.Save(outputStream);
}
```

The image below demonstrates the TOC containing JS bookmark invoking the alert window:



Pic. 23 Working with PDF Bookmarks

3.5 Annotations

An annotation associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a way to interact with the user by means of the mouse and keyboard. PDF includes a wide variety of standard annotation types, described in detail in section 12.5.6 “*Annotation Types*” of the PDF specification.

Many of the standard annotation types may be displayed in either the open or the closed state. When closed, they appear on the page in some distinctive form, such as an icon, a box, or a rubber stamp, depending on the specific annotation type. When the user activates the annotation by clicking it, it exhibits its associated object, such as by opening a pop-up window displaying a text note or by playing a sound or a movie.

This is how the PDF specification defines an annotation and there is not much to add. The most commonly used annotations are text, link or popup annotations while other types are less known. Nevertheless all of them can be created, added and saved to PDF file using Apitron PDF Kit for .NET.

Annotations can be added or removed using `Annotations` property of the `Page` instance; it provides access to a collection containing all annotations which exist on the page.

3.5.1 Add text annotation to the PDF page

A text annotation represents a “sticky note” attached to a point in the PDF document. When closed, the annotation shall appear as an icon; when open, it shall display a pop-up window containing the text of the note in a font and size chosen by the conforming reader.

See the code below that creates simple text annotation:

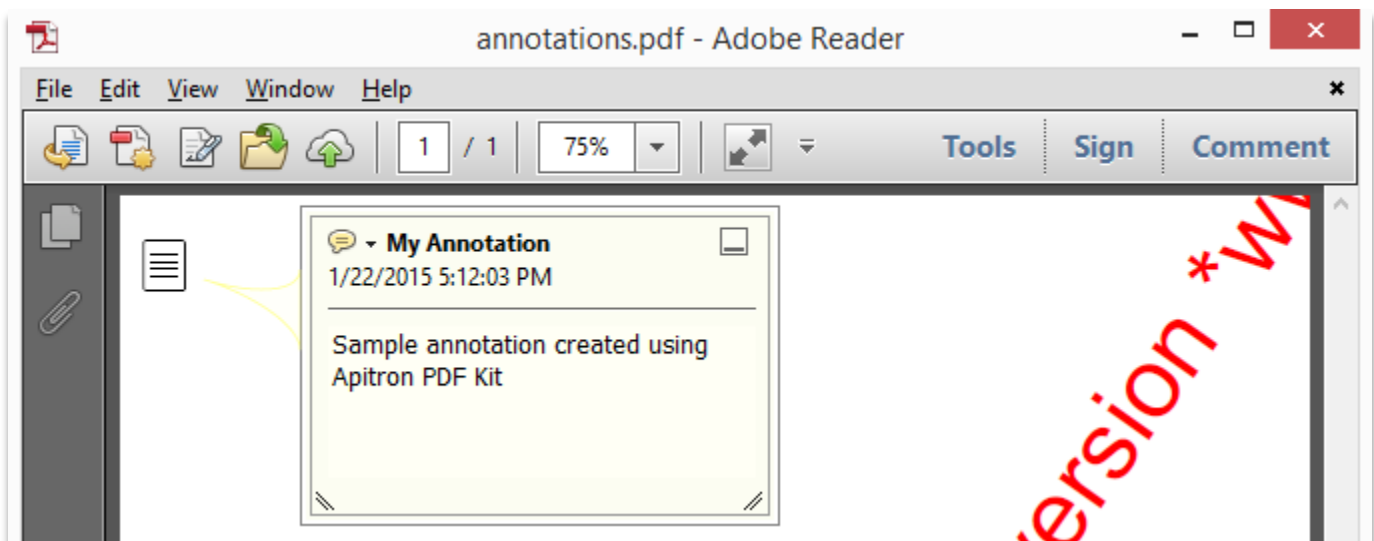
```
// create output PDF file
using (FileStream outputStream = new FileStream("annotations.pdf", FileMode.Create,
FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());

    // create text annotation attached to given point and set its properties
    TextAnnotation textAnnotation = new TextAnnotation(10,800);
    textAnnotation.IsOpen = true;
    textAnnotation.Title = "My Annotation";
    textAnnotation.Subject = "My Subject";
    textAnnotation.Contents = "Sample annotation created using Apitron PDF Kit";

    // add annotation to the PDF page
    document.Pages[0].Annotations.Add(textAnnotation);

    document.Save(outputStream);
}
```

The results are shown below:



Pic. 24 Add text annotation to the PDF page

3.5.2 Use link annotations for quick navigation

A special annotation called *Link Annotation* can be used to add a link pointing to any part of the PDF document or some external resource. When user interacts with this annotation a navigation event may occur. Such annotations can be invisible or have some visual appearance.

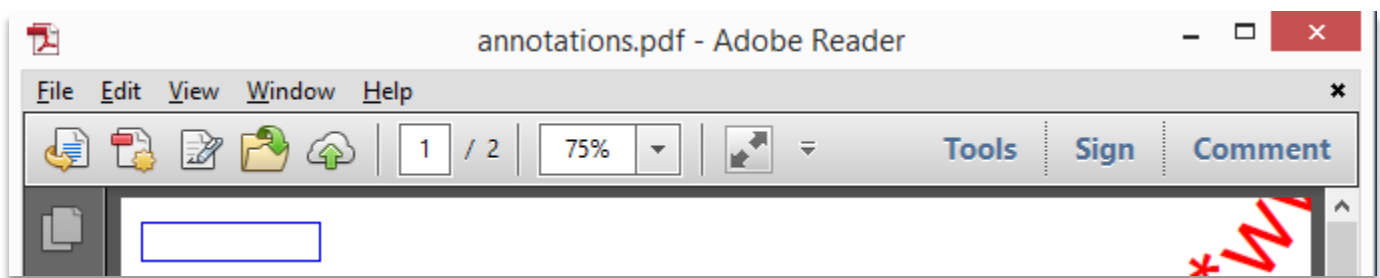
The following code shows how to create link annotation navigating to the second page of the created document. It uses object of type `Destination` to define its target.

```
// create output PDF file
using (FileStream outputStream = new FileStream("annotations.pdf", FileMode.Create,
FileStream.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    document.Pages.Add(new Page());
    // second page we are going to add a link to
    Page page2 = new Page();
    document.Pages.Add(page2);

    // create link annotation that points to second page
    LinkAnnotation linkAnnotation = new LinkAnnotation(new Boundary(10,810,100,830));
    linkAnnotation.Destination = new Destination(page2);
    linkAnnotation.BorderStyle = new AnnotationBorderStyle(1);
    linkAnnotation.Color = new double[] { 0, 0, 0.9 };

    // add annotation to the PDF page
    document.Pages[0].Annotations.Add(linkAnnotation);
    document.Save(outputStream);
}
```

Resulting link annotation shown on the image below will navigate the user to the second page when it's clicked:



Pic. 25 Creation of link annotation on PDF page

3.6 Actions

In addition to jumping to a destination in the document as shown in section [3.5.2 Use link annotations for quick navigation](#), an annotation or outline item may specify an action to perform, such as launching an application, playing a sound, changing an annotation appearance state. A variety of other circumstances may trigger an action as well (see section 12.6.3 “*Trigger Events*” of PDF specification). PDF includes a wide variety of standard action types, described in detail in section 12.6.4 “*Action Types*” of the PDF specification.

For the samples of most commonly used actions like *Go-To action*, *URI action*, *Named action* and *JavaScript action* see articles under this section.

3.6.1 Navigate to the parts of PDF document using Go-To action

A Go-To action changes the view to a specified destination (page, location, and magnification factor). Apitron PDF Kit provides you with a special class `GoToAction` for this.

Consider the following code creating link annotation and using `GoToAction` to perform navigation when it's clicked. This code looks almost identical to the code from chapter [3.5.2 Use link annotations for quick navigation](#) and produces the same results.

```
// create output PDF file
using (FileStream outputStream = new FileStream("actions.pdf", FileMode.Create, FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();

    document.Pages.Add(new Page());
    // second page we are going to add a link to
    Page page2 = new Page();
    document.Pages.Add(page2);

    // create link annotation that points to second page
    LinkAnnotation linkAnnotation = new LinkAnnotation(new Boundary(10, 810, 100, 830));
    // use Go-To action to perform navigation
    linkAnnotation.Action = new GoToAction(new Destination(page2));
    linkAnnotation.BorderStyle = new AnnotationBorderStyle(1);
    linkAnnotation.Color = new double[] { 0, 0, 0.9 };

    // add annotation to the PDF page
    document.Pages[0].Annotations.Add(linkAnnotation);
    document.Save(outputStream);
}
```

The main difference here is the use of the `Action` property instead of `Destination`.

3.6.2 Navigate using Remote Go-To action and Embedded Go-To action

It's also possible to navigate user to the destination in other PDF file by using `RemoteGoToAction`:

```
// use RemoteGoTo action to perform navigation to the second page of remote document
linkAnnotation.Action = new RemoteGoToAction(2, new FileSpecification("remotefile.pdf"));
```

Or to the destination in attached PDF file using `EmbeddedGoToAction`:

```
// use EmbeddedGoToAction action to perform navigation to the first page of attached document
linkAnnotation.Action = new EmbeddedGoToAction(new Destination(1), new
EmbeddedGoToActionTarget("attachment.pdf"));
```

You see that it can be achieved by doing a minor change in code for [3.6.1 Navigate to the parts of PDF document using Go-To action](#).

3.6.3 Navigate using Named actions

Named actions support several names defined by PDF specification which can be used to perform quick navigation. These names are *NextPage*, *PrevPage*, *FirstPage*, *LastPage*.

The code for such navigation would look as follows:

```
// use NamedAction to perform navigation to the next page
linkAnnotation.Action = new NamedAction(NamedActions.NextPage);
```

3.6.4 Navigate to specific URI using URI Actions

A uniform resource identifier (URI) is a string that identifies (resolves to) a resource on the Internet - typically a file that is the destination of a hypertext link, although it may also resolve to a query or other entity. If the `IsMap` property is true and the user has triggered the URI action by clicking an annotation, the coordinates of the mouse position at the time the action has been triggered will be included into the uri e.g. <http://www.apitron.com/?57,7>.

The code below shows how to add a link annotation navigating to a website:

```
// use URIAction action to perform navigation
linkAnnotation.Action = new URIAction(new Uri("http://www.apitron.com/product/pdf-kit"));
```

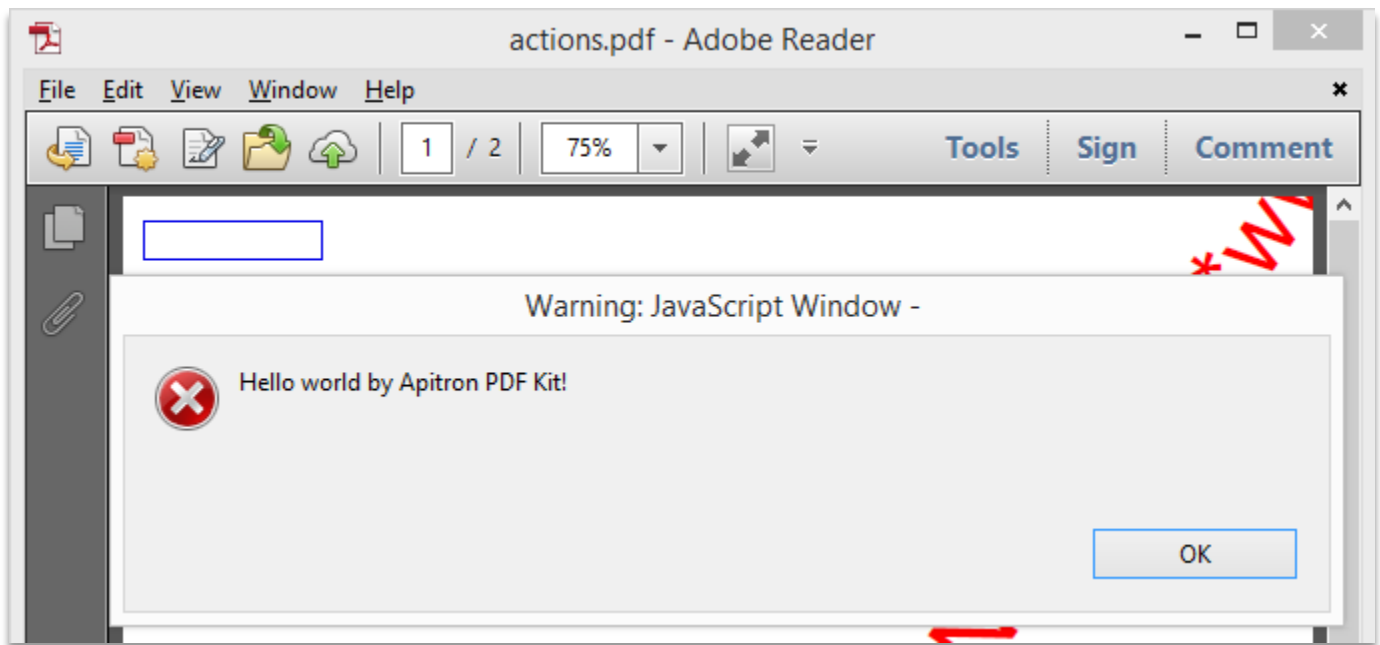
3.6.5 JavaScript actions

Upon invocation of a JavaScript action, a conforming processor shall execute a script that is written in the JavaScript programming language. Depending on the nature of the script, various interactive form fields in the document may update their values or change their visual appearances. *Mozilla Development Center's Client-Side JavaScript Reference* and the *Adobe JavaScript for Acrobat API Reference* give details on the contents and effects of JavaScript scripts.

The code that creates JavaScript action invoking simple message box is shown below:

```
// use JavaScript action to show message box  
linkAnnotation.Action = new JavaScriptAction("app.alert('Hello world by Apitron PDF Kit!');");
```

Results of invocation:



Pic. 26 Invoking JavaScript action

3.7 Interactive forms

An interactive form sometimes referred to as an *AcroForm* - is a collection of fields for gathering information interactively from the user. A PDF document may contain any number of fields appearing on any combination of pages, all of which make up a single, global interactive form spanning the entire document. Arbitrary subsets of these fields can be imported or exported from the document.

A PDF form can be seen as two separate parts - data model that use *document fields* to store information entered by user and its view presenting the data to the user and providing interaction.

If a field has visual representation on PDF page then it's provided by using special type of annotation object called *Widget Annotation*. There are widgets defined for common control elements e.g. button, checkbox, textbox etc. See section 12.7 "*Interactive Forms*" of the PDF specification for the detailed description of PDF forms.

It's possible to programmatically create PDF forms or edit existing forms using Apitron PDF Kit for .NET, see samples provided for this chapter.

3.7.1 Create, save and edit PDF form

The following example shows how to create a simple PDF form, fill it with default values, save to PDF file and then load and read its stored data.

```
public static void CreatePDFFormAndReadItsData()
{
    // create output PDF file
    using (FileStream outputStream = new FileStream("forms.pdf", FileMode.Create, FileAccess.Write))
    {
        FixedDocument document = new FixedDocument();
        document.Pages.Add(new Page());

        // create fields and add them into the document
        TextField txtName = new TextField("Name", "John Doe");
        TextField txtBirthDate = new TextField("birthDate", "12.05.1975");
        PushbuttonField btnChange = new PushbuttonField("btnChange", "Change");

        document.AcroForm.Fields.Add(txtName);
        document.AcroForm.Fields.Add(txtBirthDate);
        document.AcroForm.Fields.Add(btnChange);

        // create views for fields and set their properties
        TextFieldView nameView = new TextFieldView(txtName, new Boundary(10, 800, 100, 825));
        nameView.FontResourceID = "Arial";
        nameView.FontSize = 14;
        nameView.BorderStyle = new AnnotationBorderStyle(2, AnnotationBorderStyle.Inset);
        nameView.BorderColor = new double[] { 1 };

        TextFieldView birthDateView = new TextFieldView(txtBirthDate, new Boundary(10, 775, 100, 795));
        birthDateView.FontResourceID = "Courier";
        birthDateView.FontSize = 12;
        birthDateView.BorderStyle = new AnnotationBorderStyle(1, AnnotationBorderStyle.Inset);
        birthDateView.BorderColor = new double[] { 1 };

        PushbuttonFieldView buttonView = new PushbuttonFieldView(btnChange, new Boundary(10, 745, 50, 770));
        // add views to the PDF page
        AnnotationCollection pageAnnotations = document.Pages[0].Annotations;

        pageAnnotations.Add(nameView);
        pageAnnotations.Add(birthDateView);
        pageAnnotations.Add(buttonView);

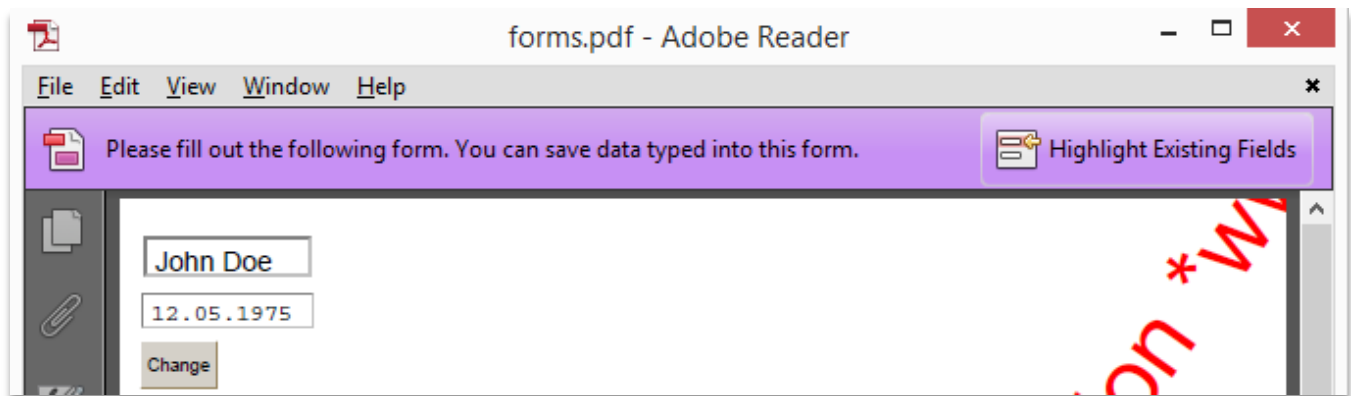
        document.Save(outputStream);
    }
    ...code continues on next page
}
```



```
// open file for reading
using (FileStream inputStream = new FileStream("forms.pdf", FileMode.Open, FileAccess.Read))
{
    FixedDocument document = new FixedDocument(inputStream);

    // read text fields
    foreach (TextField textField in document.AcroForm.Fields.OfType<TextField>())
    {
        Console.WriteLine("Field \"{0}\" has value: {1}", textField.FieldName, textField.Text);
    }
}
Console.ReadLine();
}
```

Resulting PDF file looks as follows:



Pic. 27 Create and save PDF form

Looking into the code we can see that fields storing form values are created first and then we create and assign views for them. After that we put these views on page using its annotations collection.

The results of reading this file using Apitron PDF Kit for .NET are shown on the image below:



Pic. 28 Read form fields from the PDF document

It's possible to change saved fields values and write the file back, thus making it possible to programmatically fill the desired PDF form.

3.7.2 Change form field value using JavaScript action

To remember what Actions are and how to work with JavaScript actions in particular, take a look at chapters [3.6 Actions](#) and [3.6.5 JavaScript actions](#) of this document.

What if we would like to use “Change” button from previous code sample (see [3.7.1 Create, save and edit PDF form](#)) for changing default value of the “Name” field belonging to a generated PDF form? A JavaScript action looks like an obvious choice and we could simply assign such action to this button and change field value if click occurs.

See the code:

...

