# iText 7 Building Blocks

Bruno Lowagie

iText Software

# iText 7: Building Blocks

iText Software

This book is for sale at http://leanpub.com/itext7buildingblocks

This version was published on 2016-08-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 iText Software

# Contents

**Before we start: Overview of the classes and interfaces** . . . . . . . . . . . . . . . . . . . . **2**

**Chapter 1: Introducing the PdfFont class** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **9**
Creating a PdfFont object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
Embedding a font . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
Choosing the appropriate encoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Font properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Reusing styles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**Chapter 2: Working with the RootElement** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **24**
Using Canvas to add content inside a Rectangle . . . . . . . . . . . . . . . . . . . . . . . 24
Converting text to PDF with the Document class . . . . . . . . . . . . . . . . . . . . . . 31
Changing the Document renderer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
Switching between different renderers . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
Flushing the Document renderer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
Changing content that was previously added . . . . . . . . . . . . . . . . . . . . . . . . 46
Adding a Page X of Y footer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
Adding text with showTextAligned . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Using iText 7 add-ons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Improving the typography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

**Chapter 3: Using ILeafElement implementations** . . . . . . . . . . . . . . . . . . . . . . . . . **58**
Working with Tab elements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
Limitations of the Tab functionality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Adding links . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
Extra methods available in the Text class . . . . . . . . . . . . . . . . . . . . . . . . . . 69
Introducing images . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Changing the position and width of an image . . . . . . . . . . . . . . . . . . . . . . . . 74
Adding an image to an existing PDF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
Resizing and rotating an image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
Image types supported by iText . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

**Chapter 4: Adding AbstractElement objects (part 1)** . . . . . . . . . . . . . . . . . . . . . . . **93**
Grouping elements with the Div class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
Drawing horizontal lines with the LineSeparator object . . . . . . . . . . . . . . . . . . . . 97
Keeping content together . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
Changing the leading of a Paragraph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
Creating a custom renderer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
Lists and list symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
Adding ListItem objects to a List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
Nested lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

**Chapter 5: Adding AbstractElement objects (part 2)** . . . . . . . . . . . . . . . . . . . . . . . **119**
My first table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
Table and cell Alignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
Row and cell height . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
Cell margins and padding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
Table and cell borders . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
Nesting tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
Repeating headers and footers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
Images in tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
Splitting cells versus keeping content together . . . . . . . . . . . . . . . . . . . . . . . . 142
Table and cell renderers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
Tables and memory use . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152

**Chapter 6: Creating actions, destinations, and bookmarks** . . . . . . . . . . . . . . . . . . **154**
URI actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
Named actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
GoTo actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
Named destinations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
Remote GoTo actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
JavaScript actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
Chained actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
Destinations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
Link annotations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171
Outlines aka bookmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
Color and style of the outline elements. . . . . . . . . . . . . . . . . . . . . . . . . . . . 177
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 180

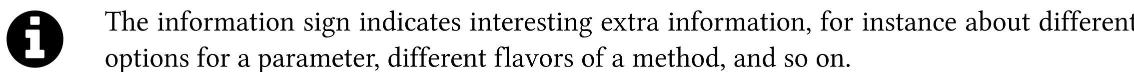
**Chapter 7: Handling events; setting viewer preferences and writer properties** . . . . . . **181**
Implementing the IEventHandler interface . . . . . . . . . . . . . . . . . . . . . . . . . . 181
Adding a background and text to every page . . . . . . . . . . . . . . . . . . . . . . . . 184
Solving the “Page X of Y” problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187

Adding a transparent background image . . . 191
Insert and remove page events . . . 195
Page labels . . . 196
Page display and page mode . . . 198
Viewer preferences . . . 202
Printer preferences . . . 204
Open action and additional actions . . . 206
Writer properties . . . 209
Summary . . . 215

**Appendix . . . 216**
A: AbstractElement methods . . . 216
B: BlockElement methods . . . 217
C: RootElement methods . . . 218

This is the second tutorial in the iText 7 series. In the first tutorial, iText 7: Jump-Start Tutorial[1], we discussed a series of examples that explained the core functionality of iText 7. In this book, we'll focus on the high-level building blocks that were introduced in the first chapter of that tutorial: Introducing basic building blocks[2]. In that chapter, we created PDFs using objects such as `Paragraph`, `List`, `Image` and `Table`, but we didn't go into detail. This tutorial is the extended version of that chapter. In this tutorial, you'll discover which building blocks are available and how they all relate to each other.

Throughout the book, we'll use the following symbols:

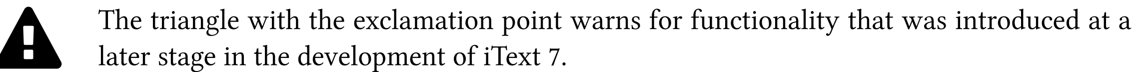


The information sign indicates interesting extra information, for instance about different options for a parameter, different flavors of a method, and so on.

The question mark will be used when this information is presented in the form of a question and an answer.

The bug highlights an `Exception` that gets thrown if you make a common mistake.

The text balloons are used for a chatty remark or a clarification.



The triangle with the exclamation point warns for functionality that was introduced at a later stage in the development of iText 7.

The key indicates functionality that is new in iText 7, or at least very different from what developers were used to in iText 5 or earlier versions.

All the examples of this book along with the resources needed to build them, are available online at the following address: http://developers.itextpdf.com/content/itext-7-building-blocks/examples[3]

[1] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial
[2] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/chapter-1-introducing-basic-building-blocks
[3] http://developers.itextpdf.com/content/itext-7-building-blocks/examples

# Before we start: Overview of the classes and interfaces

When we talk about iText 7's basic building blocks, we refer to all classes that implement the `IElement` interface. iText 7 is originally written in Java, then ported to C#. Because of our experience with both programming languages, we've adopted the convenient habit â€“typical for C# developersâ€“ to start every name of an interface with the letter `I`.

Figure 0.1 shows an overview of the relationship between `IElement` and some other interfaces.

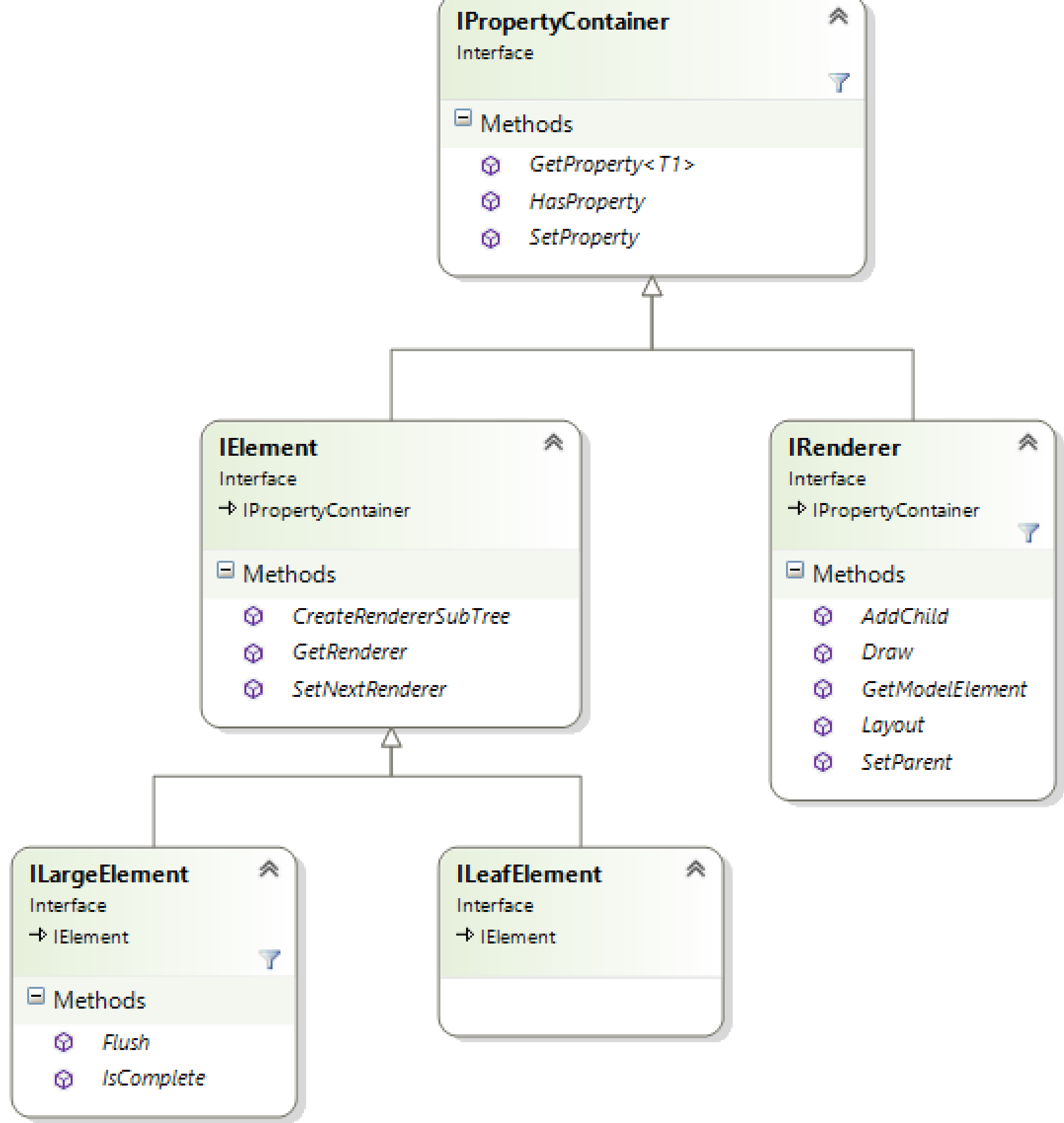



**Figure 0.1: Overview of the interfaces**

At the top of the hierarchy, we find the `IPropertyContainer` interface. This interface defines methods to set, get, and delete properties. This interfaces has two direct subinterfaces: `IElement` and `IRenderer`. The `IElement` interface will be implemented by objects such as `Text`, `Paragraph` and `Table`. These are the objects that we'll add to a document, either directly or indirectly. The `IRenderer` interface will be implemented by objects such as `TextRenderer`, `ParagraphRenderer` and `TableRenderer`. These renderers are used internally by iText, but we can subclass them if we want to tweak the way an object is rendered.

The `IElement` interface has two subinterfaces of its own. The `ILeafElement` interface will be implemented by building blocks that can't contain any other elements. For instance: you can add a `Text` or an `Image` element to a `Paragraph` object, but you can't add any object to a `Text` or an `Image`

element. `Text` and `Image` implement the `ILeafElement` interface to reflect this. Finally, there's the `LargeElement` interface that allows you to render an object before you've finished adding all the content. It's implemented by the `Table` class, which means that you add a table to a document before you've completed adding all the `Cell` objects. By doing so, you can reduce the memory use: all the table content that can be rendered before the content of the table is completed, can be flushed from memory.

The `IPropertyContainer` interface is implemented by the abstract `ElementPropertyContainer` class. This class has three subclasses; see figure 0.2.

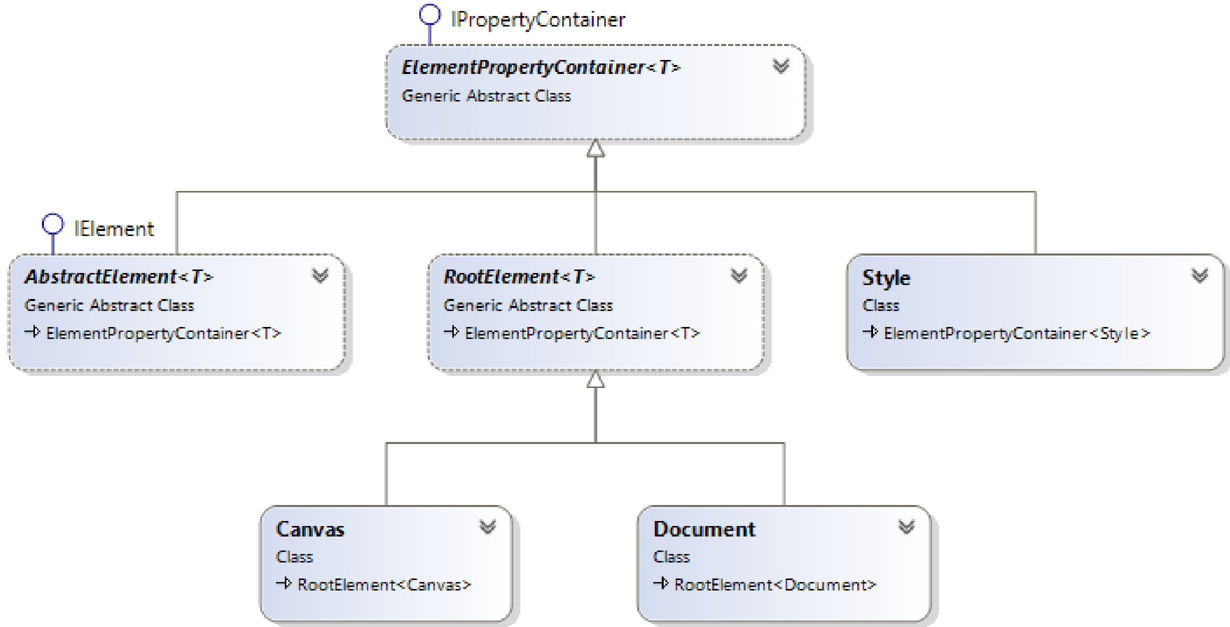



**Figure 0.2: Implementations of the IPropertyContainer interface**

The `Style` class is a container for all kinds of style attributes such as margins, paddings and rotation. It inherits style values such as widths, heights, colors, borders and alignments from the abstract `ElementPropertyContainer` class.