```
// create button view
```

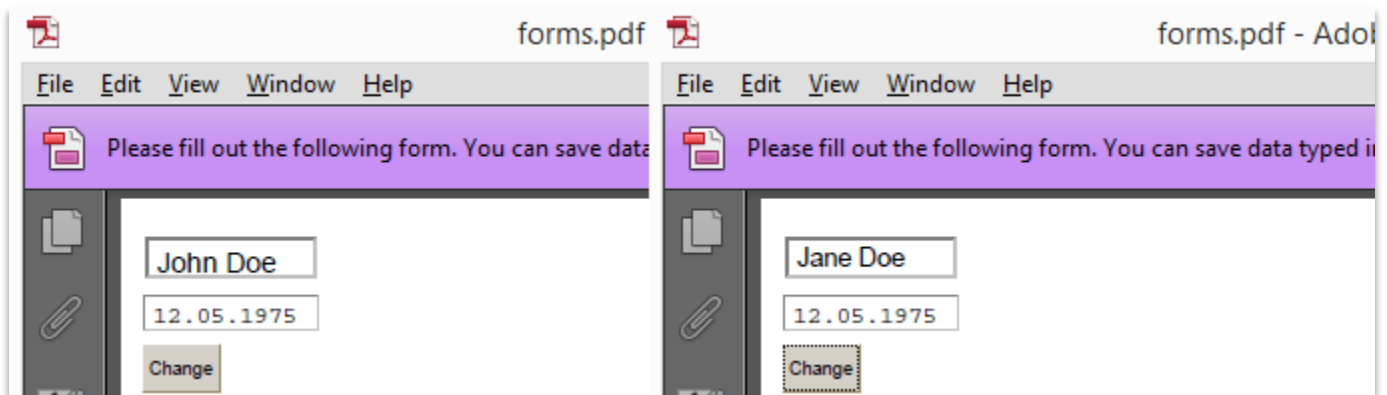
```
PushButtonFieldView buttonView = new PushbuttonFieldView(btnChange, new Boundary(10, 745, 50, 770));
```

```
// set JS action that changes text field value
```

```
buttonView.Action = new JavaScriptAction("this.getField('Name').value='Jane Doe';");
```

...

When user clicks on this button the value of field “Name” will be changed accordingly, resulting change is shown on the image below (John Doe -> Jane Doe):



Pic. 29 Change form field value using JavaScript action

3.8 Security

PDF specification allows user to set a password for PDF document, which can be used to limit access to the document protecting its content from unauthorized viewing and copying. It's possible to specify *access permissions* and set up to two passwords for a document: an *owner password* and a *user password*.

Opening the document with the correct owner password should allow full (owner) access to the document. This unlimited access includes the ability to change document passwords and access permissions.

Opening the document with the correct user password (or opening a document with the default password) should allow additional operations to be performed according to the user access permissions specified in the document encryption dictionary.

Complete list of access permissions can be found under section 7.6.3.2, “Table 22 – User access permissions” of the PDF specification. Corresponding enumeration type in Apitron PDF Kit is `Permissions`.

Owner password, user passwords and permissions can be set using `OwnerPassword`, `UserPassword` and `Permissions` properties of `SecuritySettings` object assigned to `FixedDocument` instance. See the property `FixedDocument.SecuritySettings`.

It's also possible to specify encryption algorithm to be used for document encryption using `FixedDocument.SecuritySettings.EncryptionLevel` property.

In order to have a standard way to validate document origin, PDF specification defines several methods for adding digital signatures to a document. It allows the viewer of the document quickly validate its integrity, make sure that document belongs to a particular source and wasn't changed after signing.

All these aspects will be discussed under this chapter, showing their practical application and usage.

3.8.1 Setting passwords and permissions for PDF document

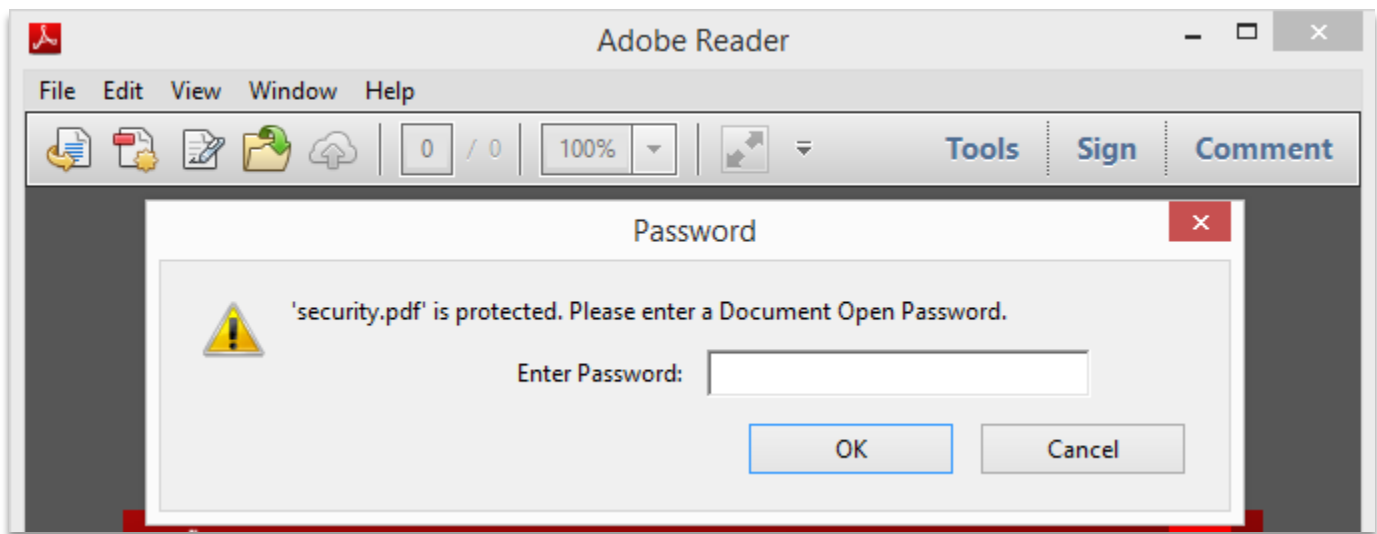
Setting passwords and permissions is easy and simple, see the code:

```
using (FileStream outputStream = new FileStream("security.pdf", FileMode.Create,
FileAccess.Write))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    // add empty page
    document.Pages.Add(new Page());

    // set both passwords and allow all permissions
    document.SecuritySettings = new SecuritySettings();
    document.SecuritySettings.OwnerPassword = "owner";
    document.SecuritySettings.UserPassword = "user";
    document.SecuritySettings.Permissions = Permissions.AllowAllPermissions;

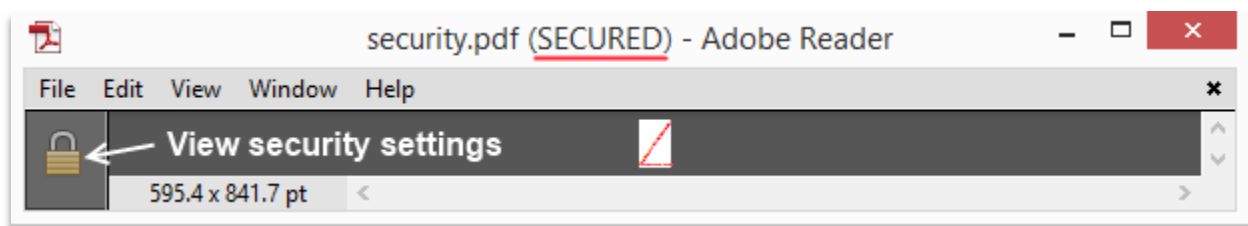
    document.Save(outputStream);
}
```

Once you try to open the resulting file, you'll see the following:



Pic. 30 Password request window

And once you enter the password, the file will be opened. You'll see the caption "SECURED" and be able to view file security settings by clicking the lock icon on the left:



Pic. 31 Opening PDF file with passwords and permissions

3.8.2 Reading password-protected PDF file

If we would like to open file created in previous example programmatically, we could use the code below to open and process it:

```
using (FileStream inputStream = new FileStream("password.pdf", FileMode.Open, FileAccess.Read))
{
    // open password protected PDF document
    FixedDocument document = new FixedDocument(inputStream, "owner");
    // write number of pages in PDF document
    Console.WriteLine("Document contains: {0} pages", document.Pages.Count);
}
```

We pass one of the document passwords (in this case it's owner password) to the FixedDocument constructor and it either opens the file if password is ok or throws an exception.

3.8.3 Add digital signature to the PDF document

Signatures are a complex subject explained in details in section 12.8 “*Digital Signatures*” of the PDF specification. Signature usually consists of a signature field added to document Fields collection and optionally its visual representation on one or more pages added using corresponding widget annotation object called *SignatureFieldView* (*Apitron specific class*). Fields, in general, are discussed in chapters [2.9 Working with fields](#) and [3.7 Interactive forms](#), annotations in chapter [3.5 Annotations](#). See the code and explanation below:

```
// create output PDF file
using (FileStream outputStream = new FileStream("security.pdf", FileMode.Create,
FileAccess.ReadWrite))
{
    // create new PDF document
    FixedDocument document = new FixedDocument();
    // add empty page
    document.Pages.Add(new Page());
    // register signature image
    document.ResourceManager.RegisterResource(new
Image("signatureImageJoe", "signatureImage.png"));

    // create signature field and add to the document
    SignatureField signatureField = new SignatureField("mySignature");
    document.AcroForm.Fields.Add(signatureField);

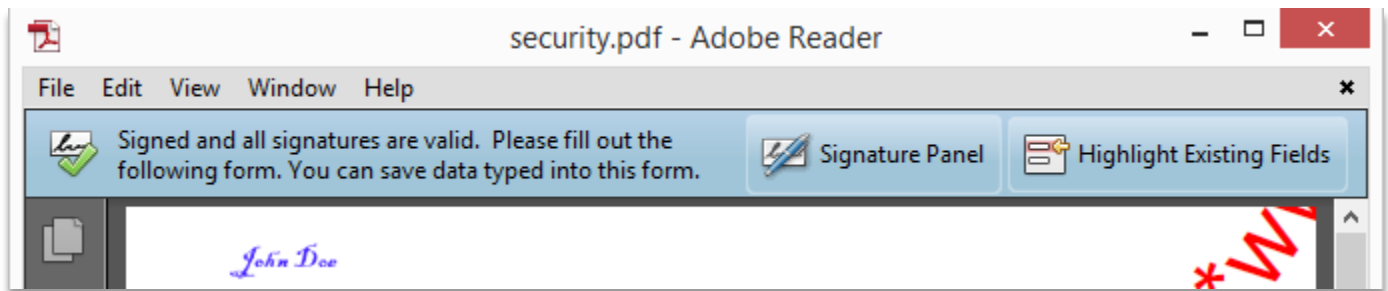
    // set signature certificate that will be used to sign the document
    using (Stream signatureStream = new FileStream("johndoe.pfx", FileMode.Open, FileAccess.Read))
    {
        signatureField.Signature = Signature.Create(new Pkcs12Store(signatureStream, "password"));
    }

    // create annotation object representing a signature
    SignatureFieldView signatureView = new SignatureFieldView(signatureField, new
Boundary(10,800,150,830));
    signatureView.ViewSettings = new SignatureFieldViewSettings()
    {
        Graphic = Graphic.Image,
        GraphicResourceID =
            "signatureImageJoe",
        Description = Description.None
    };

    // add visual representation of the signature onto PDF page
    document.Pages[0].Annotations.Add(signatureView);
    document.Save(outputStream);
}
```

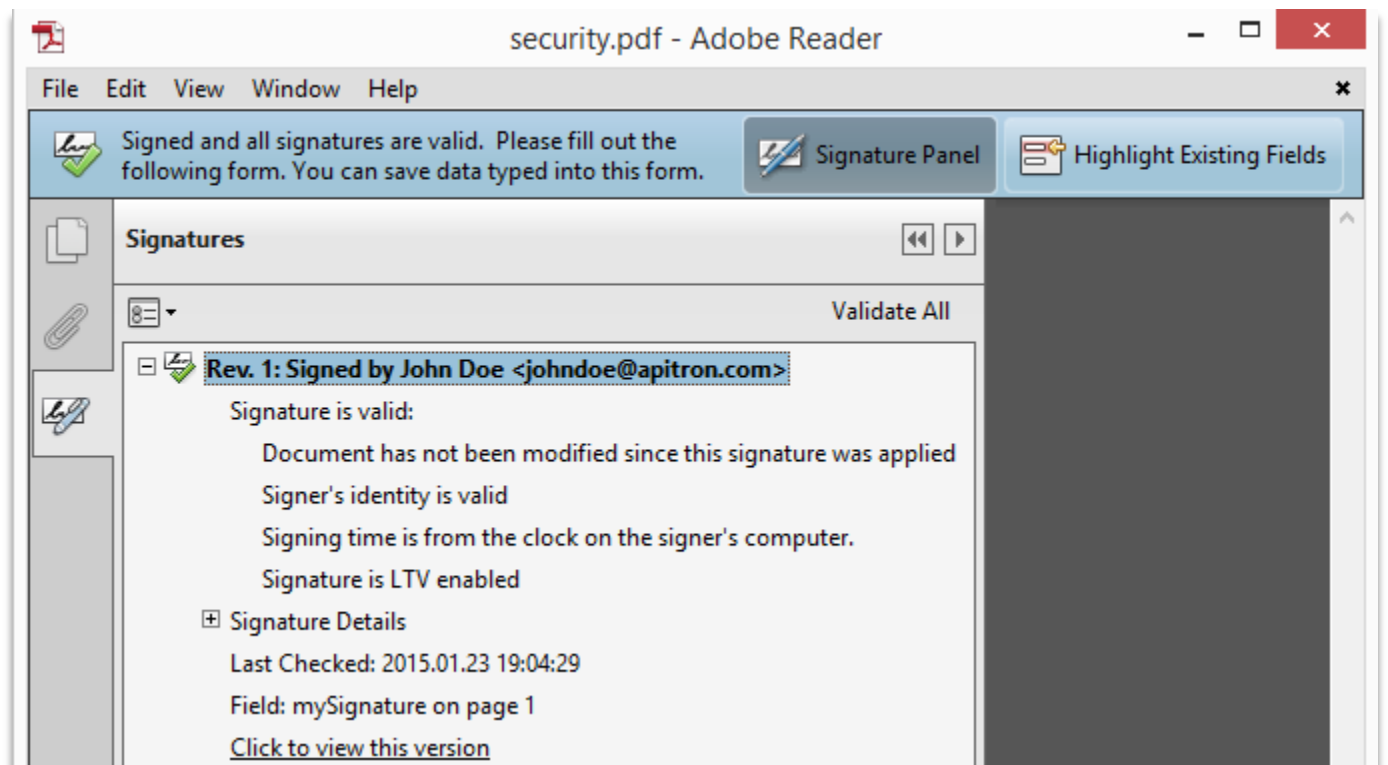
This code creates a signature object from a self-signed x509 certificate with a password set, and adds SHA1 RSA signature to the target document. It also creates its visual representation that looks as a handwritten signature, using an [image XObject](#). Any PDF document can contain several signatures and they all can be easily added using Apitron PDF Kit API. If you use a self-signed certificate and it's not added to the list of the trusted certificates then the reader may warn you that it can't validate the certificate.

Image below shows the output containing visual representation of the signature:



Pic. 32 Add digital signature to PDF page

Signature panel on the right shows signature properties:



Pic. 33 Signature panel shows signature properties

3.8.4 Validate digital signatures of the PDF document

The code below checks whether the document has changes added after signing. It doesn't validate the certificate itself because such algorithm is not defined in PDF specification and it's up to user to decide whether certificate can be trusted or not.