The `RootElement` class defines methods to add content, using either an `add()` method or a `showTextAligned()` method. The `Document` object will add this content to a page. The `Canvas` object doesn't know the concept of a page. It acts as a bridge between the high-level *layout* API and the low-level *kernel* API.

Figure 0.3 gives us an overview of the `AbstractElement` implementations.

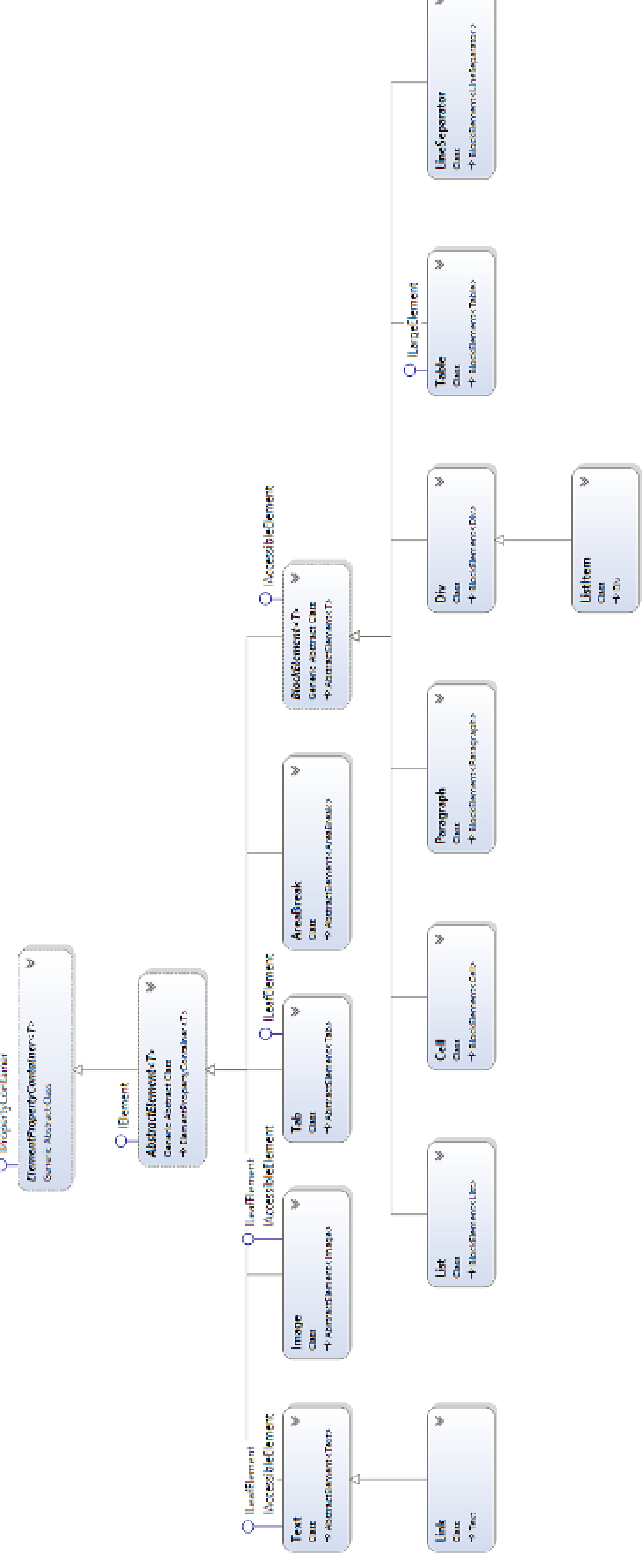



**Figure 0.3: Implementations of the IElement interface**

All classes derived from the `AbstractElement` class implement the `IElement` interface. `Text`, `Image`, `Tab` and `Link` also implement the `ILeafElement` interface. The `ILargeElement` interface is only implemented by the `Table` class. The basic building blocks make it very easy for you to create *tagged PDF*. Tagged PDF is a requirement for PDF/A, a standard for long-term preservation of document, and, PDF/UA, an accessibility standard. A properly tagged PDF includes semantic information about all the relevant content.

An ordinary PDF can show a human reader content that is organized as a table. This table is rendered using a bunch of text snippets and lines. To a machine, the table isn't more than that: text positioned at arbitrary places, lines drawn at arbitrary places. A seeing person can detect rows and columns and understand which rows are actually header or footer rows and which rows are body rows. There is no simple way for a machine to do this. When a machine detects a text snippet, it doesn't know if that text snippet is part of a paragraph, part of a title, part of a cell, or part of something else. When a PDF is tagged, it contains a structure tree that allows a machine to understand the structure of the content. Some text will be marked as part of a cell in a header row, other text will be marked as the caption of the table. All *real content* will be tagged. Other content, such as lines between rows and columns, running headers, page numbers, will be marked as an *artifact*.

In iText, we have introduced the `IAccessibleElement` interface. It is implemented by all the basic building blocks that contain real content: `Text`, `Link`, `Image`, `Paragraph`, `Div`, `List`, `ListItem`, `Table`, `Cell`, `LineSeparator`. If we define a `PdfDocument` as a tagged PDF using the `setTagged()` method, iText will create a structure tree so that a `Table` is properly tagged as a table, a `List` properly tagged as a list, and so on. There is no real content in a `Tab` or an `AreaBreak`, which is why these classes don't implement that interface. It's just white space; a tab and an area break don't even need to be marked as an artifact.

In this tutorial, we won't create tagged PDF; iText will just render the content to the document using the appropriate `IRenderer` implementation. Figure 0.4 shows an overview of the `IRenderer` implementations.

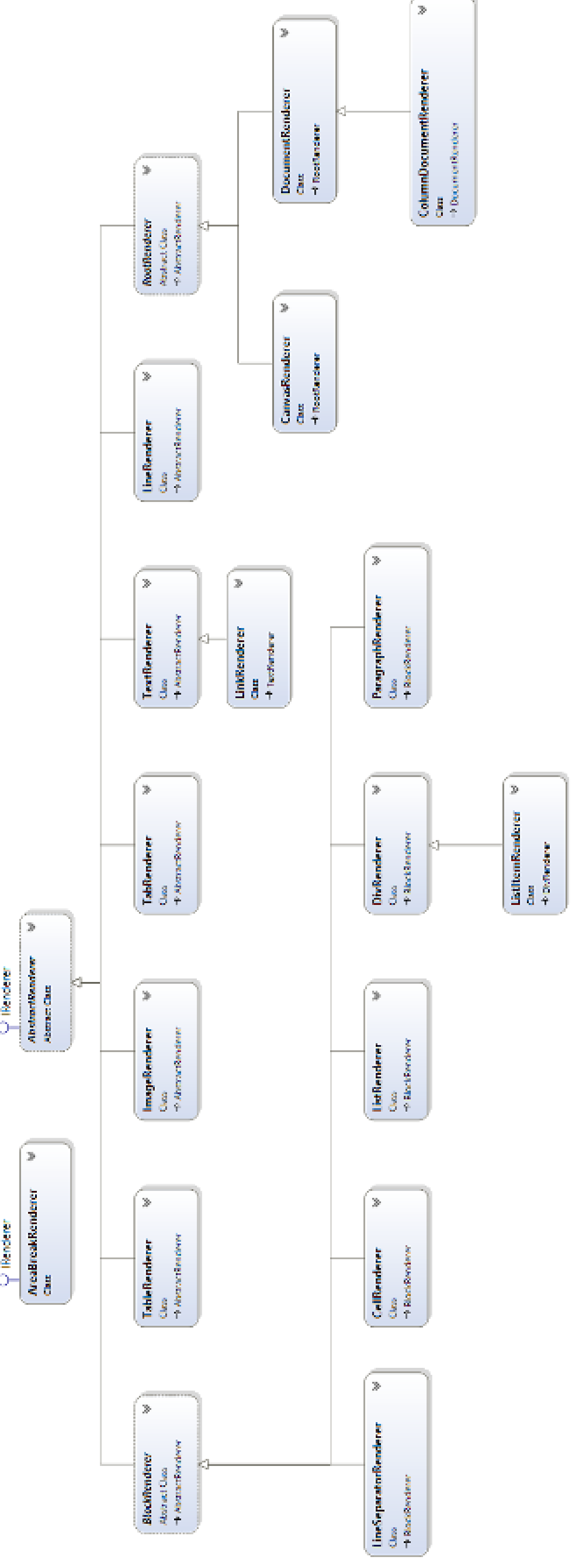



**Figure 0.4: Implementations of the IRenderer interface**

When you compare figure 0.4 with 0.3, you'll discover that each `AbstractElement` and each `RootElement` has its corresponding renderer. We won't discuss figure 0.4 in much detail. The concept of renderers will become clear the moment we start making some examples.

# Chapter 1: Introducing the PdfFont class

When writing a tutorial, I always prefer working with real-world use cases. That's not always easy because real-world use cases can get quite complex, whereas a tutorial needs to explain different concepts as simple as possible. While I was looking for a theme for this tutorial, I stumbled upon the short story "The Strange Case of Dr. Jekyll and Mr. Hyde" by Robert Louis Stevenson. I made a first example turning a plain text file into a PDF eBook and I liked the result. When I discovered how many movies, cartoons and series were made based on this work, I saw an opportunity to create a database that could be converted into a table. The movie posters could serve as sample material when discussing images in PDF.

But first things first: let's start with an example that displays the title and the author in different fonts. The `PdfFont` class doesn't appear in any of the hierarchical charts showing the relationship between element interfaces and classes, but it's needed for all the building blocks that involve text. We could spend a complete tutorial about fonts (and we probably will), but this chapter will explain the basic font functionality that you need to be aware of.

## Creating a PdfFont object

If we look at figure 1.1, we see that three different fonts were used to create a PDF document with the title and the author of the Jekyll and Hyde story: Helvetica, Times-Bold and Times-Roman. In reality, three other fonts are used by the viewer: ArialMT, TimesNewRomanPS-BoldMT and TimesNewRomanPSMT.