```
using (FileStream inputStream = new FileStream("security.pdf", FileMode.Open, FileAccess.Read))
{
    // open password protected PDF document
    FixedDocument document = new FixedDocument(inputStream);

    // check all signature fields
    foreach (SignatureField signatureField in document.AcroForm.Fields.OfType<SignatureField>())
    {
        Console.WriteLine("Signature found: {0}", signatureField.IsValid ? "Valid" : "Invalid");
    }
}
```

Output produced by this code using the PDF file generated from the previous sample.



Pic. 34 Validate digital signatures of the PDF document

3.9 Content extraction

It's possible to extract content from PDF document using API provided by Apitron PDF Kit. You may choose whether to use low-level API and process PDF commands yourself or use higher-level API designed for this task.

Generally, images and text can be extracted without much effort but it, of course, depends on the nature of PDF document and the way it was created. Not all PDF generation software follows PDF specification strictly and generates correct PDF documents.

3.9.1 Extract text from the PDF document

For the text extraction see chapter [2.1](#) from the *Document Level API* section.

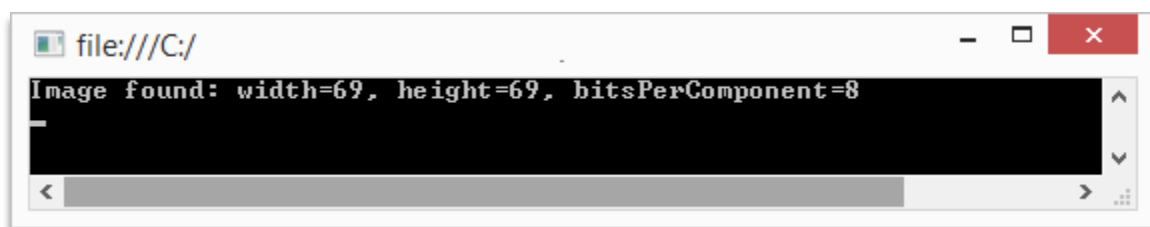
3.9.2 Extract images from the PDF document

Consider the PDF file created using the code from chapter [3.3.5.1 Drawing an image on PDF page](#). It adds an image to the PDF document and draws it twice on the first page.

The following code could be used to extract all images from PDF document:

```
using (FileStream inputStream = new FileStream("images.pdf", FileMode.Open, FileAccess.Read))
{
    // open document for reading
    FixedDocument document = new FixedDocument(inputStream);
    int imageIndex = 0;
    // enumerate through all pages and extract content
    foreach (Page page in document.Pages)
    {
        foreach (ImageInfo imageInfo in page.ExtractImages())
        {
            // dump image info
            Console.WriteLine("Image found: width={0}, height={1}, bitsPerComponent={2}",
imageInfo.Width,imageInfo.Height, imageInfo.BitsPerComponent);
            // save image to file
            using (FileStream fs = new FileStream(string.Format("image{0}.bmp",
imageIndex++),FileMode.Create, FileAccess.Write))
            {
                imageInfo.SaveToBitmap(fs);
            }
        }
    }
}
```

Its output is shown below; it prints image information and saves it as bitmap:



Pic. 35 Extract images from the PDF document

3.9.3 Extract attached files from PDF document

See the chapter [2.10 How to manage attachments in PDF document](#) for the tutorial on how to add and remove attachments. The code below enumerates the attachments in PDF document and saves them to files:

```
using (FileStream inputStream = new FileStream("attachments.pdf", FileMode.Open, FileAccess.Read))
{
    // open document for reading
    FixedDocument document = new FixedDocument(inputStream);

    // parse attachments and save to file
    foreach (var attachment in document.Attachments)
    {
        // save attachment to file
        using (FileStream fs = new FileStream(attachment.Key, FileMode.Create, FileAccess.Write))
        {
            attachment.Value.GetStream().CopyTo(fs);
        }
    }
}
```

3.10 Miscellaneous

Chapters below describe techniques and Fixed Layout API subsets aiming to simplify some of the tasks one may encounter during the PDF processing and which are not covered in other sections.

3.10.1 Create document from a set of images

There is sometimes a need to create a PDF document from a set of scanned receipts, photos, drawings or whatever things stored as images. While this task could be easily wrapped up by the code that registers a bunch of [image XObjects](#) and draws them on pages one by one, there is another way one could use for this task. `Page` class offers `CreateFromImage` static member function which creates a page object using a given image. It accepts a `FromImagePageSettings` object instance which can be used to define various page settings, e.g. orientation, padding, border etc.

See the sample code below:

```
// create output file
using (Stream outputStream = File.Create("from_images.pdf"))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument();

    // create a page from image
    fixedDocument.Pages.Add(Page.CreateFromImage("scan1.jpg",
        fixedDocument.ResourceManager, new FromImagePageSettings(Boundaries.Letter)
        {
            PageOrientation = PageOrientation.Portrait,
            ScaleMode = ImageScaleMode.PreserveAspectRatio
        }));

    // save document
    fixedDocument.Save(outputStream);
}
```

It creates PDF page from given image and sets its orientation and image scale mode. This functionality replicates the one previously released as separate *Image2Pdf* product and supersedes it. Results are given on the next page.

The image below demonstrates first page of the PDF file created using the code from previous page. Front page screenshot of www.apitron.com was used as an input.



Pic. 36 Create PDF page from image

3.10.2. Performing incremental updates

Section 7.5.6 “*Incremental Updates*” from PDF specification says:

The contents of a PDF file can be updated incrementally without rewriting the entire file. When updating a PDF file incrementally, changes shall be appended to the end of the file, leaving its original contents intact.

NOTE: The main advantage in updating a file this way is that small changes to a large document can be saved quickly. There are additional advantages: In certain contexts, such as when editing a document across HTTP connection or using OLE embedding (a Windows-specific technology), a conforming writer cannot overwrite the contents of the original file. Incremental updates may be used to save changes to documents in these contexts.

Apitron PDF Kit provides a way to perform incremental updates and write update data to the same stream the `FixedDocument` instance was created from (stream should support writing).

The following code shows how to update the existing PDF file using this technique:

```
// open file for updating
using (Stream inputStream = File.Open("file_to_update.pdf", FileMode.Open, FileAccess.ReadWrite))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument(inputStream);

    // create path, set color and fill it
    Path path = new Path();
    path.AppendRectangle(0,0,100,100);
    fixedDocument.Pages[0].Content.SetDeviceNonStrokingColor(new double[]{1,0,0});
    fixedDocument.Pages[0].Content.FillPath(path);

    // save changes as incremental update to the original document
    fixedDocument.Save();
}
```

Notice the `Save()` call without any arguments at the end of the sample. It performs an incremental update using the same stream file was created from. Resulting file will have a red rectangle drawn at its lower left corner.

4. Flow layout API

This API provides a new way to generate PDF files programmatically. In opposite to Fixed Layout API, designed to be as close as possible to PDF specification, Flow Layout API provides you with a number of high-level content building blocks e.g. *TextBlock*, *Grid*, *Image* which can be then placed on a page and automatically positioned using so-called *layout rules*. The layout engine analyzes various properties of these content elements and then logically processes them generating the resulting PDF document.

Properties of these content elements can be assigned directly or defined by *styles* thus it becomes possible to easily manage groups of objects with similar properties by changing corresponding style objects.

A process called *style matching* assigns a *style* to a particular object based on *style selector* – a rule that defines which elements should be affected by this style. It works similar to HTML+CSS and one familiar with the latter can easily use this experience and build PDF documents within a minute from the start. Actually styles and content element are so native and easy to work with that we would recommend using it for generation of quickly changing content, e.g. invoices, bills, reports, forms etc.

A list of supported features is rather big and we'd recommend checking code samples included along with the download package and showing the full power of Apitron PDF Kit and its flow layout API in action. To name a few, it supports floating text, automatic pagination of grids and sections, ordered and unordered lists with various types of markers (including nested lists), text justification, dynamic content generation, automatic links and bookmarks creation and much more. It can even be mixed with elements from [Fixed Layout API](#).

This API is built around `FlowDocument` class and it's possible to save created layout as XML template and load from it. It separates document design from actual processing and can be used to alter document appearance and introduce future changes without recompiling the application.

4.1. Create simple PDF file using Flow layout API

To quickly show you a generated file we will start our tour by creating a simple PDF document with “Hello world” message on the first page.

Consider the following code:

```
// create output file
using (Stream outputStream = File.Create("hello_world.pdf"))
{
    // create new document
    FlowDocument document = new FlowDocument();

    // add textual content element
    document.Add(new TextBlock("Hello world using flow layout API"));

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Here you can see the basics of flow layout workflow, we create `FlowDocument` instance first, then we fill it with some content (simple text block is used in this case) and at the end we write it out.

For writing we need an *output stream*, a `ResourceManager` instance that holds all resource objects used in the document (empty in our case) and `PageBoundary` object describing page dimensions. All pages in the target document will have the same dimensions unless they’re explicitly adjusted using techniques described in section [4.9.1 Change page size and styles on the fly](#).

As you can see, there is no notion of any style object or explicit property usage in this sample. It was made intentionally to give you a very quick overview and a working PDF file before moving on to more advanced topics. And the results of the execution look like this:

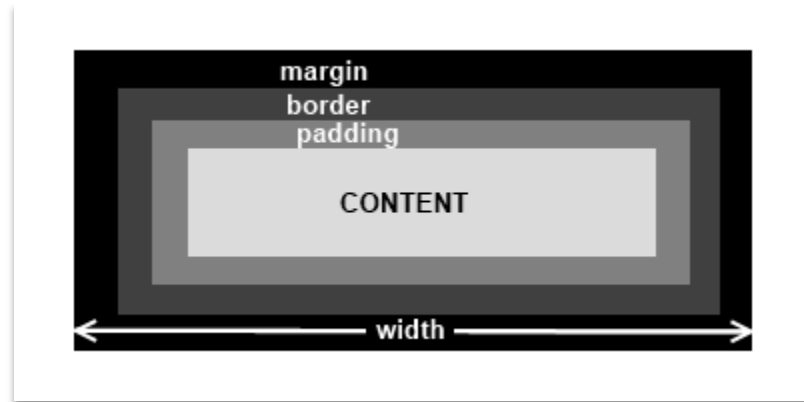


Pic. 37 Create simple PDF file using flow layout API

4.2 Box model and layout basics

Flow Layout API uses a slightly modified traditional box model to represent its content elements. Vertically infinite canvas limited by page width in horizontal dimension is used to lay them out and to form the content of the PDF document.

Flow layout box model assumes that margin, border and padding are parts of element width and height. See the image below:



Pic. 38 Box model of content elements

An element gets rendered within its box and later placed on a page according to layout rules and corresponding properties. It's important to note that every element will always have an implicit box even if you don't specify its dimensions. One of the important properties, affecting layout of the element, is *Display*. It specifies whether the element should be processed as *Block* - creating line break before and after, *Inline* - added to the current line, *InlineBlock* - special kind of Block element that doesn't create line breaks or *None* - thus shouldn't be processed.

The main difference between *Block*, *InlineBlock* and *Inline* elements is that pure inline elements can't be sized explicitly by setting their dimensions. It will be ignored much like in HTML. Also several style properties can be inherited only by Block elements (actually a very few of them) and it will be discussed later in section [4.3.2 Available styling properties](#).

Main type for specifying various dimensions of the elements in flow layout API is `Length`. Its constructor accepts values in points, pixels, centimeters, inches, % and special value `Auto`.

4.3 Styling system

Every content element or control element in flow layout API has the same set of properties affecting its appearance and behavior at the time of processing. This set is called a *style* and can be of two kinds:

- Internal object style, holding the properties you can set directly using object instance.
- Matched style, provided by the style manager of the document and assigned using specific *selector*.

A `Style` class works as properties container and `StyleManager` class as document style manager. `FlowDocument` class has special property called `StyleManager` and all necessary styles should be added using this property.

A style however doesn't come alone. It has a *selector*, an object that describes which elements should be affected by this style. In our case, the selector is a string written using special syntax that defines which elements to match.

If you ever worked with HTML and CSS then the syntax of these selectors will look quite familiar. Anyway, they're so easy to use that you'll get used to them very quick.

So, everything works as follows: you define styles and content elements that should be matched, add styles to the `StyleManager`, add content elements to the `FlowDocument` object and that's it. It saves lots of time, code, and adds great flexibility allowing creating rich and demanding content very quickly.

4.3.1 Selectors and style matching

Selector – is a rule that defines which elements should be styled using the given style, it consists of a rule that looks very similar to CSS notation and is bound to a property set defined by Style object.

Supported selectors:

- **Universal selector** (“*”), selects all elements in document, represented as *.
- **Type selector** (“_typename_”), selects all elements of the given type. E.g. “textbox”. It's possible to list several types using comma, e.g. "textbox, section, image".
- **Class selector** (“.classname”), selects all elements that belong to specific class. E.g. “*.myclass” selects all elements with class “myclass”. Because element can have multiple classes set, it is possible to specify several classes in selector, for example “*.myclass1.myclass2”.
- **Id selector** (“#elementID”). An element can have an id set, and this selector matches the element with specific id. Sample: “textbox#help”, selects the textbox element with ID “help”.
- **Child selector** (“[ancestor]>child”), selects all elements that are immediate children of ancestor element, square brackets indicate optionality of the ancestor, in case of absence it will behave as if “*” was set.

Samples:

“> textbox” – same as “textbox”, selects all textboxes which are children of any element.

“section > textbox ” – select all textboxes which are placed in sections

“flowdocument > section> textbox.subheader” – selects all textboxes with class “subheader” set, nested in any section that is an immediate child of root element.

- **Descendant selector** (“element_1 element_2”) selects all elements which are descendants of the given element.

Sample:

“section#header image” – selects all images in section with ID=“header” regardless of the inclusion level.

- **Adjacent element selector** (“predessor + element”), selects the element that precedes some element.

Sample:

“image + textbox” – selects all images that precede a textbox.

How the styles are being resolved:

1. Directly assigned value is requested first. It's the value you set using arbitrary property of the content element.
2. If no value is set using p.1, we are trying to find the value using selectors and styles registered.
 - a) look for all selectors that match given element
 - b) select value according to matching rules (these rules are: specificity of the selector and order of addition, if specificity is the same, we use order)
3. Request parent's value if property is inheritable using the same scheme

In general, this styling system is close to CSS used with HTML, so one familiar with the latter can easily find similarities.

4.3.2 Available styling properties

There are 36 different properties defined in `Style` class, plus some of the content elements may have their own specific properties which are not necessary related to styling. It provides great flexibility and gives lots of opportunities for generation of PDF documents.

Each property will be explained in subsequent sections to give you an overview of the styling system and possible effects, custom content elements properties will be described in corresponding sections under [4.4 Content elements and controls](#).

4.3.2.1 Layout-specific properties

Align - gets or sets the horizontal alignment of the element content, it can be Left, Right, Center aligned or Justified. Only acceptable for block elements, this property will be ignored if the element used is not a block. Left, Right, Center values are supported by all elements. In case of Justify is used – only text elements will be affected. All elements inherit value from their block containers; only block elements can override parent Align setting.

Clear - gets or sets the clear flag for the element indicating whether it should ignore floating elements (if any) and start a new line. Not inheritable.

Float - gets or sets the value indicating that element can float. Not inheritable.

Display - gets or sets the display setting for the element. Not inheritable.

Height - gets or sets the height of the element. Not inheritable.

Width - gets or sets the width of the element. Not inheritable.

LineHeight - gets or sets the height of the line. On a block container element whose content is composed of inline-level elements, 'line-height' specifies the minimal height of line boxes within the element. On a non-inline element, 'line-height' specifies the height that is used in the calculation of the line box height. Inheritable.

Margin - gets or sets the margin around the element. Not inheritable.

Padding - gets or sets the padding. Not inheritable.

VerticalAlign - gets or sets the vertical align. Not inheritable.

4.3.2.2 Text-specific properties

Color - gets or sets the foreground color for the element. Inheritable.

CharacterSpacing - gets or sets the character spacing. Inheritable.

Font - gets or sets the font for the element. Inheritable.

ScriptLevel - gets or sets the value used to create subscripting or superscripting effect. It defines the level of effect, zero can be used for normal scripting, positive values are for superscripting, negative for subscripting. Affects textual elements. Not inheritable.

TextDecoration - gets or sets the text decoration. Inheritable.

TextIndent - gets or sets the text indent. Inheritable.

TextRenderingMode - gets or sets the text rendering mode used to draw textual elements. By default all text is being drawn using Fill setting. Inheritable.

WordSpacing - gets or sets the word spacing. Inheritable.

4.3.2.3 Background-specific properties

Background - gets or sets the background color. Not inheritable.

BackgroundImage - gets or sets the background image for the element. The background area of an element takes the total space occupied by the element content, including padding (but not the margin and border). By default, a background-image is placed at the top-left corner of an element, and repeated both vertically and horizontally. Not inheritable.

BackgroundPosition - gets or sets the background position value for the element. The background position property sets the starting position of a background image. Not inheritable.

BackgroundRepeat - gets or sets the background repeat value for the element. The background repeat property sets if/how a background image will be repeated. By default, a background image is repeated both vertically and horizontally. Not inheritable.

4.3.2.4 Border-specific properties

Border - gets or sets the border around the element. Not inheritable.

BorderColor - gets or sets the color of the border. Not inheritable.

BorderBottom - gets or sets the bottom border. Not inheritable.

BorderLeft - gets or sets the left border. Not inheritable.

BorderRight - gets or sets the right border. Not inheritable.

BorderTop - gets or sets the top border. Not inheritable.

BorderRadius - gets or sets the border radius. Can be set using percentage value, in this case it's calculated using content width including padding as a base. If *BorderRadius* is being set to a non-null value, all other border values set except *Border* are ignored. Not inheritable.

4.3.2.5 Grid-specific properties

CellPadding - gets or sets the cell padding, affects only *Grid* elements. Not inheritable.

InnerBorder - gets or sets the inner border, affects only *Grid* elements. Not inheritable.

InnerBorderColor - gets or sets the color of the inner border affects only *Grid* elements. Not inheritable.

4.3.2.6 List specific properties

ListCounter - gets or sets the list counter. Not inheritable.

ListMarker - gets or sets the list marker appearance. Inheritable.

ListMarkerPadding - gets or sets the list marker padding. Inheritable.

ListStyle - gets or sets the list style. Not inheritable.

4.3.3 Use type selector to style a text block

Let's check the styling in action and define a simple style for all textblocks in flow document:

```
// create output file
using (Stream outputStream = File.Create("selectors.pdf"))
{
    // create new document
    FlowDocument document = new FlowDocument();

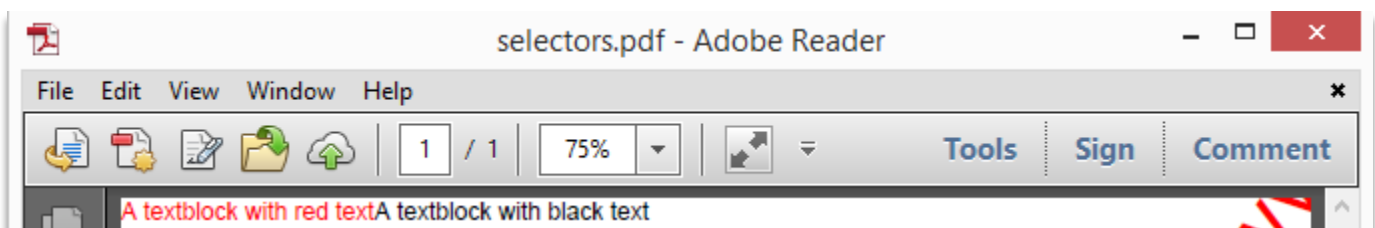
    // register style that matches all textblocks and sets their text color
    document.StyleManager.RegisterStyle("textblock", new Style(){Color = RgbColors.Red});

    // add textual content element
    document.Add(new TextBlock("A textblock with red text"));
    // add textual content element and directly set its property overriding the matched style
    document.Add(new TextBlock("A textblock with black text"){Color = RgbColors.Black});

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

You see from the code that we defined a type selector that matches all textblocks in the document. But according to Rule 1 of style matching (see [4.3.1 Selectors and style matching](#)) the second text box will have its color assigned to black, because any direct property assignment overrides property value provided by matching style.

The results are shown on the image below:



Pic. 39 Use type selector to style a text block

4.4 Content elements and controls

Content elements, as their name suggests, are building blocks for the flow document content. They all have the same set of properties but may react to them differently depending of their nature. All content elements are inherited from a single base class `ContentElement`.

The following content elements are supported at the moment:

- `Textblock` – for adding textual content.
- `Image` – for adding images.
- `Br` – indicates that line break should be created.
- `Hr` – creates a horizontal line much like in HTML.
- `Section` – container, for grouping elements together and creation of lists.
- `Grid` - for a creation of grids, support column and row spans.
- `PageCount` – a control that shows a number of pages in the document.
- `PageBreak` – indicates that a page break needs to be generated.
- `ContentReference` – used to place reusable pieces of content created with `FlowContent`, `FixedContent` or `Image` resource objects.

Control elements, or controls are special content elements designed to build forms. From the PDF point of view, they produce widget annotation objects (see chapter [3.7 Interactive forms](#)) and should have backing fields created for them. But you don't have to worry about how content elements or controls are being wrapped to PDF entities, because almost all PDF specific details are hidden and managed by Apitron PDF Kit library. The following control elements are supported at the moment:

- `TextBox` – used to create editable text fields that user can use for entering text.
- `PushButton` – a button that can have various Actions assigned (see [3.6 Actions](#)).
- `RadioButton` – a control providing a way to choose a single item from a group.
- `Choice` – works much like a combo box or a list box depending on its properties.
- `CheckBox` – provides a way to create various check boxes in PDF form.
- `SignatureControl` – can be used to provide a visual representation for a signature field in Flow Layout context, see also [3.8.3 Add digital signature to the PDF document](#) for an example explaining its under-the-hood mechanics using Fixed Layout API.

4.4.1 TextBlock

This element is designed to put textual content on page and has only one specific property `Text` for setting the text value. It's also possible to create dynamic text objects using a delegate, which will be able to use some information present at the time of execution for advanced text generation, e.g. current time or page number. Right to left and bidirectional text is supported automatically. Special characters supported are as follows: ' ,' (adds non-breaking space), '<,' (adds '<'), '>,' (adds '>'). By default this element appears as *Inline*.

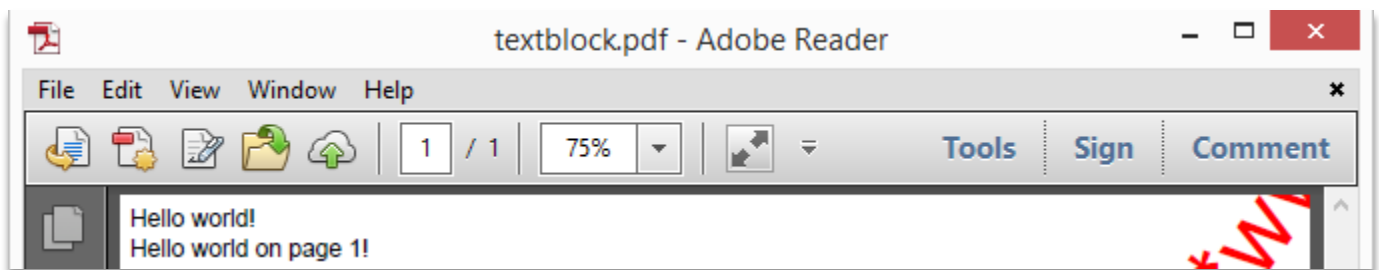
See the code:

```
using (Stream outputStream = File.Create("textblock.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument(){Margin = new Thickness(5);

    // create simple textblock
    TextBlock textBlock = new TextBlock("Hello world!");
    //create dynamic textblock and use PageGenerationContent for printing current page
    TextBlock textBlockDynamic = new TextBlock((ctx) => string.Format("Hello world on page {0}!",
ctx.CurrentPage+1));
    // add created text blocks
    document.Add(textBlock);
    document.Add(new Br());
    document.Add(textBlockDynamic);

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Produced results are below:



Pic. 40 Regular and dynamic TextBlocks

`TextBlock` uses one of the standard PDF fonts called Helvetica, size 10, as default font. One may use its `Font` property to change this behavior.

4.4.2 Image

This element represents an image on PDF page. You have to create an [image XObject](#) first and register it using document `ResourceManager`. After this you'll be able to put it on page using `Image` class from flow layout API. Its `ResourceId` property can be used to check what resource this image instance is bound to. By default this element appears as *Inline*.

Note: Despite the fact that Image content element is Inline by default, its dimensions can be set explicitly and these settings will affect resulting image size. It's an exclusion from the common rule which however seems logical for objects like image.

Consider the code below:

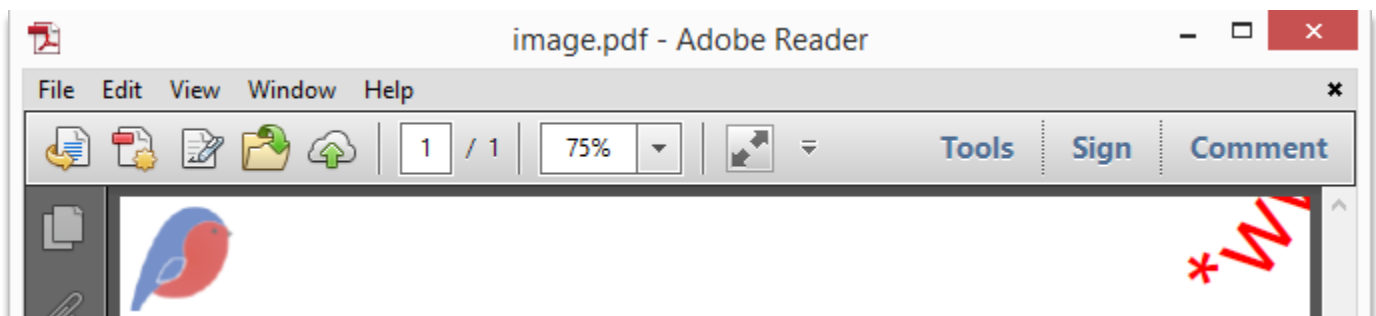
```
// create output file
using (Stream outputStream = File.Create("image.pdf"))
{
    // create resource manager that will hold our resources
    ResourceManager resourceManager = new ResourceManager();
    // add image XObject to resource manager
    resourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.Xobjects.Image("logo","apitron.png"));

    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // append image to document
    document.Add(new Image("logo"));

    // save to output stream with page size A4
    document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

Result looks as follows:



Pic. 41 Image content element added using flow layout API

4.4.3 Br and Hr

Content element `Br` represents a line break, default height of the element is `Opt.`, but it can be changed using its `Height` property.

Content element `Hr` represents a horizontal line, useful for visual formatting. Default thickness is `1pt.`, but it can be changed by using its `Height` property.

Both elements are always considered as *Block* elements.

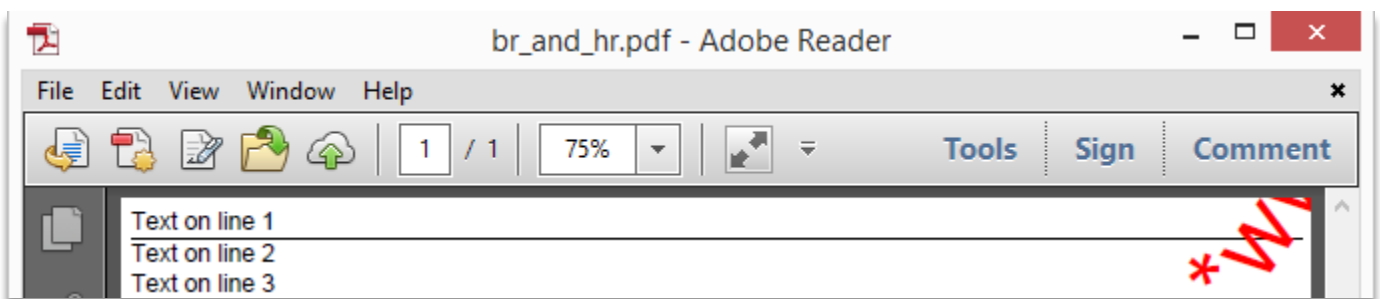
See the sample below:

```
// create output file
using (Stream outputStream = File.Create("br_and_hr.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // add content
    document.Add(new TextBlock("Text on line 1"));
    document.Add(new Hr());
    document.Add(new TextBlock("Text on line 2"));
    document.Add(new Br());
    document.Add(new TextBlock("Text on line 3"));

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Resulting image looks as follows:



Pic. 42 Br and Hr usage

`Br` element could be simulated by setting value of *Display* property for target element to *Block* in some cases. But if you'd like to create vertically spaced content lines with custom spacing between them, `Br` might be a better choice.

4.4.4 Section

`Section` is a container for grouping elements together. Can be styled and put in other containers and also can be used to create ordered and unordered lists by setting its `ListStyle` property via matching style or directly. By default this element appears as *Block*.

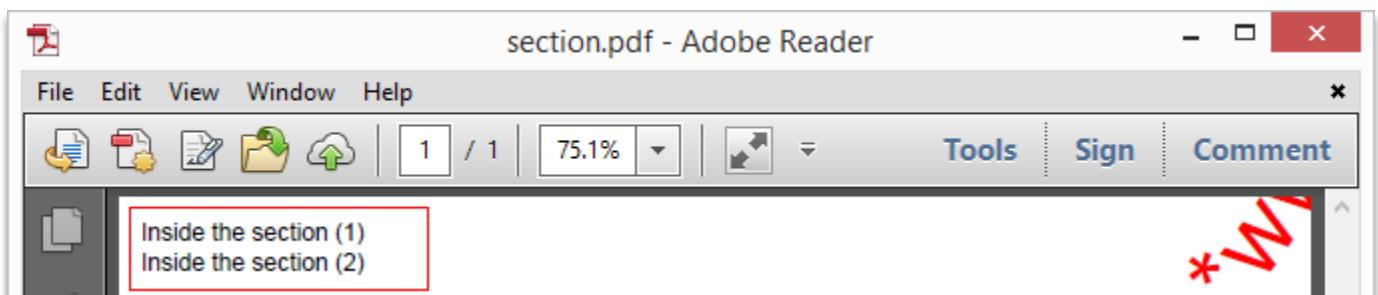
Consider the following example, creating a basic section with two `TextBlocks` inside:

```
// create output file
using (Stream outputStream = File.Create("section.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // create a section and set its properties directly
    Section section = new Section()
    {
        Border = new Border(1),
        BorderColor = RgbColors.Red,
        Padding = new Thickness(Length.FromPoints(5)),
        Width = 150
    };

    // add some content to section
    section.Add(new TextBlock("Inside the section (1)"));
    section.Add(new Br());
    section.Add(new TextBlock("Inside the section (2)"));
    // add 100heckbo to document
    document.Add(section);
    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Resulting image is shown below:



Pic. 42 Section content element

4.4.5 Grid

`Grid` content element can be used to group other elements in rows and columns. It's a *container* for `GridRow` elements which can be used to define grid rows. This element has special properties which can be set via matching `Style` or directly: `InnerBorder` for inner border definition (in addition to `Border`) and also `InnerBorderColor`. Additional property `CellPadding` can be used to define inner cell padding. By default this element appears as *Block*.

Let's try to create a basic grid showing imaginary shipping info for some company:

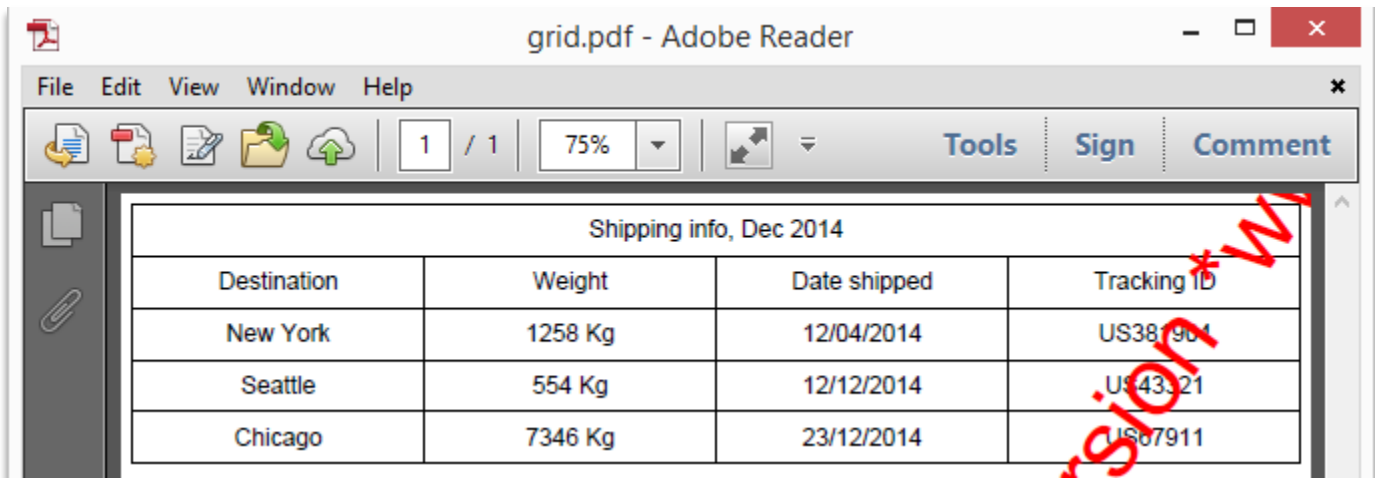
```
// create output file
using (Stream outputStream = File.Create("grid.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // create grid with 4 columns each taking 25% of page width
    // set cellpadding and content alignment for each cell
    Grid grid = new Grid(
        Length.FromPercentage(25),
        Length.FromPercentage(25),
        Length.FromPercentage(25),
        Length.FromPercentage(25))
        {CellPadding = new Thickness(5), Align = Align.Center};

    // fill the grid, notice the first row it uses colspan for making first cell wider
    grid.Add(new GridRow(new TextBlock("Shipping info, Dec 2014"){ColSpan = 4}));
    grid.Add(new GridRow(new TextBlock("Destination"),new TextBlock("Weight"),
new TextBlock("Date shipped"), new TextBlock("Tracking ID")));
    grid.Add(new GridRow(new TextBlock("New York"),new TextBlock("1258 Kg"), new
TextBlock("12/04/2014"), new TextBlock("US381904")));
    grid.Add(new GridRow(new TextBlock("Seattle"),new TextBlock("554 Kg"), new
TextBlock("12/12/2014"), new TextBlock("US43321")));
    grid.Add(new GridRow(new TextBlock("Chicago"),new TextBlock("7346 Kg"), new
TextBlock("23/12/2014"), new TextBlock("US67911")));

    // add to document
    document.Add(grid);
    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

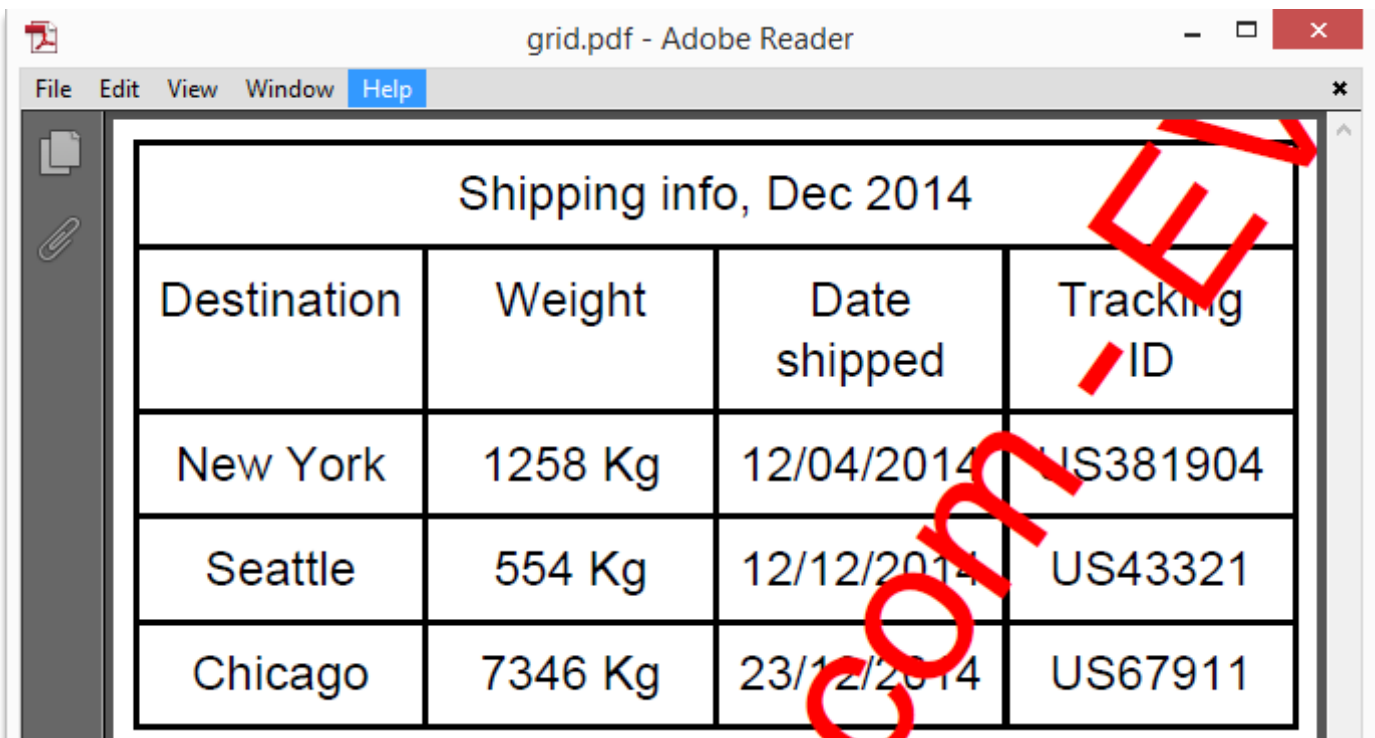
The resulting file looks as follows:



Shipping info, Dec 2014			
Destination	Weight	Date shipped	Tracking ID
New York	1258 Kg	12/04/2014	US381904
Seattle	554 Kg	12/12/2014	US43321
Chicago	7346 Kg	23/12/2014	US67911

Pic. 43 Grid content element usage, page size A4

Easy and clear, notice that we used *ColSpan* to make a top row span across the grid. We also used percentage-based column widths, so our grid will look similar even if we change page size. Let's do it and set it to A6.