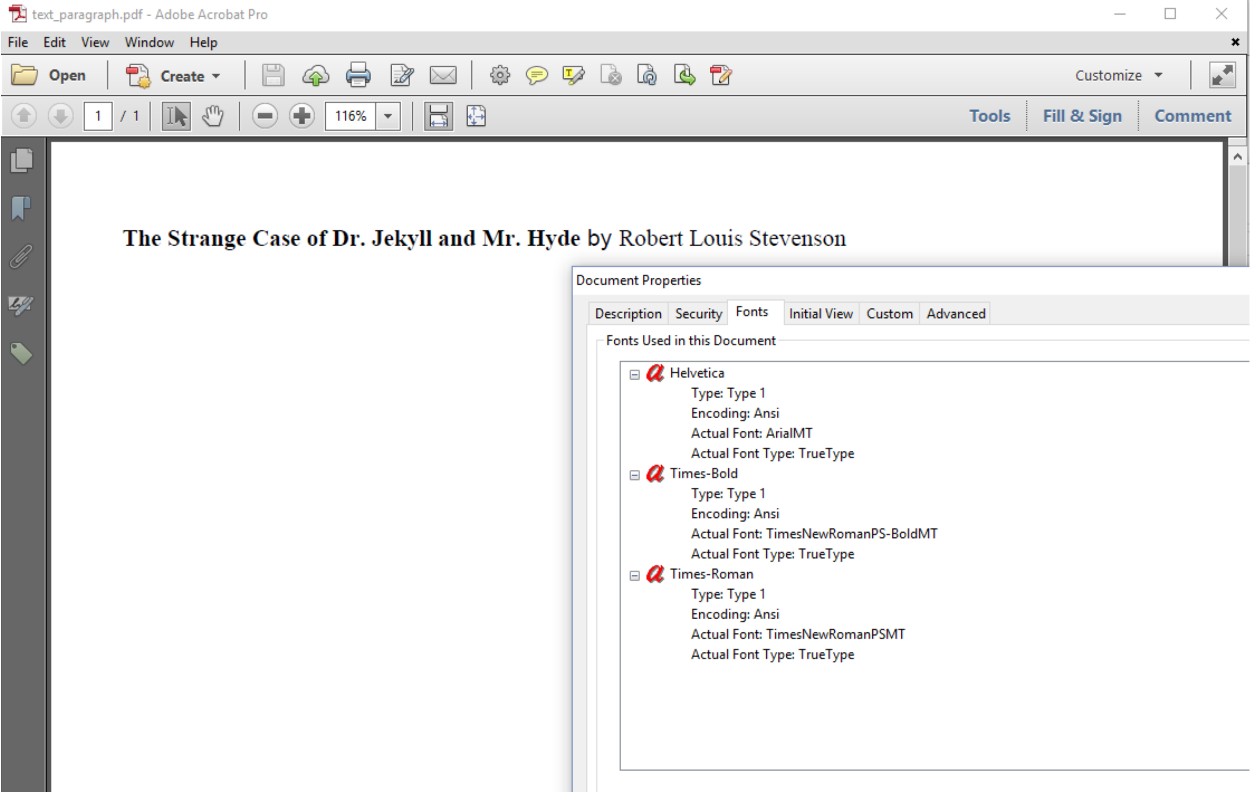



**Figure 1.1: Standard Type 1 fonts**

The MT in the names of the *Actual Font* refers to the vendor of the fonts: the Monotype Imaging Holdings, Inc. These are fonts shipped with Microsoft Windows. If you'd open the same file on a Linux machine, other fonts would be used as actual fonts. This is typically what happens when you don't embed fonts. The viewer searches the operating system for the fonts that are needed to present the document. If a specific font can be found, another font will be used instead.

Traditionally, there are 14 fonts that every PDF viewer should be able to recognize and render in a reliable way: four Helvetica fonts (normal, bold, oblique, and bold-oblique), four Times-Roman fonts (normal, bold, italic, and bold-italic), four Courier fonts (normal, bold, oblique, and bold-oblique), Symbol and Zapfdingbats. These fonts are often referred to as the Standard Type 1 fonts. Not every viewer will use that exact font, but it will use a font that looks almost identical.

To create the PDF shown in figure 1.1, we used three of these fonts: we defined two fonts explicitly; one font was defined implicitly. See the Text_Paragraph[4] example.

[4]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1822-c01e01_text_paragraph.java

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  Document document = new Document(pdf);
3  PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
4  PdfFont bold = PdfFontFactory.createFont(FontConstants.TIMES_BOLD);
5  Text title =
6      new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
7  Text author = new Text("Robert Louis Stevenson").setFont(font);
8  Paragraph p = new Paragraph().add(title).add(" by ").add(author);
9  document.add(p);
10 document.close();
```

In line 1, we create a PdfDocument using a PdfWriter as parameter. These are low-level objects that will create PDF output based on your content. We're creating a Document instance in line 2. This is a high-level object that will allow you to create a document without having to worry about the complexity of PDF syntax.

In lines 5 and 6, we create a PdfFont using the PdfFontFactory. In the FontConstants object, you'll find a constant for each of the 14 Standard Type 1 fonts. In line 7, we create a Text object with the title of Stevenson's short story and we set the font to TIMES_BOLD. In line 8, we create a Text object with the name of the author and we set the font to TIMES_ROMAN. We can't add these Text objects straight to the document, but we add them to a BlockElement, more specifically a Paragraph, in line 9.

Between the title and the author, we add " by " as a String object. Since we didn't define a font for this String, the default font of the Paragraph is used. In iText, the default font is Helvetica. This explains why we see the font Helvetica listed in the font overview in figure 1.1.

In line 10, we add the paragraph to the document object; we close the document object in line 11.

We have created our first Jekyll and Hyde PDF using fonts that aren't embedded. As a result, slightly different fonts can be used when rendering the document. We can avoid this by embedding the fonts.

# Embedding a font

iText supports the Standard Type 1 fonts, because the io-jar contains the Adobe Font Metrics (AFM) files of those 14 fonts. iText can't embed these 14 fonts because the PostScript Font Binary (PFB) files are proprietary. They can't be shipped with iText because iText Group doesn't have a license to do so. We are only allowed to ship the metrics files.

In the Text_Paragraph_Cardo[5] example, we use three fonts of the Cardo font family. These are fonts that were released under the Summer Institute of Logistics (SIL) Open Font License (OFL). The result is shown in figure 1.2.

[5]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1823-c01e02_text_paragraph_cardo.java

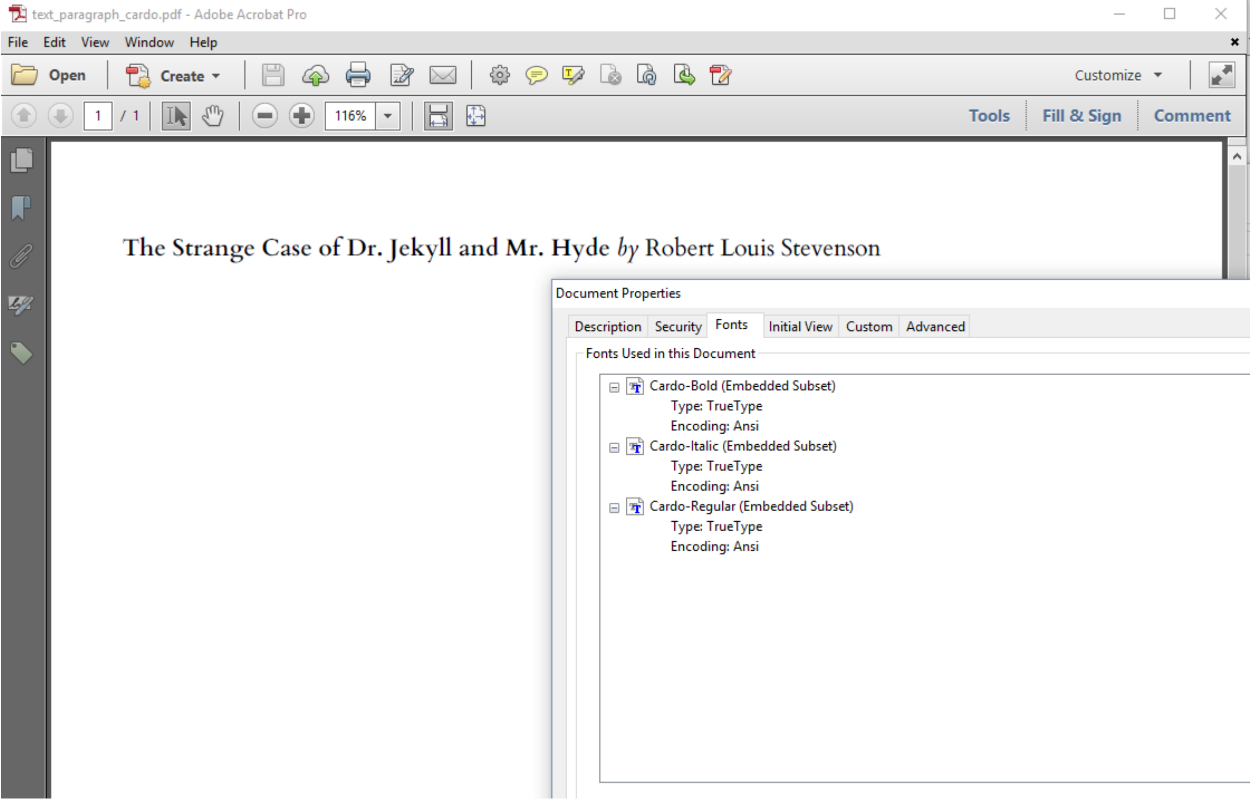



**Figure 1.2: Embedded fonts**

First we need the path to the font programs for the three Cardo fonts: `Cardo-Regular.ttf`, `Cardo-Bold.ttf` and `Cardo-Italic.ttf`:

```
1 public static final String REGULAR =
2     "src/main/resources/fonts/Cardo-Regular.ttf";
3 public static final String BOLD =
4     "src/main/resources/fonts/Cardo-Bold.ttf";
5 public static final String ITALIC =
6     "src/main/resources/fonts/Cardo-Italic.ttf";
```

In line 1 to 3 of the following snippet, we use these paths as the first parameter of the `createFont()` method. The second parameter is a Boolean indicating whether or not we want to embed the font.

```java
PdfFont font = PdfFontFactory.createFont(REGULAR, true);
PdfFont bold = PdfFontFactory.createFont(BOLD, true);
PdfFont italic = PdfFontFactory.createFont(ITALIC, true);
Text title =
    new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
Text author = new Text("Robert Louis Stevenson").setFont(font);
Paragraph p = new Paragraph().setFont(italic)
    .add(title).add(" by ").add(author);
document.add(p);
```

Line 4 to 6 are identical to what we had before, but in line 7, we change the default font of the `Paragraph` to `italic`. This explains why `" by "` was written in italic in figure 1.2 and why the font Helvetica no longer appears in the font list. In line 7, we add the `Paragraph` to the `Document` instance.

Figure 1.3 shows what would happen if we don't embed the fonts.

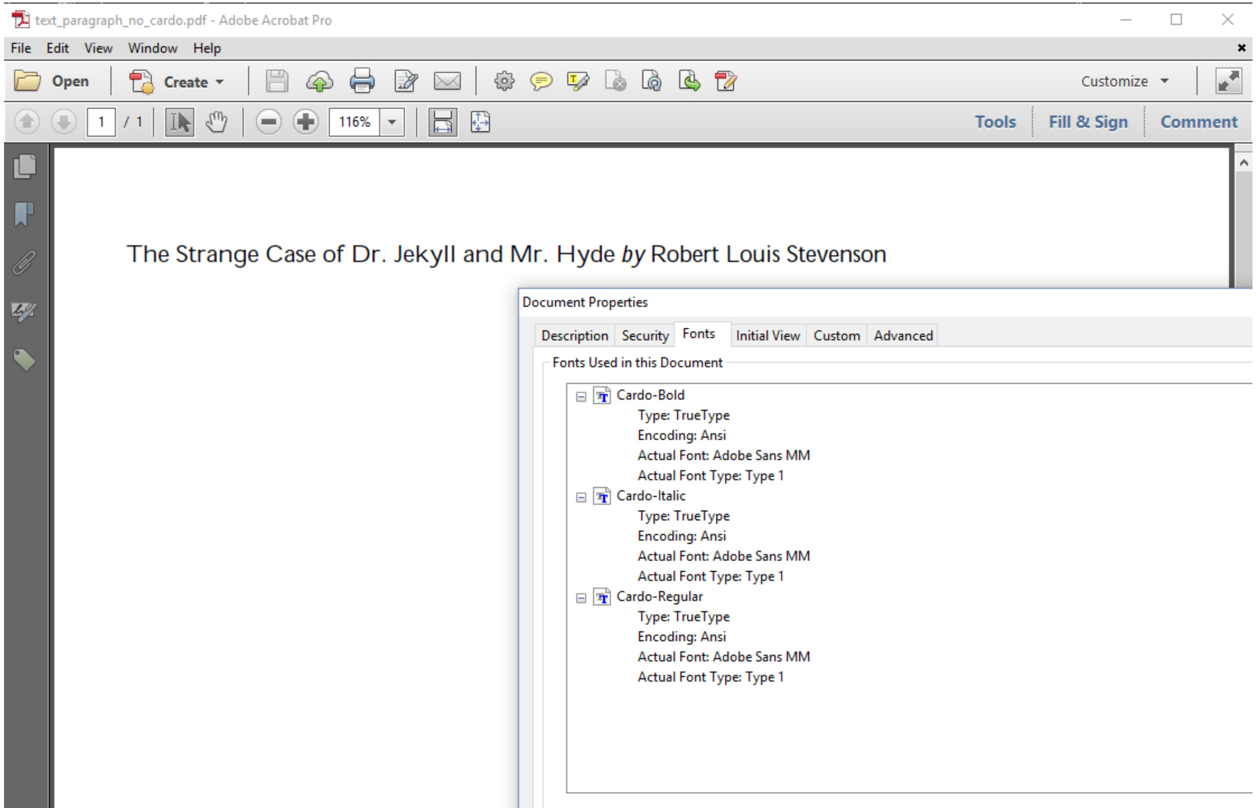



**Figure 1.3: Ugly font substitution**

In the Text_Paragraph_NoCardo[6] example, we have defined the fonts like this:

[6]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1824-c01e03_text_paragraph_nocardo.java

```
1 PdfFont font = PdfFontFactory.createFont(REGULAR);
2 PdfFont bold = PdfFontFactory.createFont(BOLD);
3 PdfFont italic = PdfFontFactory.createFont(ITALIC);
```

The constants `REGULAR`, `BOLD` and `ITALIC` refer to the correct Cardo `.ttf` files, but we omitted the parameter that tells iText to embed the font. Incidentally, the Cardo fonts aren't present on my PC. Adobe Reader replaced them with Adobe Sans MM. As you can see, the result doesn't look nice. If you don't use any of the standard Type 1 fonts, you should always embed the font.

The problem is even worse when you try to create PDFs in different languages. In figure 1.4, we try to add some text in Czech, Russian and Korean. The Czech text looks more or less OK, but we'll soon discover that there one character missing. The Russian and Korean text is invisible.

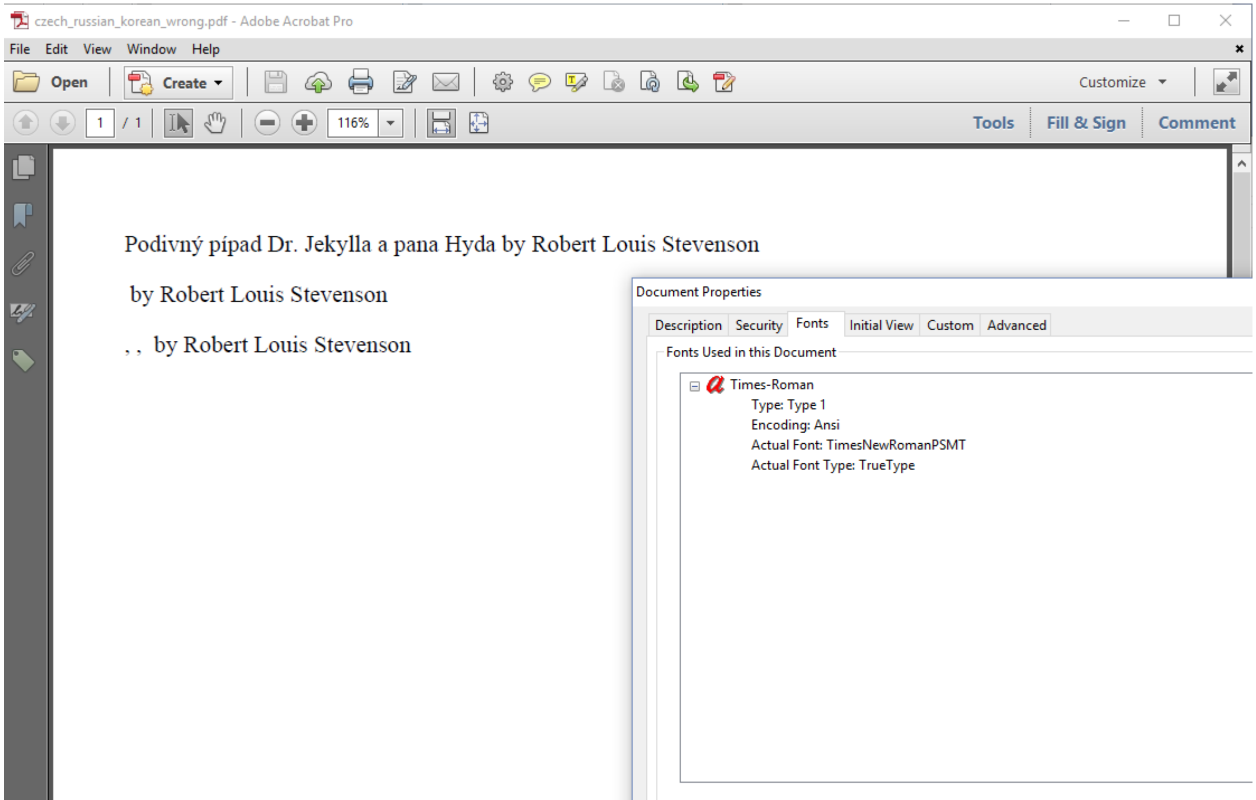



**Figure 1.4: Wrong rendering of Czech, Russian and Korean**

Not embedding the font isn't the only problem here. We also need to define the appropriate encoding.

# Choosing the appropriate encoding

In figure 1.4, we tried to render the following text:

> Podivný případ Dr. Jekylla a pana Hyda by Robert Louis Stevenson

> Странная история доктора Джекила и мистера Хайда by Robert Louis Stevenson
>
> ☒☒☒, ☒☒, ☒ by Robert Louis Stevenson

The first line is the Czech translation of "The Strange Case of Dr. Jekyll and Mr. Hyde." If you look closely at figure 1.4, you'll see that the character ř is missing. That's because the ř character is missing in the Winansi encoding. Winansi, also known as code page 1252 (CP-1252), Windows 1252, or Windows Latin 1, is a superset of ISO 8859-1 also known as Latin-1. It's a character encoding of the Latin alphabet, used by default in many applications on Western operating systems.

For the Czech text, we need to use another encoding. One option is to use code page 1250, an encoding to represent text in Central European and Eastern European languages that use Latin script. The second line reads as *Strannaya istoriya doktora Dzhekila i mistera Khayda*. For this text, we could use code page 1251, an encoding designed to cover languages that use the Cyrillic script. Cp1250 and Cp1251 are 8-bit character encodings. The third line is Korean for *Hyde, Jekyll, Me*, a South-Korean television series loosely based on the Jekyll and Hyde story. We can't use an 8-bit encoding for Korean. To render this text, we need to use Unicode. Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.

When you create a font using an 8-bit encoding, iText will create a *simple font* for the PDF. A simple font consists of at most 256 characters that are mapped to at most 256 glyphs. When you create a font using Unicode (in PDF terms: *Identity-H* for horizontal writing systems or *Identity-V* for vertical writing systems), iText will create a *composite font*. A composite font can contain 65,536 characters. This is less than the total number of available code points in Unicode (1,114,112). This means that no single font can contain all possible characters in every possible language.

Instead of Cp1250 and Cp1251, we could also use Unicode for the Czech and Russian text. Actually, when we store hard-coded text in source code, it is preferred to store Unicode values.

```
1  public static final String CZECH =
2          "Podivn\u00fd p\u0159\u00edpad Dr. Jekylla a pana Hyda";
3  public static final String RUSSIAN =
4          "\u0421\u0442\u0440\u0430\u043d\u043d\u0430\u044f "
5          + "\u0438\u0441\u0442\u043e\u0440\u0438\u044f "
6          + "\u0434\u043e\u043a\u0442\u043e\u0440\u0430 "
7          + "\u0414\u0436\u0435\u043a\u0438\u043b\u0430 \u0438 "
8          + "\u043c\u0438\u0441\u0442\u0435\u0440\u0430 "
9          + "\u0425\u0430\u0439\u0434\u0430";
10 public static final String KOREAN =
11         "\ud558\uc774\ub4dc, \uc9c0\ud0ac, \ub098";
```

We'll use the values `CZECH`. `RUSSIAN` and `KOREAN` in our next couple of examples.

### Why should we always use Unicode notations for special characters?

When the source code file is stored on disk, committed to a version control system, or transferred in any way, there's always a risk that the encoding gets lost. If a Unicode file is stored as plain text, two-byte characters change into two single-byte characters. For example, the character □ with Unicode value `\ud0ac` will change into two characters with ASCII code `d0` and `ac`. When this happens the syllable □ (pronounced as "kil") changes into `Ð¬` and the text becomes illegible. It is good practice to use the Unicode notation as done in the above snippet; this will help you avoid encoding problems with your source code.

Using the correct encoding isn't sufficient to solve every font problem you might encounter. In the Czech_Russian_Korean_Wrong[7] example, we create the `Paragraph` objects like this:

```
1 PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
2 document.add(new Paragraph().setFont(font)
3         .add(CZECH).add(" by Robert Louis Stevenson"));
4 document.add(new Paragraph().setFont(font)
5         .add(RUSSIAN).add(" by Robert Louis Stevenson"));
6 document.add(new Paragraph().setFont(font)
7         .add(KOREAN).add(" by Robert Louis Stevenson"));
```

This won't work because we didn't use the correct encoding, but also because we didn't define a font that supports Russian and Korean. We fix this problem in the Czech_Russian_Korean[8] example by embedding the free font "FreeSans" for the Czech and Russian translation of the title. We'll use a Hancom font "HCR Batang" for the Korean text.

```
1 public static final String FONT = "src/main/resources/fonts/FreeSans.ttf";
2 public static final String HCRBATANG = "src/main/resources/fonts/HANBatang.ttf";
```

We'll use these paths as the first parameter for the `PdfFont` constructor. We pass the desired encoding as the second parameter. The third parameter indicates that we want to embed the font.

[7] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1836-c01e04_czech_russian_korean_wrong.java
[8] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1837-c01e05_czech_russian_korean_right.java

```
PdfFont font1250 = PdfFontFactory.createFont(FONT, PdfEncodings.CP1250, true);
document.add(new Paragraph().setFont(font1250)
        .add(CZECH).add(" by Robert Louis Stevenson"));
PdfFont font1251 = PdfFontFactory.createFont(FONT, "Cp1251", true);
document.add(new Paragraph().setFont(font1251)
        .add(RUSSIAN).add(" by Robert Louis Stevenson"));
PdfFont fontUnicode =
    PdfFontFactory.createFont(HCRBATANG, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(fontUnicode)
        .add(KOREAN).add(" by Robert Louis Stevenson"));
```

Figure 1.5 shows the resulting PDF.

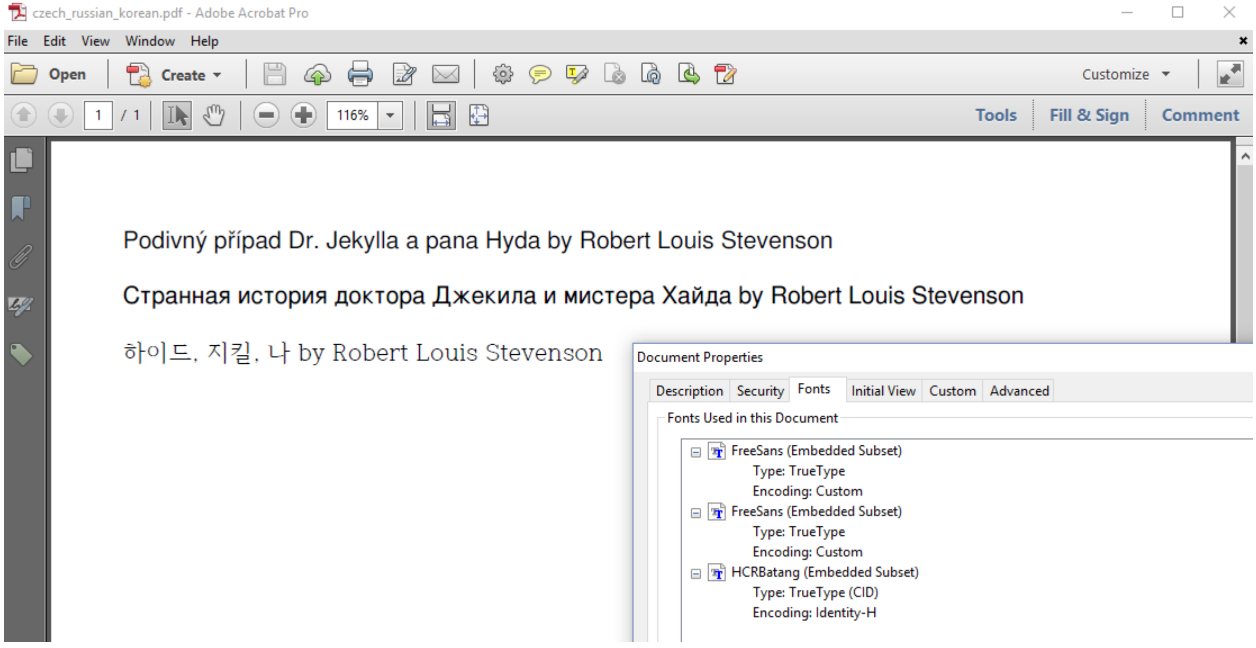



**Figure 1.5: Correct rendering of Czech, Russian and Korean**

When we look at the Fonts panel in the document properties, we notice that FreeSans is mentioned twice. That is correct: we've added the font once with the encoding Cp1250 and once with the encoding Cp1251, In the Czech_Russian_Korean_Unicode[9] example, we'll create one composite font, `freeUnicode`, for both languages, Czech and Russian.

[9] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1844-c01e06_czech_russian_korean_unicode.java