Shipping info, Dec 2014			
Destination	Weight	Date shipped	Tracking ID
New York	1258 Kg	12/04/2014	US381904
Seattle	554 Kg	12/12/2014	US43321
Chicago	7346 Kg	23/12/2014	US67911

Pic. 44 Grid content element usage, page size A6

Each row of the grid is represented by the type `GridRow`, a subcontainer element that is being used to group cells together. It can only be added into `Grid` element and its styling is limited (only properties which can be inherited from plus background color value affect the appearance of the contained grid cells).

Links created using `CrossReference` or `LinkUri` and pointing to/from `GridRow` elements are also not supported. This behavior can be easily workarounded by adding a link to inner elements.

Each content element added into grid row is being treated as a separate cell, so if you need multiple elements to be placed inside the single cell you should use `Section` to group them together. If you need the cell to be spanned vertically or horizontally you may use `RowSpan` or `ColSpan` properties respectively, these properties are taken into account by `Grid` element only. Both are set to 1 by default.

Note: Every document has default style set for grid elements that uses single line `InnerBorder` and `Black` as default value for `InnerBorderColor`. This style is defined using a type selector (see [4.3.3 Use type selector to style a text block](#) as an example) and matches all grid elements in particular document. If you override this default style you will have to provide values for properties mentioned above, otherwise their values won't be set providing possibly unexpected results.

4.4.6 PageBreak

`PageBreak` content element controls pagination behavior. Works on a page-level, has no effect being added to a container other than `FlowDocument`. If the page has available space then current page will be finished and generation will continue on next page, otherwise blank page will be added right after and generation will continue starting from new page created thereafter.

Code for adding a page break:

```
// add page break
document.Add(new PageBreak());
```


4.4.7 PageCount

`PageCount` content element represents an element that shows number of pages generated for the document while saving. It can be used on any page, so you may use it to show number of pages even on the first page of the document. By default this element appears as *Inline*.

Consider the sample code:

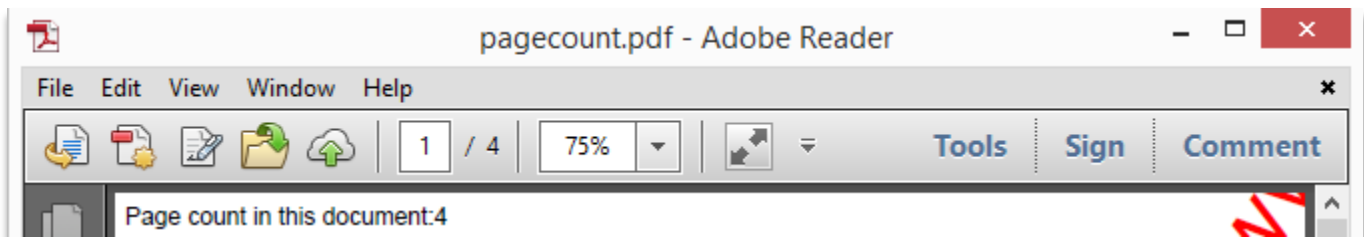
```
// create output file
using (Stream outputStream = File.Create("pagecount.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    document.Add(new TextBlock("Page count in this document:"));
    // add page count element, set counter resolution to 2
    document.Add(new PageCount(2));

    // add page breaks, it will create blank pages
    document.Add(new PageBreak());
    document.Add(new PageBreak());
    document.Add(new PageBreak());
    document.Add(new PageBreak());

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Generated file looks as follows:



Pic. 45 PageCount content element usage

You may have noticed that we created our `PageCount` element passing '2' as constructor parameter. It was done to make sure we would have enough space to fit 2 digits representing pages count as this parameter defines counter resolution.

4.4.8 ContentReference

It's a special content element that can be used to put reusable pieces of content on PDF page. Such content should be defined using `FlowContent`, `FixedContent` or `Image` classes and registered in document `ResourceManager`. By default this element appears as *Inline*.

Let's see the code:

```
// create output file
using (Stream outputStream = File.Create("contentreference.pdf"))
{
    // create document resource manager
    ResourceManager resourceManager = new ResourceManager();

    /* create form XObject (1) */
    FixedContent fixedContent = new FixedContent("rect", new Boundary(0,0,50,50));

    // create rect path
    Path path = new Path();
    path.AppendRectangle(10,10,30,30);
    // select colors and draw path
    fixedContent.Content.SetDeviceNonStrokingColor(new double[]{1,0,0});
    fixedContent.Content.SetDeviceStrokingColor(new double[] {0,0.5,1});
    fixedContent.Content.FillAndStrokePath(path);
    // register fixed content
    resourceManager.RegisterResource(fixedContent);

    /* create a piece of flow content (2) */
    Section flowContentSection = new Section() { Color = RgbColors.Black};
    flowContentSection.Add(new TextBlock("ContentReference element can be used"));
    flowContentSection.Add(new Br());
    flowContentSection.Add(new TextBlock("to place reusable pieces of content on PDF page"));

    FlowContent flowContent = new FlowContent("flowContentPiece", flowContentSection, 150, 45);
    resourceManager.RegisterResource(flowContent);

    /* create and register image resource (3) */
    resourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.Xobjects.Image("logo", "apitron.png"));

    // create new flow document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // register style for ContentReference elements using type selector
    document.StyleManager.RegisterStyle("ContentReference", new Style() { BorderColor =
    RgbColors.Magenta, Border = new Border(1) });
}
```

```
// place reusable content on page
document.Add(new ContentReference("rect"));
document.Add(new ContentReference("logo"));
document.Add(new ContentReference("flowContentPiece"));
document.Add(new ContentReference("flowContentPiece"));
document.Add(new ContentReference("rect"));
document.Add(new ContentReference("logo"));

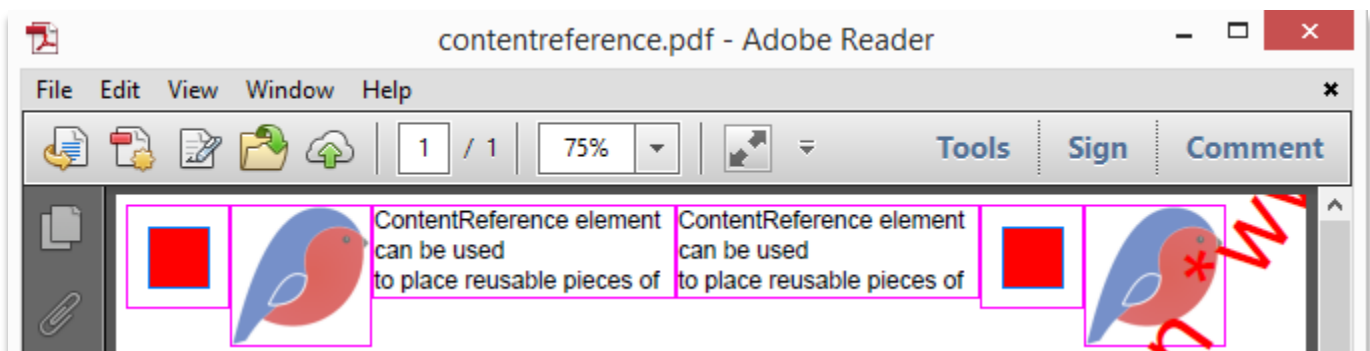
// save to output stream with page size A4
document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

In this example we use all kinds of objects which can be referenced by the ContentReference element:

- `FixedContent`, a form `XObject` described in chapter [3.3.3 Form Xobjects](#). We simply draw a rectangle using Fixed Layout API. (1)
- `FlowContent`, a static piece of content that can be created and reused multiple times using flow layout rules in opposite to `FixedContent`. Its bounds should be specified on creation and one or more elements (if `Section` is used) can be used to define its content. (2)
- `Image`, an `image XObject` object defining image to be used within PDF document and described in chapter [3.3.5 Images](#). (3)

Furthermore, all elements have the same border defined by a matching style. It means that even if all these elements represent static content, their placeholder can still be customized.

An image below shows the result:



Pic. 46 ContentReference usage

4.4.9 TextBox

Represents a `TextBox` control linked to a `TextField`. Under the hood it hosts widget annotation representing field content. It can be multiline and have various borders and styles assigned. By default this element appears as *Block*.

Consider the code below, which creates two text boxes styled in different ways:

```
// create output file
using (Stream outputStream = File.Create("textbox.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    document.StyleManager.RegisterStyle("TextBox", new Style() { Width = 100 });

    // add text field and set its properties
    TextField textField1 = new TextField("textField1", "textbox")
    {
        BorderStyle = new AnnotationBorderStyle(1),
        BorderColor = new double[] { 0 }
    };

    // add text field without setting any styles
    TextField textField2 = new TextField("textField2", "textbox");

    // add fields to flow document
    document.Fields.Add(textField1);
    document.Fields.Add(textField2);

    // add text box linked to first text field
    document.Add(new TextBox("textField1"));

    // add text box linked to second text field and style right here
    document.Add(new TextBox("textField2")
    {
        Border = new Border(1, BorderColor = RgbColors.Black, BorderRadius = 5
    });

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

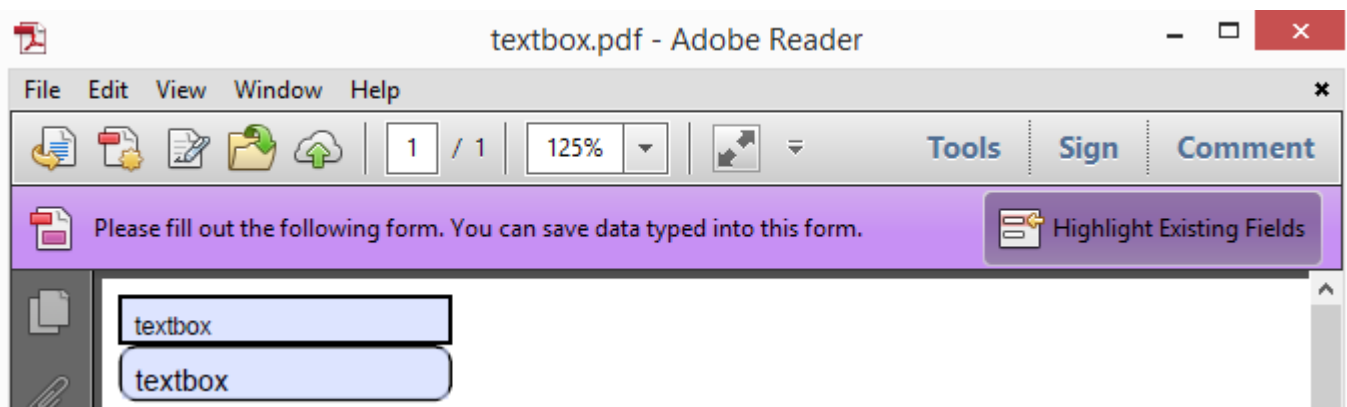
As one may notice from the code above, we define the width for both text boxes using a style bound to a type selector. Other important difference is that we style first text box using properties of a linked field while the second is being styled *in situ* by setting corresponding content element properties.

Thus, in first case, we set up actual field and all text boxes bound to this field (and there can be many) will inherit these properties as default for their widget annotation. All fields properties are defined in PDF specification.

In second case we style a particular text box, and actually we style it via its placeholder. Since PDF doesn't define properties like border radius, etc. which are supported by Apitron PDF kit, the placeholder gets styled and not the annotation itself.

It's an important difference that one should consider if he or she ever wants to copy annotations from one document to another.

An image below demonstrates the result:



Pic. 47 TextBox usage

Highlight Existing Fields viewing mode was used to select form fields in demonstrated document.

4.4.10 PushButton

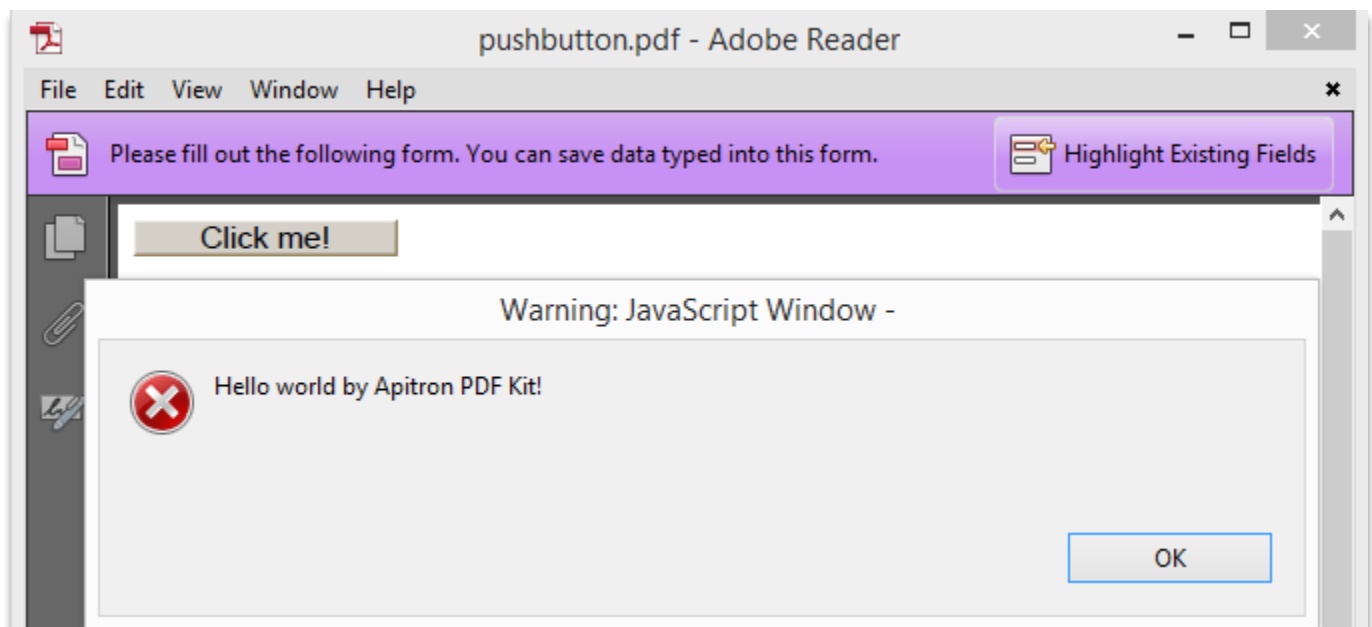
`PushButton` element represents a button control linked to a `PushButtonField`. Any action defined in PDF specification can be assigned to it. See [3.6 Actions](#) for the details. Sample below demonstrates creation of the PDF document containing a button with `JavaScriptAction` assigned. By default this element appears as *Block*.

```
// create output file
using (Stream outputStream = File.Create("pushbutton.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    document.Fields.Add(new PushbuttonField("buttonField", "Click me!"));
    PushButton pushButton = new PushButton("buttonField",
        new JavaScriptAction("app.alert('Hello world by Apitron PDF Kit!');"))
    {
        Width = 100
    };
    // add new push button that shows message box when clicked
    document.Add(pushButton);

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Result looks as follows:



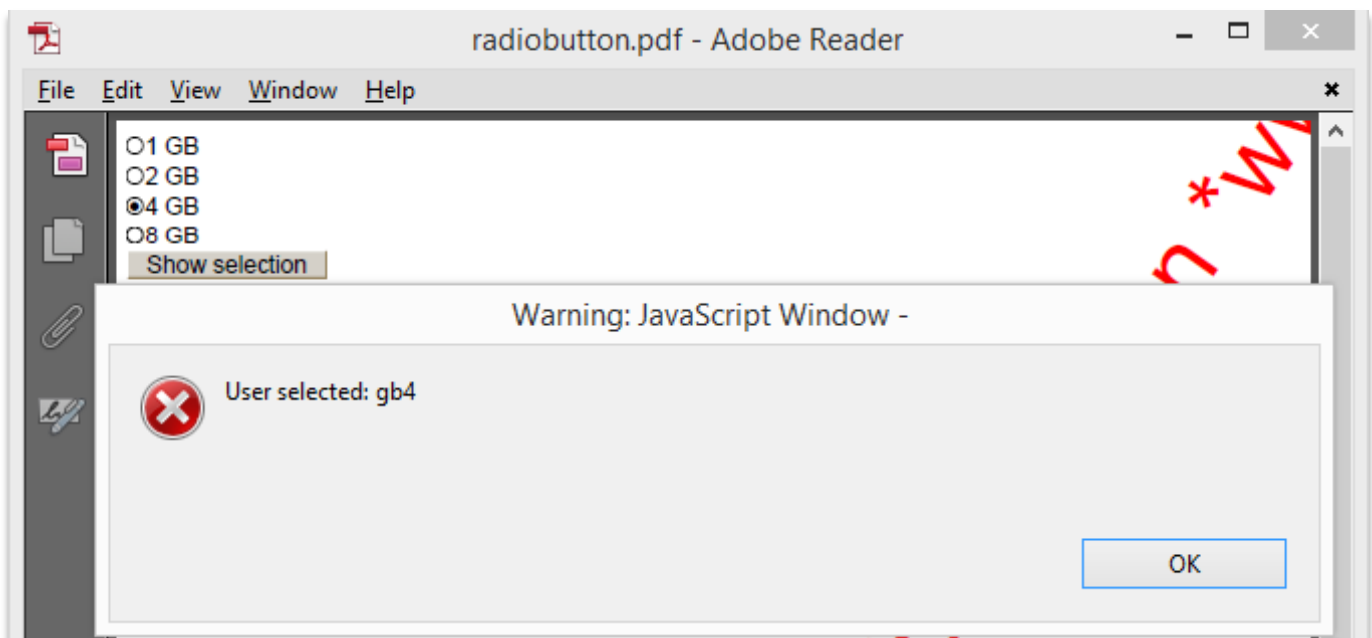
Pic. 48 PushButton usage

4.4.11 RadioButton

`RadioButton` element represents a radio button control linked to a `RadioButtonField`. You can use single field to group radio buttons together. By default this element appears as *Block*. Consider the code below:

```
using (Stream outputStream = File.Create("radiobutton.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // set all elements to have the same width
    document.StyleManager.RegisterStyle("*", new Style(){Width = 100});
    // add field and link option buttons
    document.Fields.Add(new RadioButtonField("requiredRam", "8gb"));
    document.Add(new RadioButton("requiredRam", "1 GB", "gb1"));
    document.Add(new RadioButton("requiredRam", "2 GB", "gb2"));
    document.Add(new RadioButton("requiredRam", "4 GB", "gb4"));
    document.Add(new RadioButton("requiredRam", "8 GB", "gb8"));
    // add button field and PushButton
    document.Fields.Add(new PushbuttonField("showSelection", "Show selection"));
    document.Add(new PushButton("showSelection",
        new JavaScriptAction("app.alert('User selected: ' + this.getField('requiredRam').value);")));
    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Results are shown on image below, note that we used `PushButton` to display selected value. It's also possible to define different check marks instead of a circle used by default.



Pic. 49 RadioButton usage

4.4.12 Choice (combobox and listbox)

Choice element is an element linked to `ChoiceField` which contains several text items, one or more of which shall be selected as the field value. The items may be presented to the user in one of the following two forms:

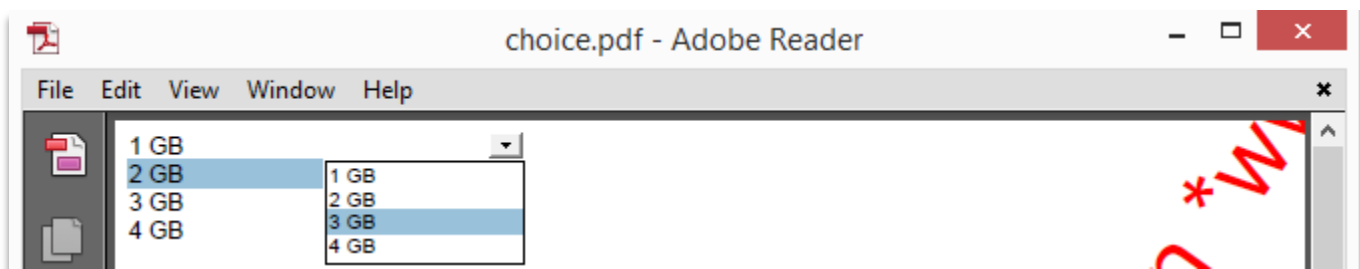
- A combo box consisting of a drop-down list. The combo box may be accompanied by an editable text box in which the user can type a value other than the predefined choices if it's allowed by the field author.
- A scrollable list box

By default this element appears as *Block*. Consider the code below:

```
using (Stream outputStream = File.Create("choice.pdf"))
{
    string [] options = new string[]{"1 GB", "2 GB", "3 GB", "4 GB"};
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // set all elements to have the same width
    document.StyleManager.RegisterStyle("*", new Style() { Width = 100 });
    // add choice as list box
    document.Fields.Add(new ChoiceField("requiredRam", "1gb", options){BorderColor = new
double[0]});
    // add choice as combo
    document.Fields.Add(new ChoiceField("requiredRam2", "1gb", options,
ChoiceFieldType.ComboBox));
    // add Choice element to show first choice field
    document.Add(new Choice("requiredRam") { Display = Display.InlineBlock });
    // add Choice element to show second choice field
    document.Add(new Choice("requiredRam2") );

    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

Resulting document is shown below:



Pic. 50 Choice usage

4.4.13 Checkbox

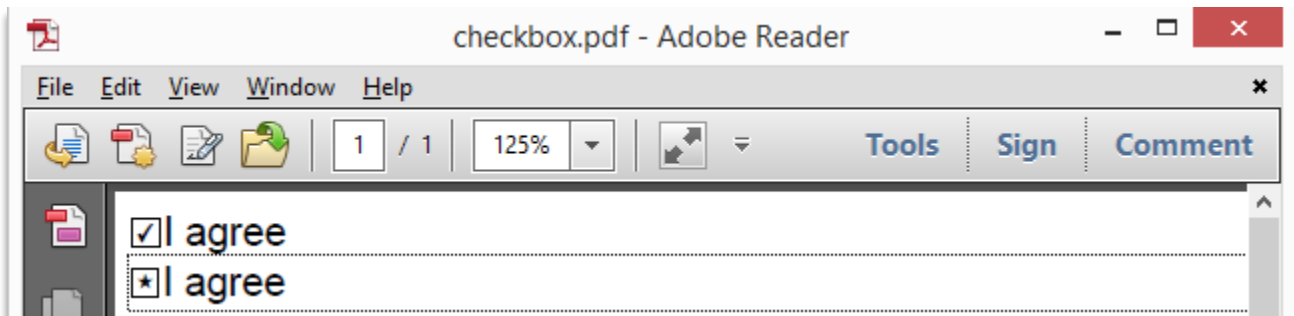
`Checkbox` element represents a checkbox control linked to a `CheckBoxField`. It supports various checkmarks and many checkboxes can be linked to a single field. By default this element appears as *Block*.

See the code:

```
// create output file
using (Stream outputStream = File.Create("checkbox.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // add checkbox field
    document.Fields.Add(new CheckBoxField("112checkbox", "I agree"));
    // add two checkbox controls linked to single field
    document.Add(new CheckBox("112checkbox", CheckBoxMark.Check));
    document.Add(new CheckBox("112checkbox", CheckBoxMark.Star));

    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

An image below demonstrates the resulting document:



Pic. 51 Checkbox usage

This document contains two checkboxes linked to a single `CheckBoxField` and they have different check marks set. When user selects one of them the other becomes checked as well.

4.4.14 SignatureControl

This element defines visual representation for `SignatureField` in flow document context. It's possible to use `SignatureFieldViewSettings` class to define view settings which widget annotation created for signature will use. You may assign it using `ViewSettings` property of the `SignatureField`. If no view settings are assigned then `SignatureControl` will show default signature representation which includes all signature properties related to signer. These properties will be loaded from certificate used for signing. By default this element appears as *Block*.

See also chapter [3.8.3 Add digital signature to the PDF document](#) from Fixed Layout API section for other details and sample demonstrating how to use image `XObject` instead of default visual representation.

The code for adding visual representation for a digital signature looks as follows:

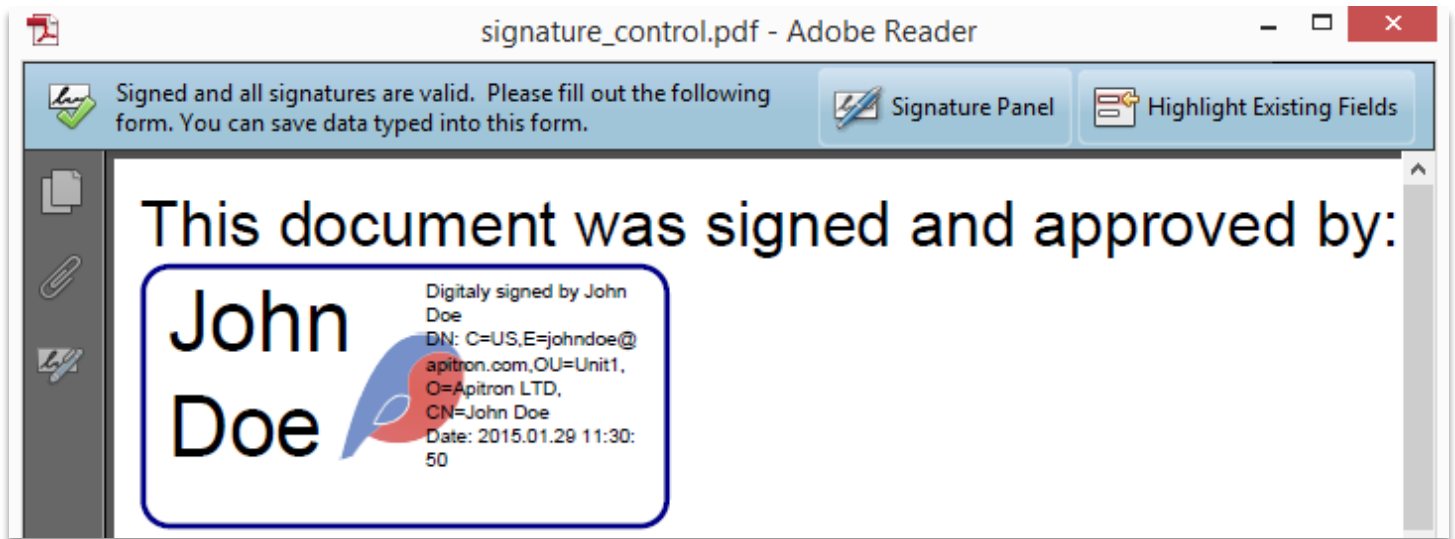
```
// create output file
using (Stream outputStream = File.Create("signature_control.pdf"))
{
    ResourceManager resourceManager = new ResourceManager();
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // add style for signature control using type selector
    document.StyleManager.RegisterStyle("SignatureControl", new Style() { Border = new Border(1),
    BorderColor = RgbColors.DarkBlue, BorderRadius = 5 });

    // open digital certificate and create signature field
    using (Stream certificateStream = File.OpenRead("johndoe.pfx"))
    {
        // add lender signature field
        document.Fields.Add(new SignatureField("signature")
        {
            Signature = Signature.Create(new Pkcs12Store(certificateStream, "password"))
        });
    }
    // add textblock and signature control to resulting document
    document.Add(new TextBlock("This document was signed and approved by:"));
    document.Add(new SignatureControl("signature") { Width = 100, Height = 50 });

    // save to output stream with page size A4
    document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

Note that unlike sample from fixed layout we didn't set an image representing "handwritten" signature in this example. We relied on default visual representation generated by Apitron PDF Kit for our signature field. Another thing to note is that we used style and matching type selector to create a custom border around the signature control.

Resulting file is shown on the image below:



Pic. 52 SignatureControl usage (scale 200%)

4.4.15 Page header and footer

Page header and footer are designed to host content which should be repeated on top or bottom of each page, e.g. page number or company title. There are no special content elements defined for header or footer and they both are of type [Section](#). One may use `PageHeader` or `PageFooter` properties of `FlowDocument` object instance to set their content.

The code below shows how to work with these elements:

```
// create output file
using (Stream outputStream = File.Create("header_and_footer.pdf"))
{
    // create document and set margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // register style for gray elements
    document.StyleManager.RegisterStyle(".grayBg", new Style() { Background = RgbColors.LightGray });
    // register styles for header and footer
    document.StyleManager.RegisterStyle(".header", new Style() { Align = Align.Center, VerticalAlign =
VerticalAlign.Middle });
    document.StyleManager.RegisterStyle(".footer", new Style() { Align = Align.Right, VerticalAlign =
VerticalAlign.Bottom });

    // set header class and add content
    document.PageHeader.Class = "header grayBg";
    document.PageHeader.Add(new TextBlock("Big Company Inc.));

    // add document's content
    document.Add(new TextBlock("This document shows page header and footer usage. Both elements
were styled using separate styles and class selectors. Common class selector was used to make them
appear gray.));
    document.Add(new PageBreak());
    document.Add(new TextBlock("Notice that footer shows total page count.));

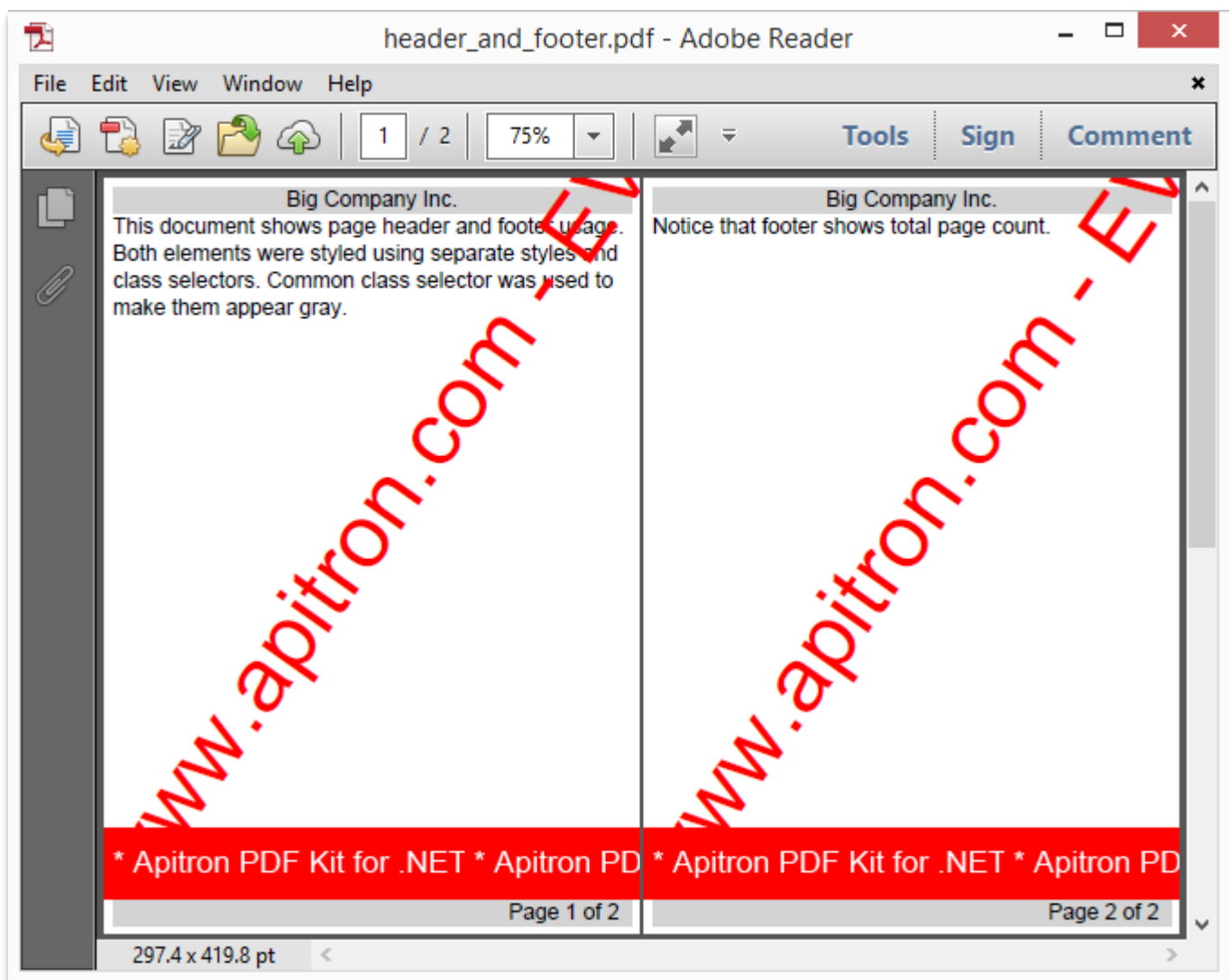
    // set footer class and add content
    document.PageFooter.Class = "footer grayBg";
    document.PageFooter.Add(new TextBlock((ctx) => string.Format("Page {0}
of&nbsp;", Convert.ToString(ctx.CurrentPage+1))));
    document.PageFooter.Add(new PageCount(2));

    // save document
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A6));
}
```

We used two different styles in order to show header and footer styling using class selectors and also applied an additional style which sets gray background color for both. Header contains static `TextBlock` and footer has dynamic page label placed on the right of the page created using `TextBlock` based on delegate combined with `PageCount` element.

The content for header and footer gets generated for each page separately. It lets you avoid code duplication and helps to easily define various repeating headers and footers separated from the main content generation routine.

An image below shows how the resulting document looks like:



Pic. 53 Page header and footer usage

4.4.16 Lists

There are two list types which can be created using Flow Layout API: ordered and unordered. The first one consists of numbered items while the second represents a plain list. Their list items can have various markers assigned and it's possible to mix markers within one list. Nested lists are supported as well.

There is no special content element representing a list and it's implemented using a `Section`. Once you have to create a list you create a section object, set its `ListStyle` property to a desired list style and use other properties described in section [4.3.2.6 List specific properties](#) to define the appearance of list items.

Content element added to the section marked as list, becomes a list item only if its `ListStyle` property is set to `ListItem`. List items are being always considered as *Block* elements.

Nested numbering e.g. 1.2.3 (for ordered lists) can be implemented by using several nested sections each marked as ordered list.

Custom markers (for unordered lists) can be created using static member function defined for `ListMarker` type and called `FromResourceId()`. It uses an id of the existing `XObject` defined in the same way as for usage with [4.4.8 ContentReference](#) element. One is required to set list item marker otherwise it will appear unmarked.

See sections [4.4.16.1 Ordered list sample](#) and [4.4.16.2 Unordered list sample](#) for the detailed list creation code samples.

4.4.16.1 Ordered list sample

Consider the sample code provided below, it creates several numbered lists side by side and styles them using different numeric markers. Third list demonstrates how to start numeration using explicitly specified counter, define multilevel lists and unmark one of the items.

```
// create output file
using (Stream outputStream = File.Create("ordered_list.pdf"))
{
    // create document and set margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // defines style for elements with class "ol" - ordered list
    document.StyleManager.RegisterStyle(".ol",
        new Style()
        {
            ListStyle = ListStyle.Ordered, Display = Display.InlineBlock, Width = Length.FromPercentage(25)
        });
    // define style for all textblocks nested into the element marked with class "ol" (ordered list)
    document.StyleManager.RegisterStyle(".ol > TextBlock",
        new Style()
        {
            ListStyle = ListStyle.ListItem
        });
    // define marker classes for to be used with ordered lists
    document.StyleManager.RegisterStyle(".decimal",
        new Style()
        {
            ListMarker = ListMarker.Decimal
        });
    document.StyleManager.RegisterStyle(".roman",
        new Style()
        {
            ListMarker = ListMarker.UpperRoman
        });
    document.StyleManager.RegisterStyle(".latin",
        new Style()
        {
            ListMarker = ListMarker.UpperLatin, ListCounter = new ListCounter(0)
        });
    document.StyleManager.RegisterStyle(".decimal10",
        new Style()
        {
            ListMarker = ListMarker.Decimal, ListCounter = new ListCounter(10)
        });
}
```

...code continues on next page

```

/* simple numbered list with decimal markers */
Section list1 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item"),
    new TextBlock("Third item"),
    new TextBlock("Fourth item"),
    new TextBlock("Fifth item"))
{
    Class = "ol decimal"
};

/* simple numbered list with roman markers */
Section list2 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item"),
    new TextBlock("Third item"),
    new TextBlock("Fourth item"),
    new TextBlock("Fifth item"))
{
    Class = "ol roman"
};

/* complex numbered list with decimal markers, numbering starts from 10 and it has nested list */
// inner numbered list part 1
Section innerList1 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item"),
    new TextBlock("Third item"),
    new TextBlock("Fourth item"),
    new TextBlock("Fifth item"))
{
    Class = "ol decimal", Width = Length.Auto, Color = RgbColors.Red
};

// inner numbered list part 2
Section innerList2 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item"),
    new TextBlock("Third item"),
    innerList1,
    new TextBlock("Fifth item"))
{
    Class = "ol decimal", Width = Length.Auto, Color = RgbColors.Green
};

```

...code continues on next page


```

// outer list
Section list3 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item") { ListMarker = ListMarker.None },
    new TextBlock("Third item"),
    new TextBlock("Fourth item"),
    innerList2,
    new TextBlock("Fifth item"))
{ Class = "ol decimal10" };

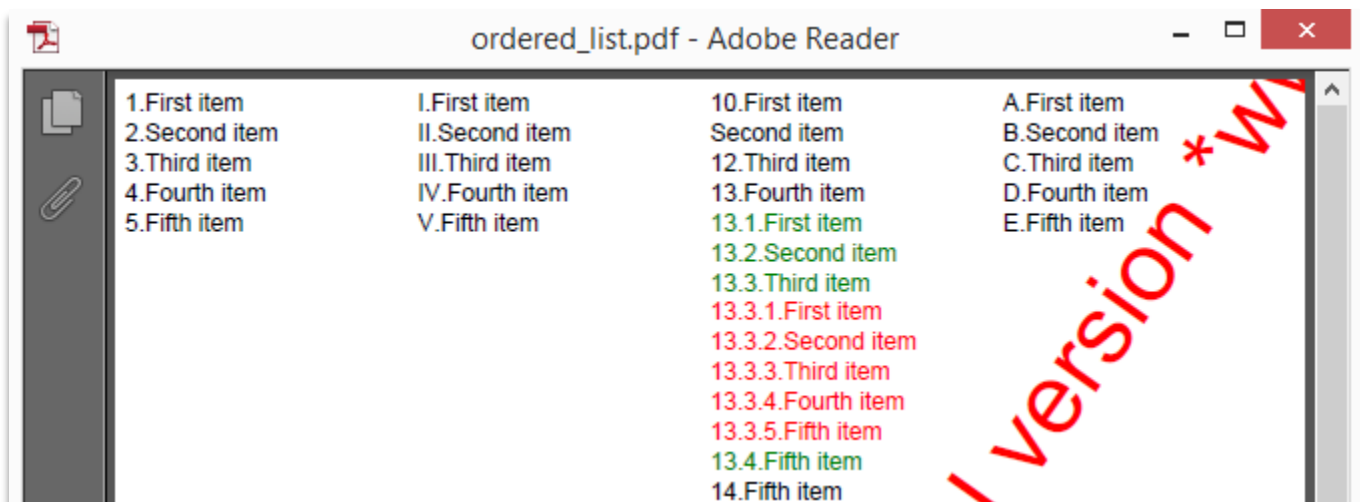
/* simple numbered list with latin markers*/
Section list4 = new Section(
    new TextBlock("First item"),
    new TextBlock("Second item"),
    new TextBlock("Third item"),
    new TextBlock("Fourth item"),
    new TextBlock("Fifth item"))
{ Class = "ol latin" };

// add lists to document
document.Add(list1);
document.Add(list2);
document.Add(list3);
document.Add(list4);

// save document
document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}

```

The image below demonstrates the result. Notice different markers used for all lists. Also note multilevel numeration, list counter starting from 10 and explicitly unmarked list item in third list ("Second item").



Pic. 54 Ordered list creation sample

4.4.16.2 Unordered list sample

The following code creates several unordered lists side by side and assigns different item markers to each of them. Child items are turned into list items using descendants selector based on parent class.

```
using (Stream outputStream = File.Create("unordered_list.pdf"))
{
    // create document and set margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // defines style for elements with class "ul" - unordered list
    document.StyleManager.RegisterStyle(".ul",
        new Style()
        {
            ListStyle = ListStyle.Unordered,
            Display = Display.InlineBlock,
            Width = Length.FromPercentage(25),
        });

    // define style for all elements nested into the element marked with class "ul"
    document.StyleManager.RegisterStyle(".ul > *",
        new Style() { ListStyle = ListStyle.ListItem });

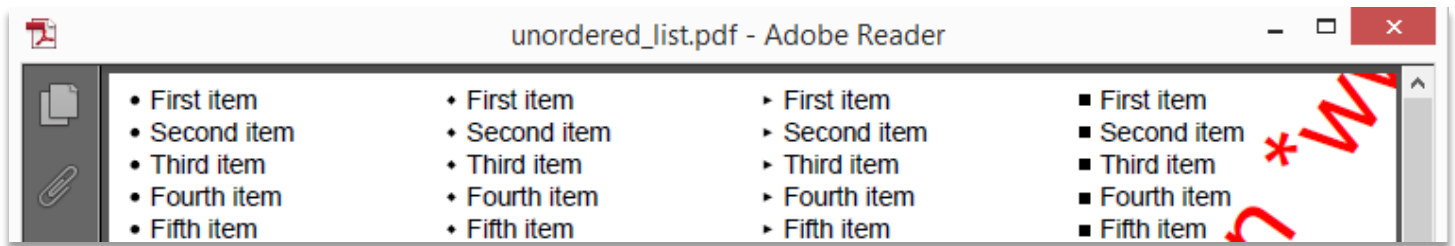
    // define unordered markers array used to create new lists
    ListMarker [] listMarkers = {ListMarker.Circle, ListMarker.Diamond, ListMarker.Triangle,
    ListMarker.Square};

    // create lists
    foreach (ListMarker listMarker in listMarkers)
    {
        // create list and set its marker using instance property
        Section list = new Section(
            new TextBlock("First item"),
            new TextBlock("Second item"),
            new TextBlock("Third item"),
            new TextBlock("Fourth item"),
            new TextBlock("Fifth item"))
        {
            Class = "ul", ListMarker = listMarker, ListMarkerPadding = new Thickness(5)
        };

        document.Add(list);
    }

    // save document
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

An image below shows the resulting PDF file generated by the code from previous page:



Pic. 55 Unordered list usage sample

Markers padding is being set using `ListMarkerPadding` property which defines how much space will be left around the marker. By default each marker doesn't have any padding assigned and may look glued to its item.

One may use marker padding to define its indent from the marked item and vertical offset that gives full control over its relative position.

4.4.16.3 Defining custom list markers

The following code sample shows how to use custom markers for list items:

```
using (Stream outputStream = File.Create("custom_markers.pdf"))
{
    // register image XObject resource for custom marker
    ResourceManager resourceManager = new ResourceManager();
    resourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.XObjects.Image("myMarker", "marker.png"));

    // create document and set margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

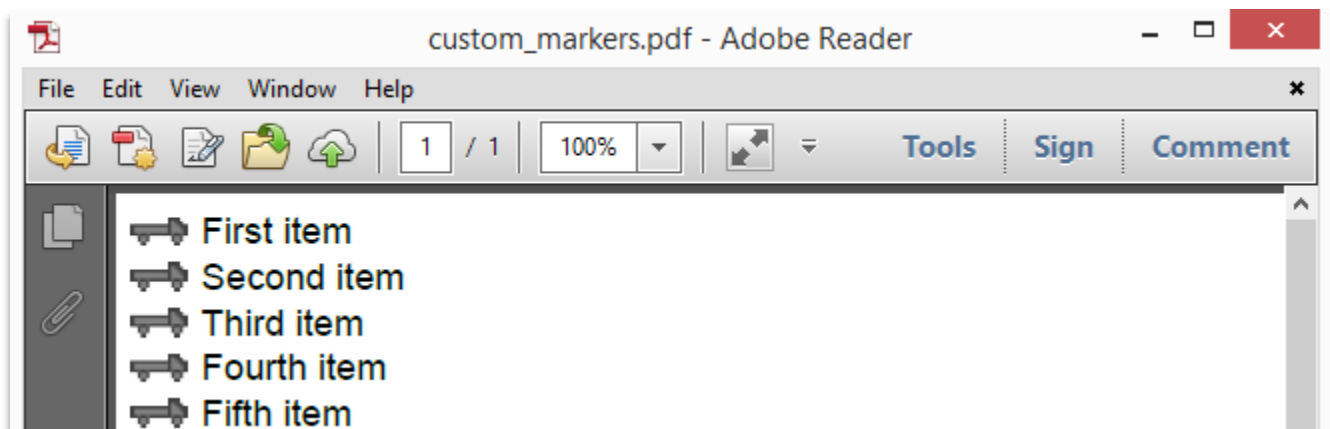
    // defines style for elements with class "ul" - ordered list
    document.StyleManager.RegisterStyle(".ul", new Style() { ListStyle = ListStyle.Unordered });

    // define style for all elements nested into the element marked with class "ul"
    document.StyleManager.RegisterStyle(".ul > *", new Style() { ListStyle = ListStyle.ListItem });

    // create list and set its marker using instance property
    Section list = new Section( new TextBlock("First item"), new TextBlock("Second item"),
        new TextBlock("Third item"), new TextBlock("Fourth item"), new TextBlock("Fifth item"))
    {
        Class = "ul", ListMarker = ListMarker.FromResourceId("myMarker"),
        ListMarkerPadding = new Thickness(0,0,5,0)
    };
    document.Add(list);

    document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

Resulting list with custom markers is shown on the image below:



Pic. 56 Custom list marker usage

4.5 Navigation

Flow layout API supports all navigation features described in PDF specification and demonstrated in sections [3.4 Document-level navigation](#), [3.5.2 Use link annotations for quick navigation](#). Also there many actions based samples from chapters under [3.6 Actions](#) section. Concepts behind the navigation techniques remain the same but, being different from Fixed layout API, Flow layout API provides own implementation for these operations.

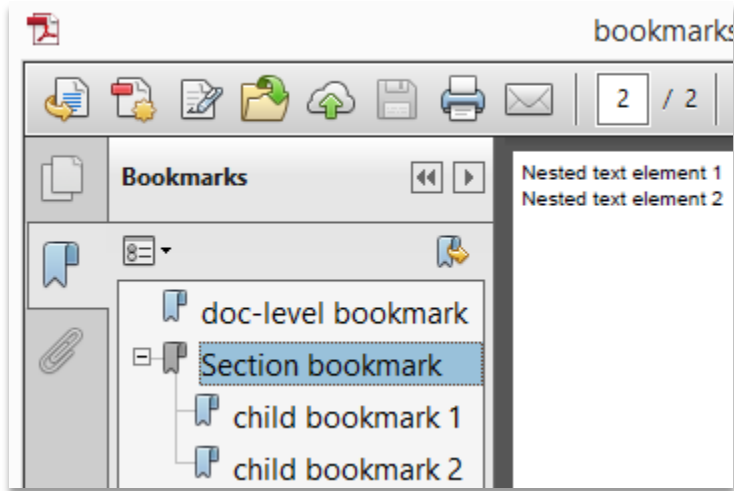
Each content element has two properties responsible for navigation:

- `Bookmark`, which controls whether bookmark entry should be created for this element in document bookmarks list. It's of type `BookmarkEntry` and defines bookmark text as well as other properties needed to display it in bookmarks tab of the viewer. Bookmarks set for content elements naturally create hierarchical structure which in its turn will be replicated in bookmarks tree of the document.
- `Link`, which makes element essentially a link pointing to some destination. Can be of several types, see chapter [4.5.2 Links](#) for details.

Please read subsequent chapters which contain descriptions for all mentioned properties and types with code samples and explanations.

4.5.1 Bookmarks

An object of type `BookmarkEntry` representing document bookmark can be attached to almost any content element; however there are a few exclusions: `Br` and `GridRow` elements. They were excluded because it's hard to imagine a situation where it could be needed. The first one is always invisible and the second is navigatable using any cell it contains.



Pic. 57 PDF document containing bookmarks

Bookmarks are organized hierarchically, so bookmark entries assigned to nested elements will be added as children to parent element bookmark if it exists. If it doesn't exist they'll go up and so on, until they reach the root.

Every bookmark can have either static or dynamic caption assigned. Dynamic caption, as its name suggests, can be generated on the fly using a delegate. It makes possible to

customize bookmarks creation according to the context state they are in. For example we can use current time or page number in bookmark caption.

Consider the code below:

```
using (Stream outputStream = File.Create("bookmarks.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5); }
    // add text element as immediate child of the document
    document.Add(new TextBlock("Text element with bookmark") { Bookmark = new
    BookmarkEntry("doc-level bookmark") });
    // create nested text elements
    TextBlock nestedElement1 = new TextBlock("Nested text element 1") { Bookmark = new
    BookmarkEntry("child bookmark 1") };
    TextBlock nestedElement2 = new TextBlock("Nested text element 2") { Bookmark = new
    BookmarkEntry("child bookmark 2") };
    // add page break and move to second page
    document.Add(new PageBreak());
    // add section containing text elements and assign a bookmark to it
    document.Add(new Section(nestedElement1, new Br(), nestedElement2) { Bookmark = new
    BookmarkEntry("Section bookmark") });
    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

4.5.2 Links

Links can be used to embed navigation elements into document page. Same limitations apply as for bookmarks, so `Br` and `GridRow` elements can't act as links. Other content elements can be used for navigation by assigning a target to their `Link` property. Once user clicks on such link, navigation will occur. Currently two link types are supported:

- `CrossReference`, navigates to a location within the same document
- `LinkUri`, based on URI and navigates to external resource

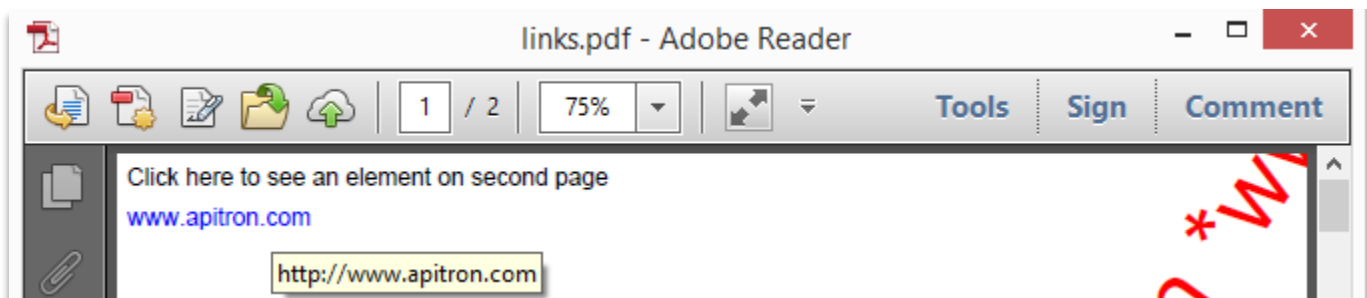
Note: An element can have both `Bookmark` and `Link` properties set.

Let's see the code:

```
using (Stream outputStream = File.Create("links.pdf"))
{
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // add link to other element in document
    document.Add(new TextBlock("Click here to see an element on second page"){Link = new
CrossReference("destination")});
    document.Add(new Br(){Height = 5});
    // add external link to a website
    document.Add(new TextBlock("www.apitron.com"){Link = new
LinkUri("http://www.apitron.com"),Color = RgbColors.Blue});

    // add page break and move to second page
    document.Add(new PageBreak());
    // create destination text block, set its id so it could be referenced by link
    document.Add(new TextBlock("Destination element") { Id = "destination"});
    // save to output stream with page size A4
    document.Write(outputStream, new ResourceManager(), new PageBoundary(Boundaries.A4));
}
```

And resulting image:



Pic. 58 Add links to PDF document

4.6 Markup parsing

Content elements defined in Flow Layout API can be created based on some markup to reflect desired document structure. It becomes especially useful when you have to create paragraphs containing blocks of text having different styles. So we designed a simple way to do it. A `ContentElement` class which serves as a base for all defined content elements contains a static method called `FromMarkup()`, it creates a collection of elements using given text as markup.

This markup looks similar to XML markup, given that each element will have its *Class* property assigned to the name or a set of names derived from the enclosing tags. The resulting list will contain parsed content elements (mostly *TextBlocks*) with assigned classes. A single markup element can contain the following attributes:

- *Link*, affects the *Link* property of the created element. If link destination starts with # then it will be considered as a *CrossReference* otherwise as a *LinkUri*. There is an additional *href* attribute supported for convenience that works exactly the same.
- *Bookmark*, affects the *Bookmark* property of the created element.
- *Id*, affects the *Id* property of the created element, so it can be linked to.

A few tags have special meaning:

- ``, can be used for placing images, e.g. `` creates [Image](#) element entry. Unit can be omitted or can be one of the following: `auto|pt|in|px|cm|%`.
- `
`, creates [Br](#) element entry.

Note: Given that markup text is in XML format and its parsing will be done by XML-aware utils, entries which don't exist in XML like ` ` should be entered differently. So instead of `<tag> mytext. </tag>` one could use `<tag> mytext. </tag>` or `<tag> mytext. </tag>` which would produce the same result.

4.6.1 Markup parsing example

For the detailed sample please take a look at the *SimpleHtmlToPDF* sample from Apitron PDF Kit for NET download package. Below is a small piece of code that shows some of the features of markup parsing in action.