```
PdfFont freeUnicode =
    PdfFontFactory.createFont(FONT, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(freeUnicode)
    .add(CZECH).add(" by Robert Louis Stevenson"));
document.add(new Paragraph().setFont(freeUnicode)
    .add(RUSSIAN).add(" by Robert Louis Stevenson"));
PdfFont fontUnicode =
    PdfFontFactory.createFont(HCRBATANG, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(fontUnicode)
    .add(KOREAN).add(" by Robert Louis Stevenson"));
```

Figure 1.6 shows the result. The page looks identical to what we saw in figure 1.5, but now the PDF only contains one FreeSans font with Identity-H as encoding.

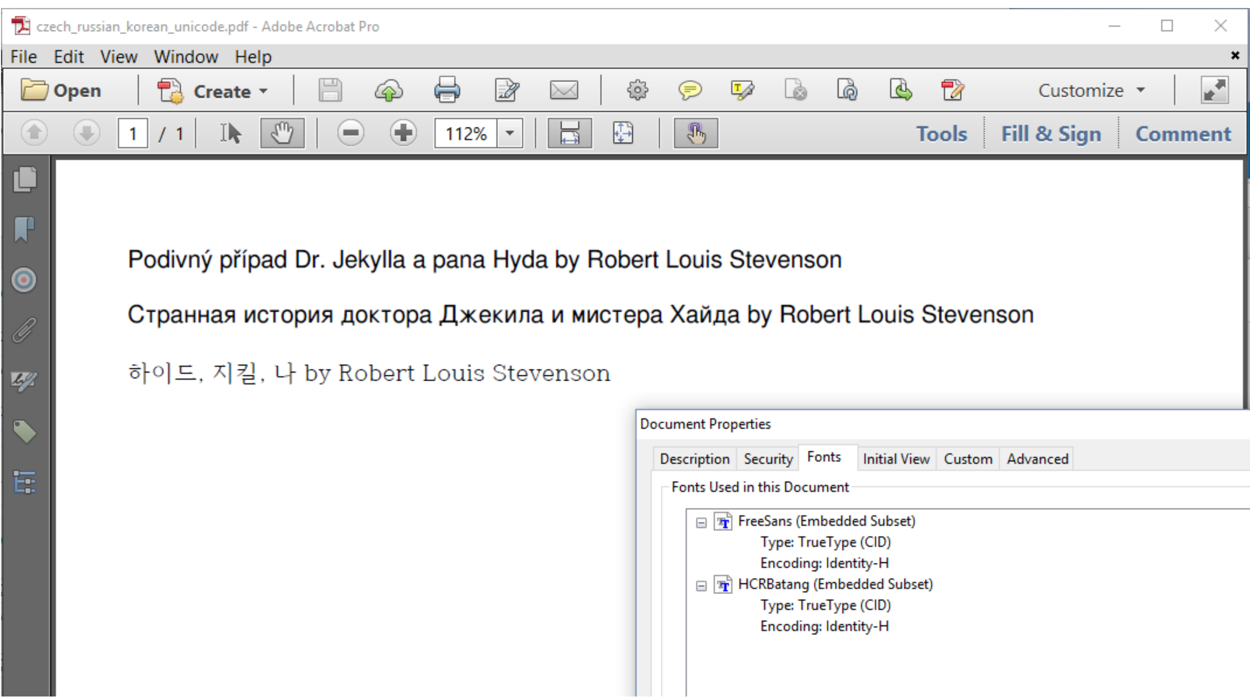



**Figure 1.6: Correct rendering of Czech, Russian and Korean (Unicode)**

Using Unicode is one of the requirements of PDF/UA and of certain flavors of PDF/A for reasons of accessibility. With custom encodings, it isn't always possible to know which glyphs are represented by each character.

In the next series of font examples, we'll experiment with some font properties such as font size, font color, and rendering mode.

# Font properties

Figure 1.7 shows a screen shot of yet another PDF with the Jekyll and Hyde title. This time, the default font Helvetica is used, but we've defined different font sizes.

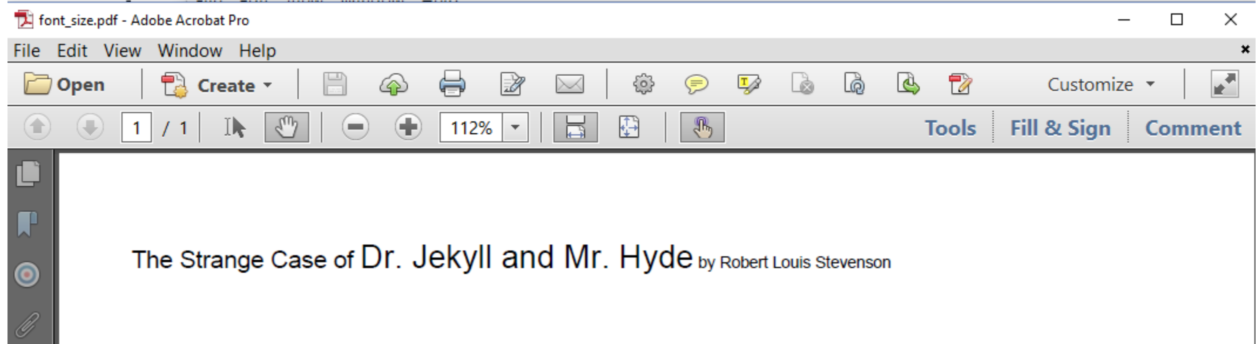



**Figure 1.7: Different font sizes**

The font size is set with the `setFontSize()` method. This method is defined in the abstract class `ElementPropertyContainer`, which means that we can use it on many different objects. In the FontSize[10] example, we use the method on `Text` and `Paragraph` objects:

```
1 Text title1 = new Text("The Strange Case of ").setFontSize(12);
2 Text title2 = new Text("Dr. Jekyll and Mr. Hyde").setFontSize(16);
3 Text author = new Text("Robert Louis Stevenson");
4 Paragraph p = new Paragraph().setFontSize(8)
5         .add(title1).add(title2).add(" by ").add(author);
6 document.add(p);
```

We set the font size of the newly created `Paragraph` to 8 pt. This font size will be inherited by all the objects that are added to the `Paragraph`, unless the objects override that default size. This is the case for `title1` for which we defined a font size of 12 pt and for `title2` for which we defined a font size of 16 pt. The content added as a `String` (`" by "`) and the content added as a `Text` object for which no font size was defined inherit the font size 8 pt from the `Paragraph` to which they are added.

In iText 5, it was necessary to create a different `Font` object if you wanted a font with a different size or color. We changed this in iText 7: you only need a single `PdfFont` object. The font size and color is defined at the level of the building blocks. We also made it possible for elements to inherit the font, font size, font color and other properties from the parent object.

In previous examples, we've worked with different fonts from the same family. For instance, we've created a document with three different fonts from the Cardo family: Cardo-Regular, Cardo-Bold,

[10] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1846-c01e07_fontsize.java

and Cardo-Italic. For most of the Western fonts, you'll find at least a regular font, a bold font, an italic font, and a bold-italic font. It will be more difficult to find bold, italic and bold-italic fonts for Eastern and Semitic languages. In that case, you'll have to mimic those styles as is done in figure 1.8. If you look closely, you see that different styles are used, yet we've only defined a single font in the PDF.

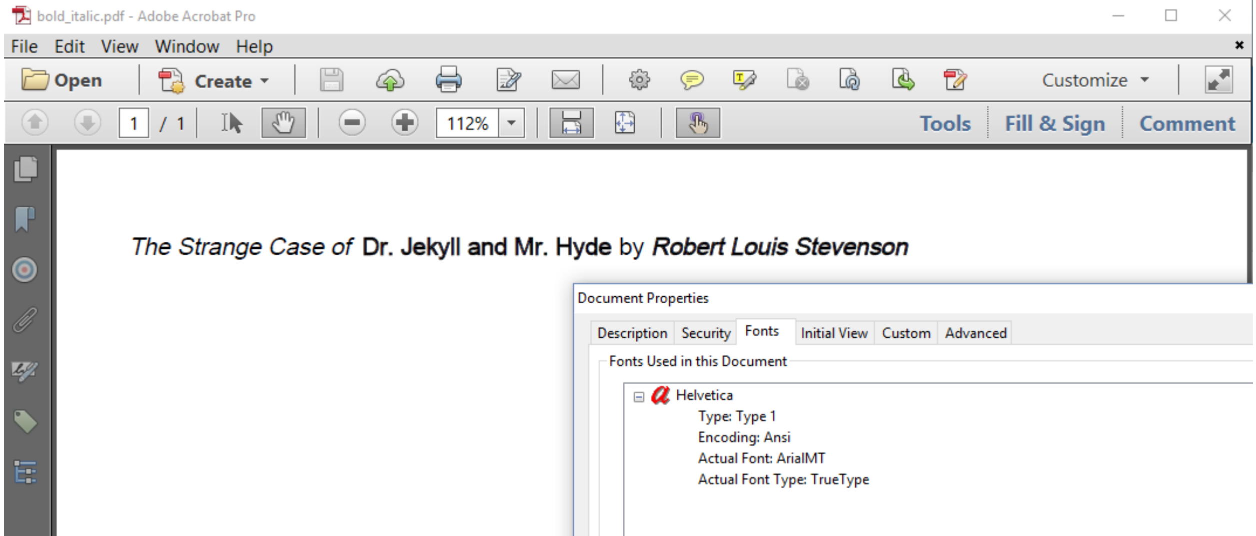



**Figure 1.8: Mimicking different font styles**

Let's take a look at the BoldItalic[11] example to find out how this was done.

```
1 Text title1 = new Text("The Strange Case of ").setItalic();
2 Text title2 = new Text("Dr. Jekyll and Mr. Hyde").setBold();
3 Text author = new Text("Robert Louis Stevenson").setItalic().setBold();
4 Paragraph p = new Paragraph()
5         .add(title1).add(title2).add(" by ").add(author);
6 document.add(p);
```

In lines 1 to 3, we use the methods `setItalic()` and `setBold()`. The `setItalic()` method won't switch from a regular to an italic font. Instead, it will skew the glyphs of the italic font in such a way that it looks as if they are italic. The `setBold()` font will change the render mode of the text and increase the stroke width. Let's introduce some color to show what this means.

Figure 1.9 shows the text using different colors and different rendering modes.

[11]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1847-c01e08_bolditalic.java

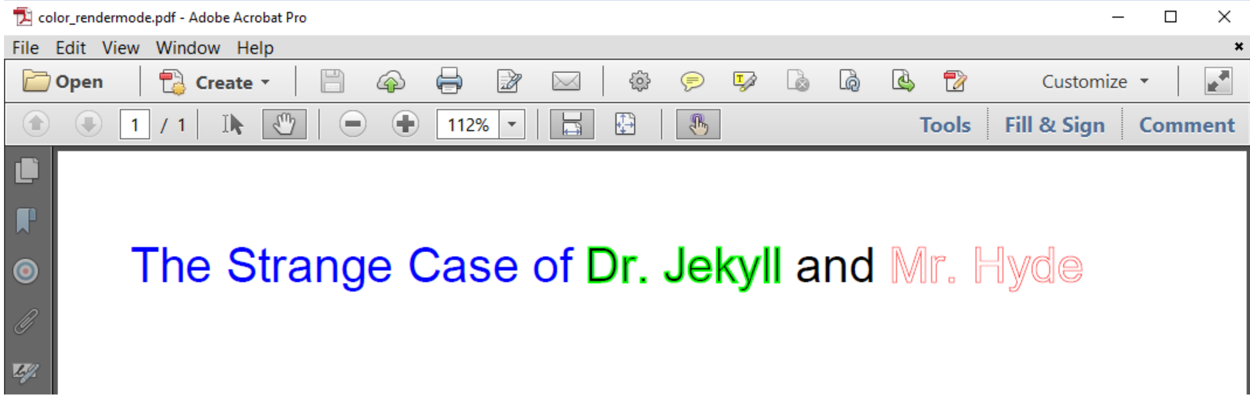



**Figure 1.9: different font colors and rendering modes**

The ColorRendering[12] example explains what happens.

```
Text title1 = new Text("The Strange Case of ").setFontColor(Color.BLUE);
Text title2 = new Text("Dr. Jekyll")
        .setStrokeColor(Color.GREEN)
        .setTextRenderingMode(PdfCanvasConstants.TextRenderingMode.FILL_STROKE);
Text title3 = new Text(" and ");
Text title4 = new Text("Mr. Hyde")
        .setStrokeColor(Color.RED).setStrokeWidth(0.5f)
        .setTextRenderingMode(PdfCanvasConstants.TextRenderingMode.STROKE);
Paragraph p = new Paragraph().setFontSize(24)
        .add(title1).add(title2).add(title3).add(title4);
document.add(p);
```

A font program contains the syntax to construct the path of each glyph. By default the path is painted using a *fill* operator, not drawn with a *stroke* operation, but we can change this default.

- In line 1, we change the font color to blue using the `setFontColor()` method. This changes the *fill color* for the paint operation that fills the paths of all the text.
- In line 2-4, we don't define a font color, which means the text will be painted in black. Instead we define a stroke color using the `setStrokeColor()` method, and we change the text rendering mode to `FILL_STROKE` with the `setTextRenderingMode()` method. As a result the contours of each glyph will be drawn in green. Inside those contours, we'll see the default fill color black.
- We don't change any of the defaults in line 5. This `Text` object will simply inherit the font size of the `Paragraph`, just like all of the other `Text` objects.
- In line 6-8, we change the stroke color to red and we use the `setStrokeWidth()` to 0.5 user units. By default, the stroke width is 1 user unit, which by default corresponds with 1 point.

[12] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1848-c01e09_colorrendering.java

> There are 72 user units in one inch by default. We also change the text rendering mode to `STROKE` which means the text won't be filled using the default fill color. Instead, we'll only see the contours of the text.

Mimicking bold is done by setting the text rendering mode to `FILL_STROKE` and by increasing the stroke width; mimicking italic is done by using the `setSkew()` method that will be discussed in chapter 3. Although this approach works relatively well, the `setBold()` and `setItalic()` method should only be used as a last resort when it's really impossible to find the appropriate fonts for the desired styles. Mimicking styles makes it very hard –if not impossible– for parsers extracting text from PDF to detect which part of the text is rendered in a different style.

# Reusing styles

If you have many different building blocks, it can become quite cumbersome to define the same style over and over again for each separate object. See for instance figure 1.10 where parts of the text –the title of a story– are written in 14 pt Times-Roman, but other parts –the names of the main characters– are written in 12 pt Courier with red text on a light gray background.

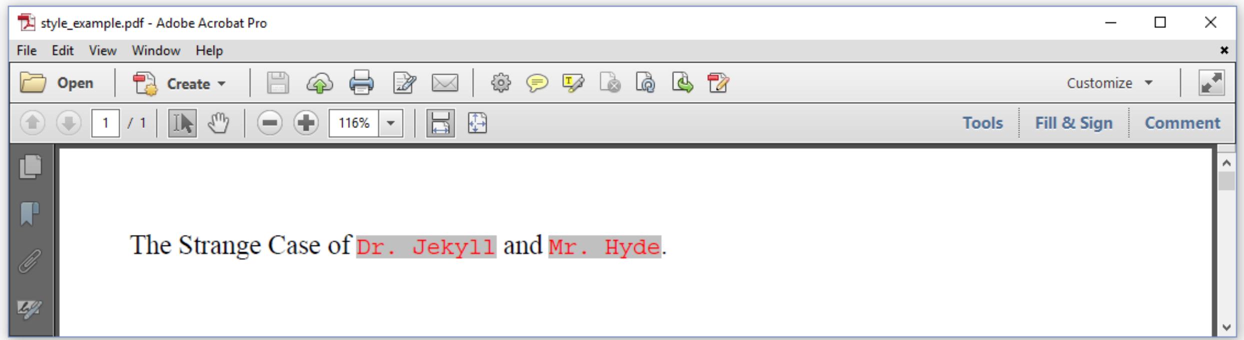



**Figure 1.10: reusing styles**

We could define the font family, font size, font color and background for each separate `Text` object that is added to the title `Paragraph`, but in the ReusingStyles[13] example, we use the `Style` object to define all the different styles at once.

[13] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1889-c01e10_reusingstyles.java

```
Style normal = new Style();
PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
normal.setFont(font).setFontSize(14);
Style code = new Style();
PdfFont monospace = PdfFontFactory.createFont(FontConstants.COURIER);
code.setFont(monospace).setFontColor(Color.RED)
        .setBackgroundColor(Color.LIGHT_GRAY);
Paragraph p = new Paragraph();
p.add(new Text("The Strange Case of ").addStyle(normal));
p.add(new Text("Dr. Jekyll").addStyle(code));
p.add(new Text(" and ").addStyle(normal));
p.add(new Text("Mr. Hyde").addStyle(code));
p.add(new Text(".").addStyle(normal));
document.add(p);
```

In line 1-3, we define a `normal` style; in line 4-7, we define a `code` style –Courier is often used when introducing code snippets in text. In line 8-13, we compose a `Paragraph` using different `Text` objects. We set the style of each of these `Text` objects to either `normal`, or `code`.

The `Style` object is a subclass of the abstract `ElementPropertyContainer` class, which is the superclass of all the building blocks we are going to discuss in the next handful of chapters. It contains a number of setters and getters for numerous properties such as fonts, colors, borders, dimensions, and positions. You can use the `addStyle()` method on every `AbstractElement` subclass to set these properties in one go.

Being able to combine different properties in one class, is one of the many new features in iText 7 that can save you many lines of code when compared to iText 5.

The `Style` class is about much more than fonts. You can even use it to define padding and margin values for `BlockElement` building blocks. But let's not get ahead of ourselves, the `BlockElement` class will be discussed in chapters 4 and 5.

# Summary

In this chapter, we've introduced the `PdfFont` class and we talked about font programs, embedding fonts and using different encodings. This allowed us to show the title of a short story by Robert Louis Stevenson in different languages: English, Czech, Russian, and Korean. We also looked at font properties such as font size, font color, and rendering mode. We even discovered how to mimic styles in case we can't find the font program to render text in italic or bold.

There's much more that could be said about fonts, but we'll leave that for a separate tutorial. In the next chapter, we'll create a PDF with the full story while we discuss the `RootElement` implementations `Document` and `Canvas`.

# Chapter 2: Working with the RootElement

Throughout this tutorial, we'll be creating PDF documents by adding `BlockElement` and `Image` objects to a `RootElement`, an abstract class that is subclassed by the `Document` object and the `Canvas` object. In the previous chapter, we've already used the `Document` class; in this chapter, we'll take a closer look at both the `Canvas` and the `Document` class.

- `Document` is the default root element when creating a self-sufficient PDF. It manages high-level operations such as setting page size and rotation, adding elements, and writing text at specific coordinates. It has no knowledge of the actual PDF concepts and syntax. A Document's rendering behavior can be modified by extending the `DocumentRenderer` class and setting an instance of this custom renderer with the `setRenderer()` method.
- `Canvas` is used for adding `BlockElement` and `Image` content inside a specific rectangle defined using absolute positions on a `PdfCanvas`. `Canvas` has no knowledge of the concept of a page and content that doesn't fit the rectangle will be lost. This class acts as a bridge between the high-level layout API and the low-level kernel API.

Lets's start with some `Canvas` examples.

## Using Canvas to add content inside a Rectangle

In figure 2.1, we see a rectangle drawn using the low-level API. Inside this rectangle, we've added some text. This text was added using the `Canvas` object.

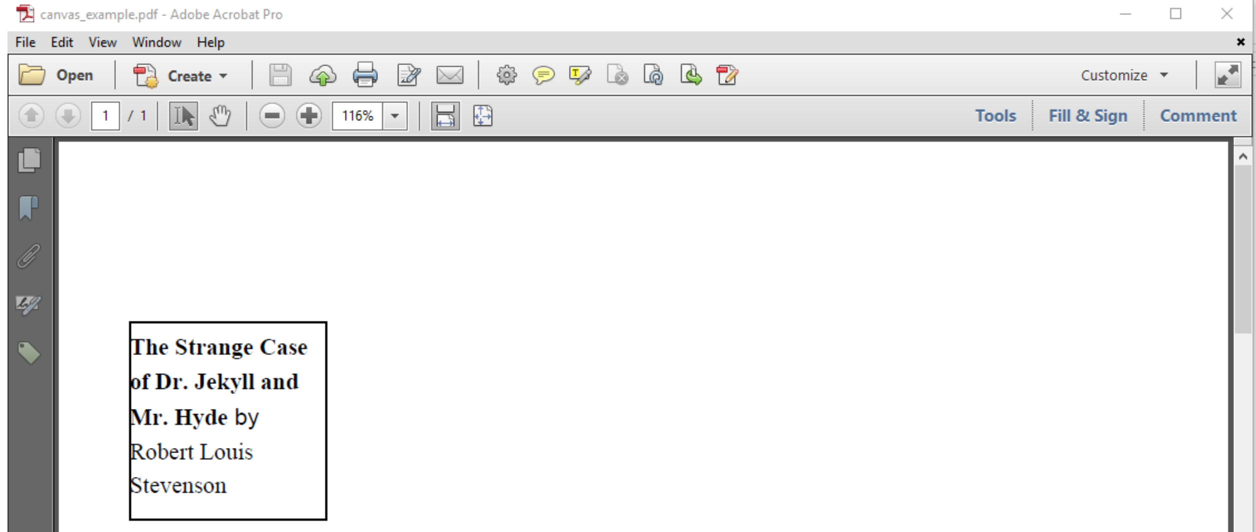



**Figure 2.1: Adding text inside a rectangle**

The CanvasExample[14] shows how it's done.

```
PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
PdfPage page = pdf.addNewPage();
PdfCanvas pdfCanvas = new PdfCanvas(page);
Rectangle rectangle = new Rectangle(36, 650, 100, 100);
pdfCanvas.rectangle(rectangle);
pdfCanvas.stroke();
Canvas canvas = new Canvas(pdfCanvas, pdf, rectangle);
PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
PdfFont bold = PdfFontFactory.createFont(FontConstants.TIMES_BOLD);
Text title =
    new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
Text author = new Text("Robert Louis Stevenson").setFont(font);
Paragraph p = new Paragraph().add(title).add(" by ").add(author);
canvas.add(p);
pdf.close();
```

Let's examine what happens in this code snippet:

- Line 1: we define a `PdfDocument`,
- Line 2: we don't use a `Document` object, so we have to create each `PdfPage` in our own code,
- Line 3: we use this `PdfPage` to create a `PdfCanvas`,
- Line 4: we define a rectangle,

[14] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1891-c02e01_canvasexample.java

- Line 5-6: we draw the rectangle using the low-level API,
- Line 7: we create a `Canvas` object using the `PdfPage`, the `PdfDocument` and the rectangle,
- Line 8-13: we create a `Paragraph`; this code is identical to what we had in the previous chapter,
- Line 14: we add the `Paragraph` to the `Canvas`.
- Line 15: we close the `PdfDocument`.

Looking at this example, it's not hard to understand the use case. Suppose that you need to add content on a specific page at a specific rectangular location. You create a `Canvas` object passing that page and that rectangle as a parameter, and you can add that content to that object. The content will be rendered inside the boundaries of that rectangle.

It is important to understand that all the content that doesn't fit the rectangle will be cut. See figure 2.2.

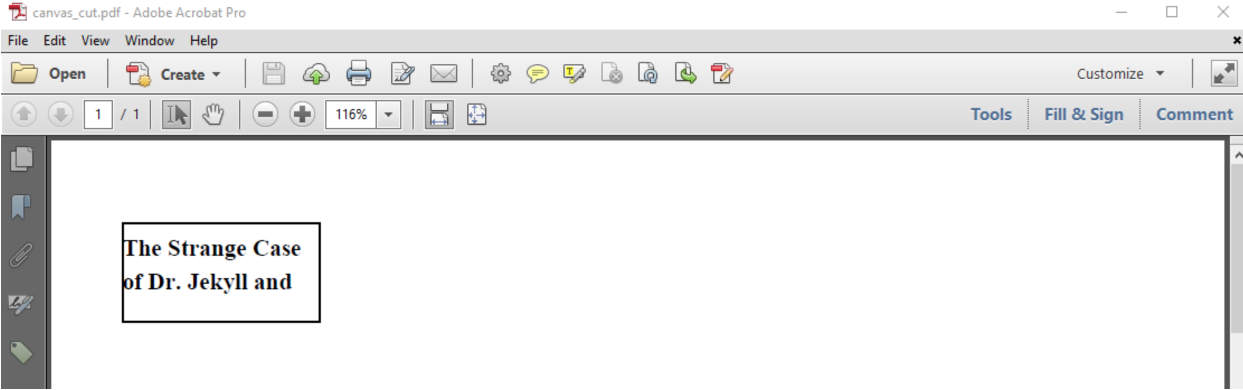



**Figure 2.2: Adding text that doesn't fit a rectangle**

In the CanvasCut[15] example, we add the same content to a smaller rectangle.