```
// create output file
using (Stream outputStream = File.Create("create_from_markup.pdf"))
{
    // create resource manager and register image to be used further
    ResourceManager resourceManager = new ResourceManager();
    resourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.XObjects.Image("logo", "apitron.png"));
    // create document and set margin
    FlowDocument document = new FlowDocument(){Margin = new Thickness(10);

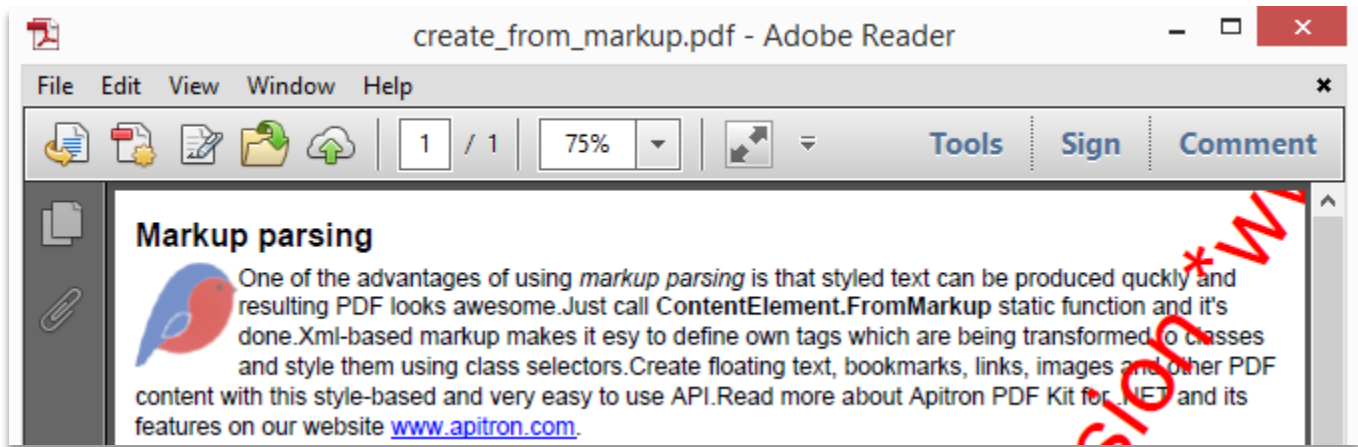
    // header style
    document.StyleManager.RegisterStyle(".h1", new Style(){Display = Display.Block, Font = new
    Font(StandardFonts.HelveticaBold, 16), Margin = new Thickness(0, 3, 0, 3)});
    // italic text style
    document.StyleManager.RegisterStyle(".i", new Style(){Font = new
    Font(StandardFonts.HelveticaOblique, 12)});
    // bold text style
    document.StyleManager.RegisterStyle(".b", new Style(){Font = new
    Font(StandardFonts.HelveticaBold, 12)});
    // link style
    document.StyleManager.RegisterStyle(".a", new Style() { Color = RgbColors.Blue, TextDecoration =
    new TextDecoration(TextDecorationOptions.Underline)});
    // image style
    document.StyleManager.RegisterStyle(".img", new Style() { Float = Float.Left});

    string markup =
        "<h1>Markup parsing</h1><img src='logo'></img>One of the advantages of using <i>markup
        parsing</i> is that" +
        " styled text can be produced quickly and resulting PDF looks awesome." +
        "Just call <b>ContentElement.FromMarkup</b> static function and it's done." +
        "XML-based markup makes it easy to define own tags which are being transformed to classes and
        style them using class selectors." +
        "Create floating text, bookmarks, links, images and other PDF content with this style-based and
        very easy to use API." +
        "Read more about Apitron PDF Kit for .NET and its features on our website <a
        href='http://www.apitron.com'>www.apitron.com</a>.";

    document.Add(new Section(ContentElement.FromMarkup(markup)));
    document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

The code from previous page registers several styles for the text markup representing a small piece of text containing an image with a text floating on the right. It also shows how to define a weblink, make text **bold** or *italic*. Special class for header becomes defined as well ("h1").

Resulting file is shown on the image below:



Pic. 59 Create PDF from XML-based markup

4.7 XML templates

`FlowDocument` support XML export and import features, and one can use it to separate PDF documents creation from the code. As an example you may imagine a set of PDF forms stored as XML template (supported by Apitron PDF Kit) which would require regular update. In this case an application wouldn't require to be rebuilt and updated and would load new templates and work as it did. Looks like a much simpler and more reliable scenario.

Limitations: some resource objects can't be saved to XML, e.g. image data (because images can be linked only), `FixedContent` or `FlowContent` objects, external font files data.

The code which saves document to XML and loads it from template is as follows:

```
// save document to XML template (1)
using (Stream outputStream = File.Create("document_template.xml"))
{
    // create document resource manager
    ResourceManager resourceManager = new ResourceManager();
    // create new document with margin
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };

    // add content elements
    document.Add(new TextBlock("text element"));

    //...create your document structure here

    document.SaveToXml(outputStream, resourceManager);
}

// load XML template from file (2)
using (Stream inputStream = File.OpenRead("document_template.xml"), outputStream =
File.Create("fromxml.pdf"))
{
    // create resource manager which will hold loaded references
    ResourceManager resourceManager = new ResourceManager();
    // load document
    FlowDocument document = FlowDocument.LoadFromXml(inputStream, resourceManager);
    // save to pdf
    document.Write(outputStream, resourceManager, new PageBoundary(Boundaries.A4));
}
```

Here we used `SaveToXml` instance method to save document as XML template (1) and `LoadFromXml` static method to load this template (2).

4.7.1 Create PDF document using modified XML template

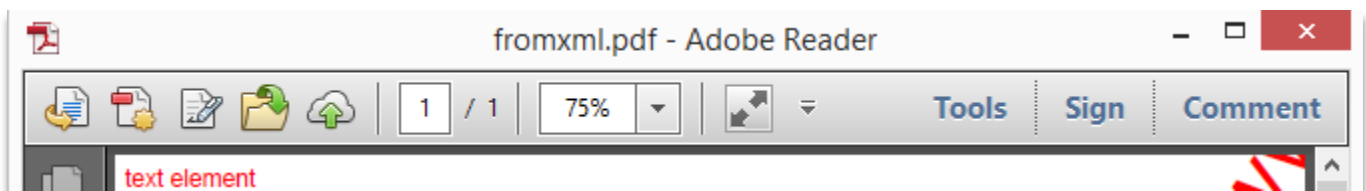
Consider the following template generated using the code from [4.7 XML templates](#):

```
<?xml version="1.0" encoding="utf-8"?>
<FlowDocument xmlns="Apitron.PDF.Kit.FlowLayout.v1">
  <Resources />
  <Styles>
    <Style selector="flowdocument">
      <Color value="Black" />
    </Style>
    <Style selector="grid">
      <InnerBorder thickness="1" />
      <InnerBorderColor value="Black" />
    </Style>
  </Styles>
  <Elements>
    <TextBlock>
      <Properties>
        <Text value="text element" /> (1)
      </Properties>
    </TextBlock>
  </Elements>
  <Properties>
    <Margin value="5,5,5,5" />
  </Properties>
</FlowDocument>
```

If you change the template by adding a section:

```
<Style selector="textblock">
  <Color value="Red" />
</Style>
```

to the **<Styles>** node of the template and load it using part (2) of the code from the previous chapter you'll get the following color change:



Pic. 60 Create PDF document using modified XML template

4.8 Integration with Fixed Layout API

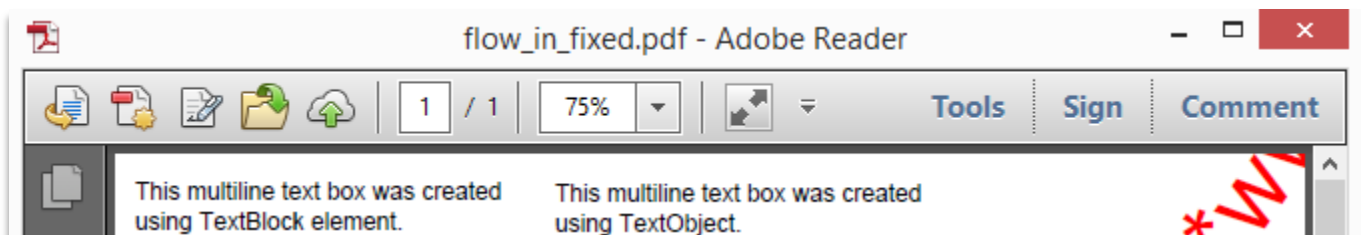
You may have noted that our Flow Layout API samples make use of many classes defined for Fixed Layout API. We decided not to create duplicating entities serving as alternatives of the existing things where it was appropriate. However while working on `FixedDocument` it would be advantageous sometimes to make use of flexibility Flow Layout API can offer. So we added a special method called `AppendContentElement()` to a `ClippedContent` class, which is implemented by every drawing context in Fixed Layout API. This method accepts `ContentElement` objects defined in Flow Layout API and allows using them directly for building PDF content with Fixed Layout API.

Consider the code:

```
using (Stream outputStream = File.Create("flow_in_fixed.pdf"))
{
    // create document and add one page to it
    FixedDocument fixedDocument = new FixedDocument();
    fixedDocument.Pages.Add(new Page());
    ClippedContent content = fixedDocument.Pages[0].Content;
    // add TextBlock content element to the page content
    content.Translate(10,790);
    content.AppendContentElement(new TextBlock("This multiline text box was created using TextBlock
element."), 190,40);

    // add text using regular textobject, you'll have to layout all text yourself
    content.Translate(210,28);
    TextObject textObject = new TextObject(StandardFonts.Helvetica, 12);
    textObject.AppendText("This multiline text box was created");
    textObject.MoveToNextLine(0,-15);
    textObject.AppendText("using TextObject.");
    content.AppendText(textObject);
    // save document
    fixedDocument.Save(outputStream);
}
```

And resulting image:



Pic. 61 Usage of ContentElement in FixedDocument

4.9 Miscellaneous

Chapters below describe techniques and Flow Layout API subsets aiming to simplify some of the tasks one may encounter during the PDF processing and which are not covered in other sections.

4.9.1 Change page size and styles on the fly

`FlowDocument` uses fixed page size and single styles set by default and every page becomes sized and styled using these settings. Sometimes where is a need to change the style (e.g. booklet with page numbers placed on the left for odd pages and on the right for even pages). Page size could also require a change for some cases.

These settings can be changed for a particular page or a range of pages by providing a handler for `NewPage` event defined by `FlowDocument` class.

*Note: only elements which are being **created** on particular page will be affected by style or page size change. If element spans across multiple pages, then its style will be defined by the start page.*

Consider the code below:

```
using (Stream outputStream = File.Create("custom_page_settings.pdf"))
{
    FlowDocument document = new FlowDocument() { Margin = new Thickness(5) };
    // register style for page header using id selector
    document.StyleManager.RegisterStyle("#header", new Style()
        { Align = Align.Left, Background = RgbColors.LightGray });
    // define new page handler
    document.NewPage += (args) =>
    {
        // redefine style for pageheader and page size for pages with indices 1, 3, 5 etc.
        if ((args.Context.CurrentPage & 0x1) != 0)
        {
            args.OverridingStyleManager.RegisterStyle("#header", new Style()
                {
                    Align = Align.Right, Background = RgbColors.Gray
                });
            args.PageBoundary = new PageBoundary(new Boundary(0, 0, 300, 500));
        }
    };
}
```

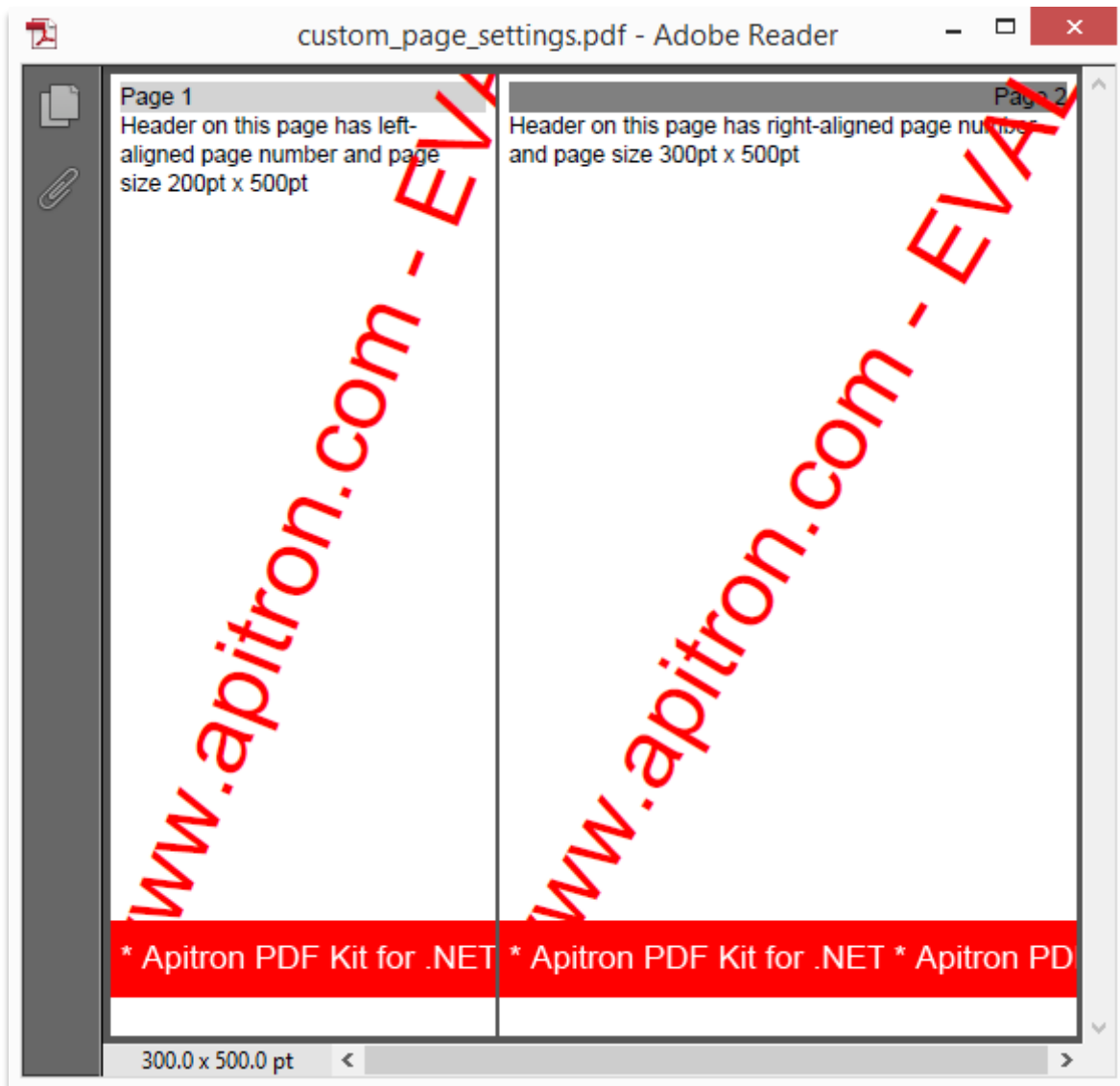
...code continues on next page

```

// define page header, display current page index
document.PageHeader.Id = "header";
document.PageHeader.Add(new TextBlock((ctx)=>string.Format("Page {0}",ctx.CurrentPage+1)));
// add content
document.Add(new TextBlock("Header on this page has left-aligned page " +
    "number and page size 200pt x 500pt"));
document.Add(new PageBreak());
document.Add(new TextBlock("Header on this page has right-aligned page number" +
    " and page size 300pt x 500pt"));
// save document
document.Write(outputStream, new ResourceManager(),
    new PageBoundary(new Boundary(0,0,200,500)));
}

```

This code redefines page header style and page size for even pages and produces the document that looks as follows:



Pic. 62 Change page size and style on the fly

4.9.2 Use patterns to fill elements background

Imagine the situation that you'd like to fill background of the PDF document or any of its elements with some repeating pattern e.g. snowflakes to create a postcard or winter sale newsletter. You could use `BackgroundImage` property of course, but it's not always helpful and is limited being a raster drawing (implies scaling limitations) and also takes space. `Color` class defined in Flow Layout API subset supports specifying colors in PATTERN colorspace and, therefore it can be used to fill elements background with all pattern types supported in PDF (it can even be used as text color or border color because it's essentially a color).

See the sample below that fills document background with pattern based on image:

```
using (FileStream fs = new FileStream(@"output.pdf", FileMode.Create))
{
    // create documents resource manager and register pattern image
    ResourceManager resourceManager = new ResourceManager();
    resourceManager.RegisterResource(new
    Apitron.PDF.Kit.FixedLayout.Resources.XObjects.Image("snowflakes", "snowflakes.png", true));

    // create colored tiling pattern, draw image inside and register it.
    TilingPattern pattern = new TilingPattern("myPattern", new Boundary(0,0,100,100), 100, 100, true);
    pattern.Content.AppendImage("snowflakes", 0, 0, 100, 100);
    resourceManager.RegisterResource(pattern);

    // create document
    FlowDocument document = new FlowDocument(){Padding = new Thickness(10)};
    // set color, defined as PATTERN color as a background color for document
    document.Background = new Color(PredefinedColorSpaces.Pattern, "myPattern");
    // add sample text block
    document.Add(new TextBlock("Sample for background patterns"){Color = RgbColors.Red});
    // generate document
    document.Write(fs, resourceManager, new PageBoundary(Boundaries.A4));
}
```

Resulting PDF file looks as follows:



Pic. 63 Use pattern background for document

5. Useful libraries and resources

5.1 Convert PDF to image

[Apitron](http://apitron.com) provides a cross-platform and fully managed PDF rendering component which you may use to convert PDF files to images using C#, VB.NET or any other .NET-compatible language. This library is available for any .NET framework version starting from 2.0 as well as for *Mono*, *Xamarin.iOS*, *Xamarin.Android*, *Windows 8/8.1*, *Windows RT* and *Windows Phone*.

You can read more about it using the following link:

<http://apitron.com/Product/pdf-rasterizer>

5.2 Search text and get its position on PDF page

While Apitron PDF Kit can be used to search any text in PDF document in general, Apitron PDF Rasterizer for .NET provides reach and easy to use API which, in addition, allows extraction of text coordinates. Supports right to left and bidirectional text.

Read more about PDF text search here:

<http://blog.apitron.com/2014/03/search-text-in-pdf-documents-sample-code.html>

5.3 PDF Viewer control for .NET

Our company provides an easy to use, full-featured and **FREE** Windows Forms PDF Viewer control for customers owning a license for our *Apitron PDF Rasterizer for .NET* product. This viewer can also be used in WPF applications via *WindowsFormsHost*.

You can read more about it using the following link:

<http://apitron.com/Product/pdf-controls>

Blog entries showing this control in action:

<http://blog.apitron.com/2013/12/free-pdf-viewer-control-for-windows-forms.html>

<http://blog.apitron.com/2014/07/adding-pdf-preview-functionality-to-WPF-application.html>

6. Contacts and links

Whenever you are interested in any of our PDF components, contact us and we'll be happy to assist you and offer the best possible solution.

Use support@apitron.com for general queries and support requests or sales@apitron.com for questions related to purchase.

More information can be found by the links below, all resources are being constantly updated and are a good source of information regarding PDF generation and processing.

www.apitron.com – our website, get latest information about our components, view samples.

blog.apitron.com – company blog, contains lots of code samples and product demos.

[@Apitron](https://twitter.com/Apitron) – our twitter feed, stay in touch and read latest news from Apitron.