```
1 Rectangle rectangle = new Rectangle(36, 750, 100, 50);
2 Canvas canvas = new Canvas(pdfCanvas, pdf, rectangle);
3 PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
4 PdfFont bold = PdfFontFactory.createFont(FontConstants.TIMES_BOLD);
5 Text title =
6     new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
7 Text author = new Text("Robert Louis Stevenson").setFont(font);
8 Paragraph p = new Paragraph().add(title).add(" by ").add(author);
9 canvas.add(p);
```

In this snippet, we add the exact same content as before, but instead of `new Rectangle(36, 650, 100, 100)`, we reduced the height from 100 to 50: `new Rectangle(36, 750, 100, 50)`. As a result,

[15] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1892-c02e02_canvascut.java

the text no longer fits the rectangle. The part that says "Mr. Hyde by Robert Louis Stevenson" got lost. No exception gets thrown because this is expected behavior.

iText 7.0.0 was rewritten from scratch. We've waited with the release of the first iText 7 version until we were 99% sure of the API. We wanted to avoid significant changes to the API in later versions. Nevertheless, we're constantly improving the library, hence you will notice that some functionality described in the tutorials will only work in the current or SNAPSHOT version of iText 7. Whenever this is the case, You'll see a "Warning" call out like this one. The `CanvasCut` example we've just discussed won't work as described in iText 7.0.0. You'll need iText 7.0.1 to get the behavior described in this tutorial.

Text getting cut without warning isn't always what you want. In some cases, you need to know if the content fit the rectangle or not. For instance, in figure 2.3, we have defined a larger rectangle to which we've added the `Paragraph` as many times as possible.

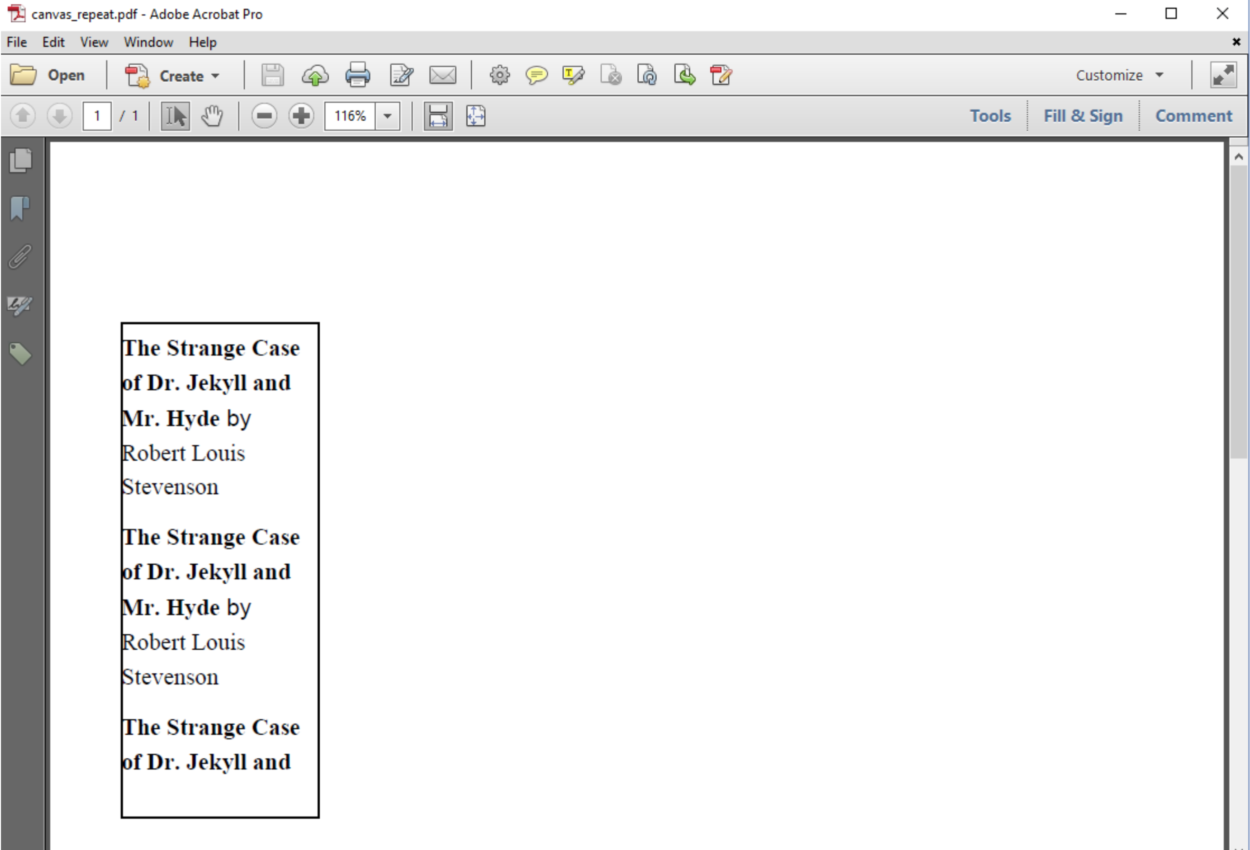



**Figure 2.3: Filling a rectangle with text**

We've added the `Paragraph` three times, because we can fit it inside the rectangle almost two and a half times. How did we know this? Let's take a look at the CanvasRepeat[16] example.

[16] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1893-c02e03_canvasrepeat.java

First we extended the `CanvasRenderer`:

```
class MyCanvasRenderer extends CanvasRenderer {
    protected boolean full = false;

    private MyCanvasRenderer(Canvas canvas) {
        super(canvas);
    }

    @Override
    public void addChild(IRenderer renderer) {
        super.addChild(renderer);
        full = Boolean.TRUE.equals(getPropertyAsBoolean(Property.FULL));
    }

    public boolean isFull() {
        return full;
    }
}
```

We introduce a member-variable `full` that indicates if the rectangle was completely filled or not. Each time a child is added to the renderer, we check the status of the `FULL` property. This status can be `null`, `false` or `true`. If it's `true`, there is no more space left to add content. We also added an `isFull()` method for our convenience.

```
Rectangle rectangle = new Rectangle(36, 500, 100, 250);
Canvas canvas = new Canvas(pdfCanvas, pdf, rectangle);
MyCanvasRenderer renderer = new MyCanvasRenderer(canvas);
canvas.setRenderer(renderer);
PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
PdfFont bold = PdfFontFactory.createFont(FontConstants.TIMES_BOLD);
Text title =
    new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
Text author = new Text("Robert Louis Stevenson").setFont(font);
Paragraph p = new Paragraph().add(title).add(" by ").add(author);
while (!renderer.isFull())
    canvas.add(p);
```

The `Rectangle` we define in line 1 is larger than what we had before. Line 3 and 4 are new. We create an instance of our custom renderer and we declare this renderer to the `Canvas` object. In line 11 and 12, we add the `Paragraph` as many times as possible as long as the `Canvas` we've defined isn't completely full.

One might wonder why we are adding the border of the rectangle using the low-level `rectangle` menu. The abstract `RootElement` extends the abstract `ElementPropertyContainer` class. The `ElementPropertyContainer` class defines methods such as `setBorder()` and `setBackgroundColor()`, but these methods can't be used because setting a border or a background isn't implemented for `Canvas`, nor for `Document`. Not every method defined in `ElementPropertyContainer` makes sense for all of its subclasses. For instance: it doesn't make sense to implement the `setFont()` method for an `Image` object. You can check which methods are implemented for the `Canvas` and `Document` class in Appendix C[17].

In figure 2.4, we created a document with two pages, but there's something special about it: we added content under the existing content of the first page *after* we added content to the second page.

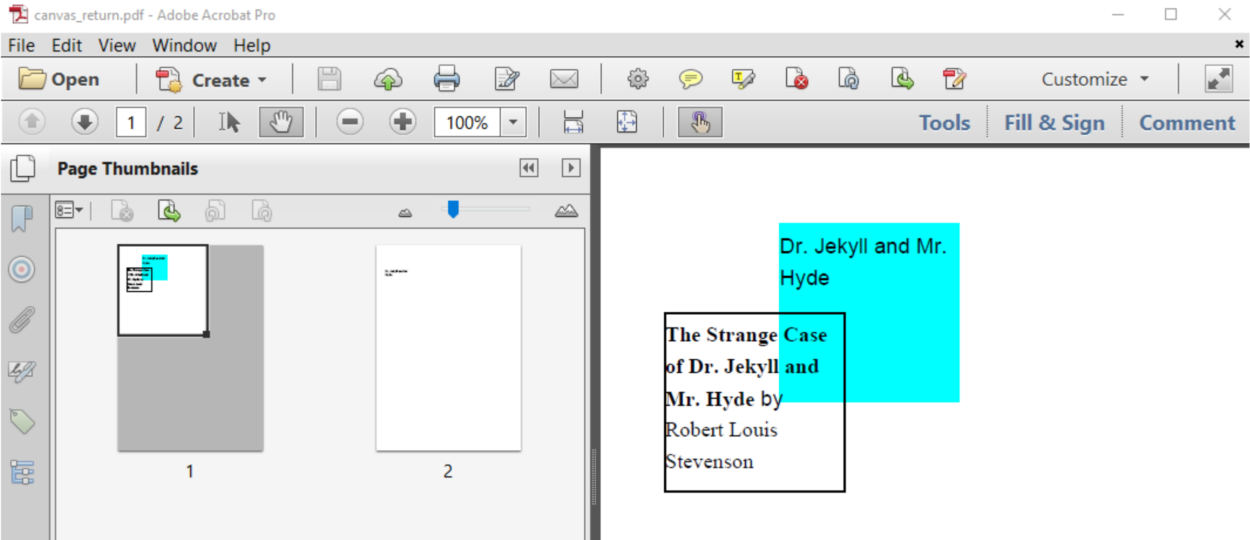



**Figure 2.4: Adding content to the previous page**

The first part of the code, is identical to what we had in the first example: we define a first page and a rectangle, we create a `Canvas` instance with this page and this rectangle. Then we define a `Paragraph` and we add this `Paragraph` to the canvas. The following code snippet taken from the CanvasReturn[18] example shows how we create a second page and add some content to that page.

```
1 PdfPage page2 = pdf.addNewPage();
2 PdfCanvas pdfCanvas2 = new PdfCanvas(page2);
3 Canvas canvas2 = new Canvas(pdfCanvas2, pdf, rectangle);
4 canvas2.add(new Paragraph("Dr. Jekyll and Mr. Hyde"));
```

We add a new page to the document with the `addNewPage()` method (line 1). We create a new `PdfCanvas` object with that page (line 2) and a new `Canvas` object using that new `PdfCanvas`, the

[17] http://developers.itextpdf.com/content/itext-7-building-blocks/c-rootelement-methods
[18] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1894-c02e04_canvasreturn.java

PdfDocument and the Rectangle we used for the first page (line 3). We add a Paragraph to that new Canvas.

This is pretty straightforward, but now look what happens next:

```
1  PdfPage page1 = pdf.getFirstPage();
2  PdfCanvas pdfCanvas1 = new PdfCanvas(
3      page1.newContentStreamBefore(), page1.getResources(), pdf);
4  rectangle = new Rectangle(100, 700, 100, 100);
5  pdfCanvas1.saveState()
6          .setFillColor(Color.CYAN)
7          .rectangle(rectangle)
8          .fill()
9          .restoreState();
10 Canvas canvas = new Canvas(pdfCanvas1, pdf, rectangle);
11 canvas.add(new Paragraph("Dr. Jekyll and Mr. Hyde"));
```

In line 1, we create a PdfPage instance for the first page using the getFirstPage() method.

The getFirstPage() method is a custom version of the getPage() method. The getPage() method allows you to get access to any page that was created before as long as the PdfDocument hasn't been closed.

In line 2 and 3, we create a PdfCanvas object using the following parameters:

- a PdfStream instance: a page consists of one or more content streams. In this case, we want to add content *under* the existing content, hence we use the newContentStreamBefore() method. If you want to add content *on top of* existing content, you should use the newContentStreamAfter() object. These methods create a new content stream, and add it to the page. You can also get access to existing content streams. The method getContentStreamCount() will tell you of how many content streams the page content consists. There's a getContentStream() method that allows you to get a specific content stream based on its *index*. There's also a getFirstContentStream() and a getLastContentStream() method.
- a PdfResources instance: the content stream on its own isn't sufficient to render a page. Each page refers to resources such as fonts and images. When adding content to that page, we'll need to reuse and update these resources.
- the PdfDocument instance: this is the low-level PdfDocument we're working with.

In line 4, we define a rectangle. We paint that rectangle in Cyan in lines 5 to 9. In line 10 and 11, we create a Canvas object to which we add a Paragraph.

Being able to go back to a previous page *and to add content to that page* is one of the new, powerful features in iText 7. The architecture of iText 5 didn't allow us to change the content of "completed" pages. This is one of the many reasons why we decided to rewrite iText from scratch.

So far, we have been using the `Canvas` class to add content to a `PdfCanvas`. In chapter 7, we'll discover another use case: you can also create a `Canvas` to add content to a `PdfFormXObject`. A *form XObject* is an object that is external to any page content stream. It represents a stream of PDF content that can be referred to more than once from the same page, or from different pages. It's a stream of reusable PDF syntax. The `Canvas` objects allows you to create that PDF syntax without any hassle.

It's high time that we create a PDF with the full Jekyll and Hyde story instead of merely adding the title and the author to a page. We'll use the `Document` class to achieve this.

# Converting text to PDF with the Document class

Figure 2.5 shows a text file with the full Jekyll and Hyde story: jekyll_hyde.txt[19]

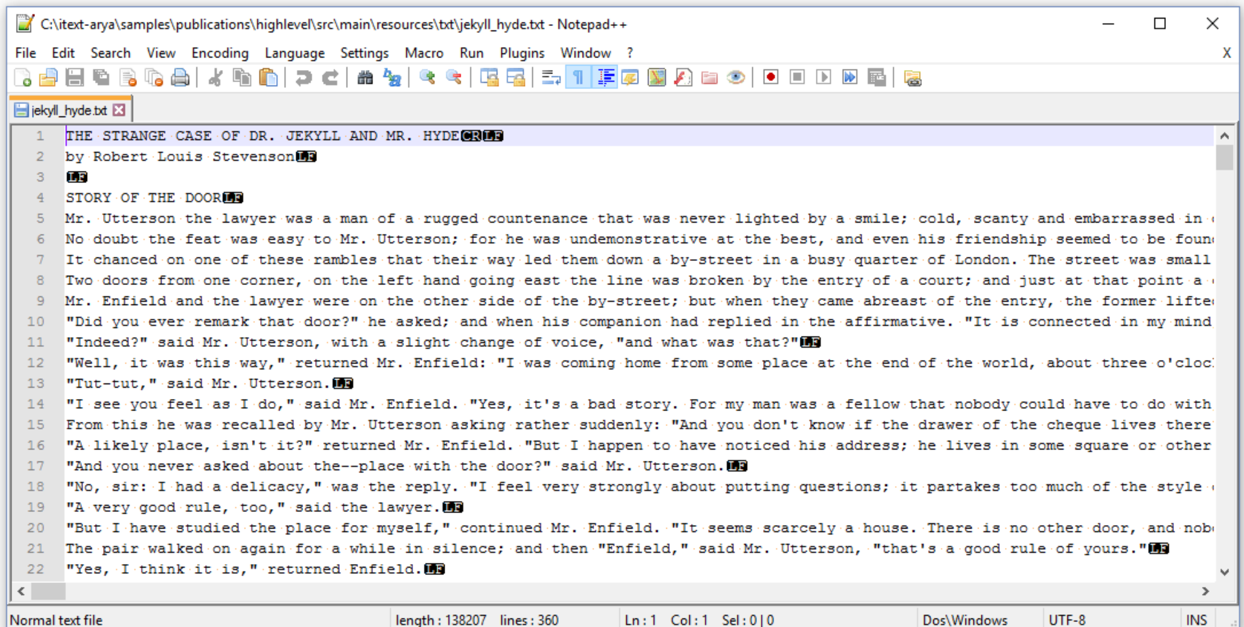
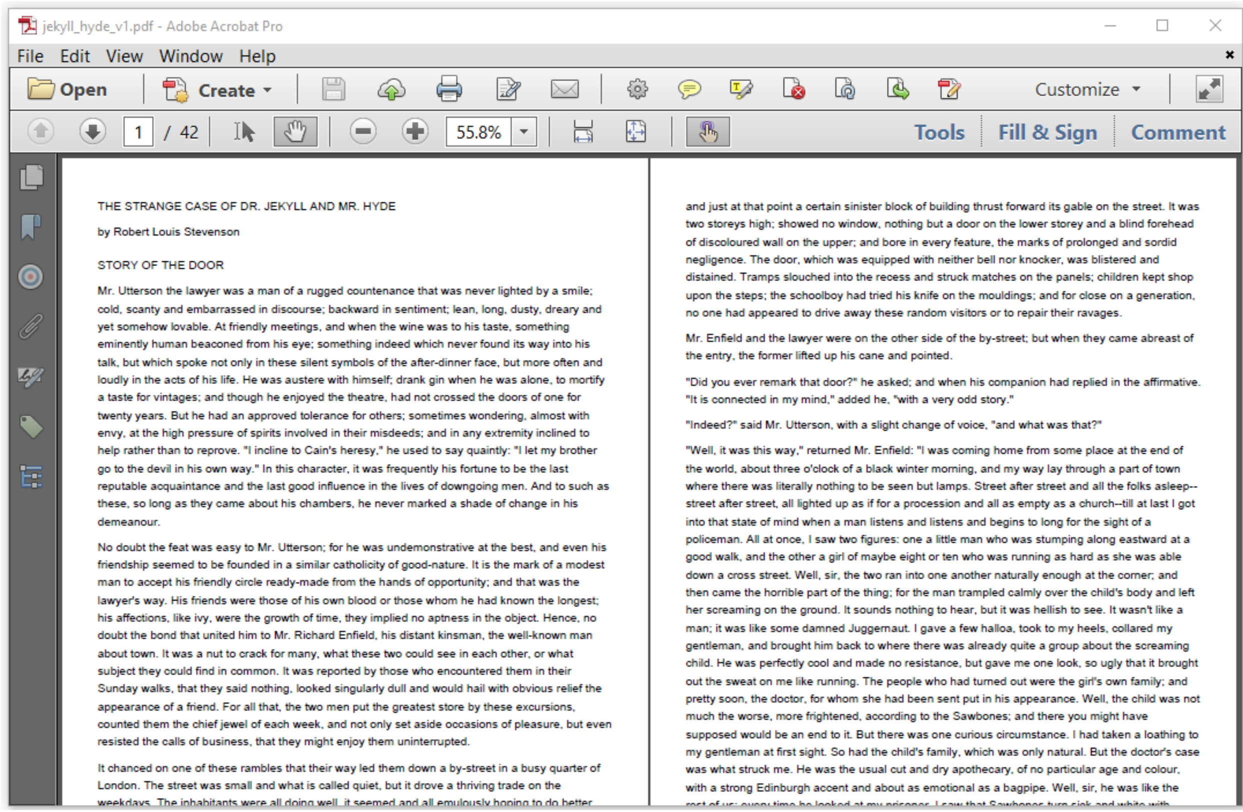



**Figure 2.5: Text file with the Jekyll and Hyde story**

We'll convert this txt file to a PDF multiple times in the next handful of examples. We'll start by creating the PDF shown in figure 2.6.

[19] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/highlevel/src/main/resources/txt/jekyll_hyde.txt




THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the

and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept shop upon the steps; the schoolboy had tried his knife on the mouldings; and for close on a generation, no one had appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the

**Figure 2.6: First attempt to convert txt to PDF**

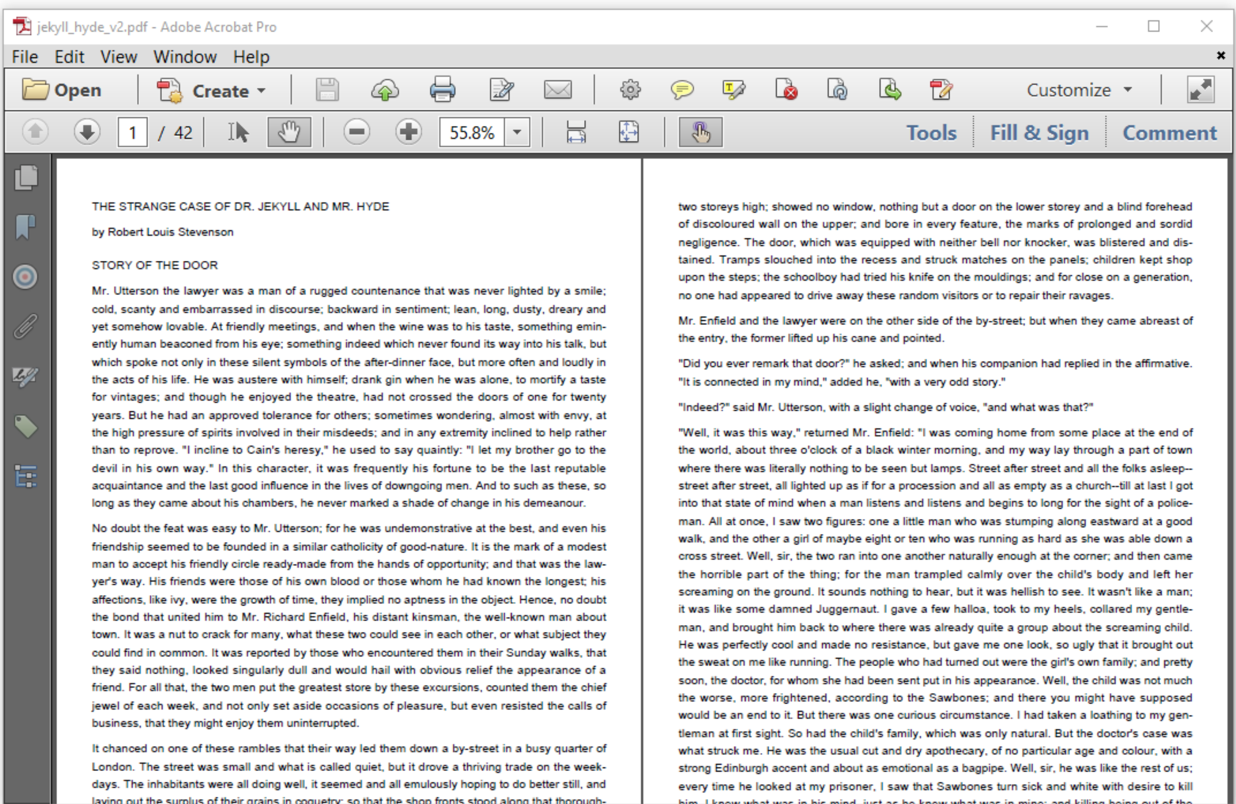
The JekyllHydeV1[20] example is very simple. You don't need any new functionality that hasn't been discussed before:

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  Document document = new Document(pdf);
3  BufferedReader br = new BufferedReader(new FileReader(SRC));
4  String line;
5  while ((line = br.readLine()) != null) {
6      document.add(new Paragraph(line));
7  }
8  document.close();
```

In line 1, we create the low-level `PdfDocument` object. In line 2, we create the high-level `Document` instance. We create a `BufferedReader` to read the txt file in line 3. We read every line in the text file in a loop in lines 4 to 7. In line 6, we wrap every line inside a `Paragraph` object, which we add to the `Document` object. In line 8. we close the document. The result is a 42-page PDF with the full story of "The Strange Case of Dr. Jekyll and Mr. Hyde."

[20] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1895-c02e05_jekyllhydev1.java

While this result is already nice, we can do better. The first thing that jumps to the eye in figure 2.7 is the fact that we changed the alignment. Instead of the default left alignment, the text is now justified on both sides of the page. If you take a closer look, you'll also notice that we've introduced hyphenation.



THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and

two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept shop upon the steps; the schoolboy had tried his knife on the mouldings; and for close on a generation, no one had appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the rest of us; every time he looked at my prisoner, I saw that Sawbones turn sick and white with desire to kill

**Figure 2.7: Second attempt to convert txt to PDF**

For the JekyllHydeV2[21] example, we copied the first example, and we added the following lines:

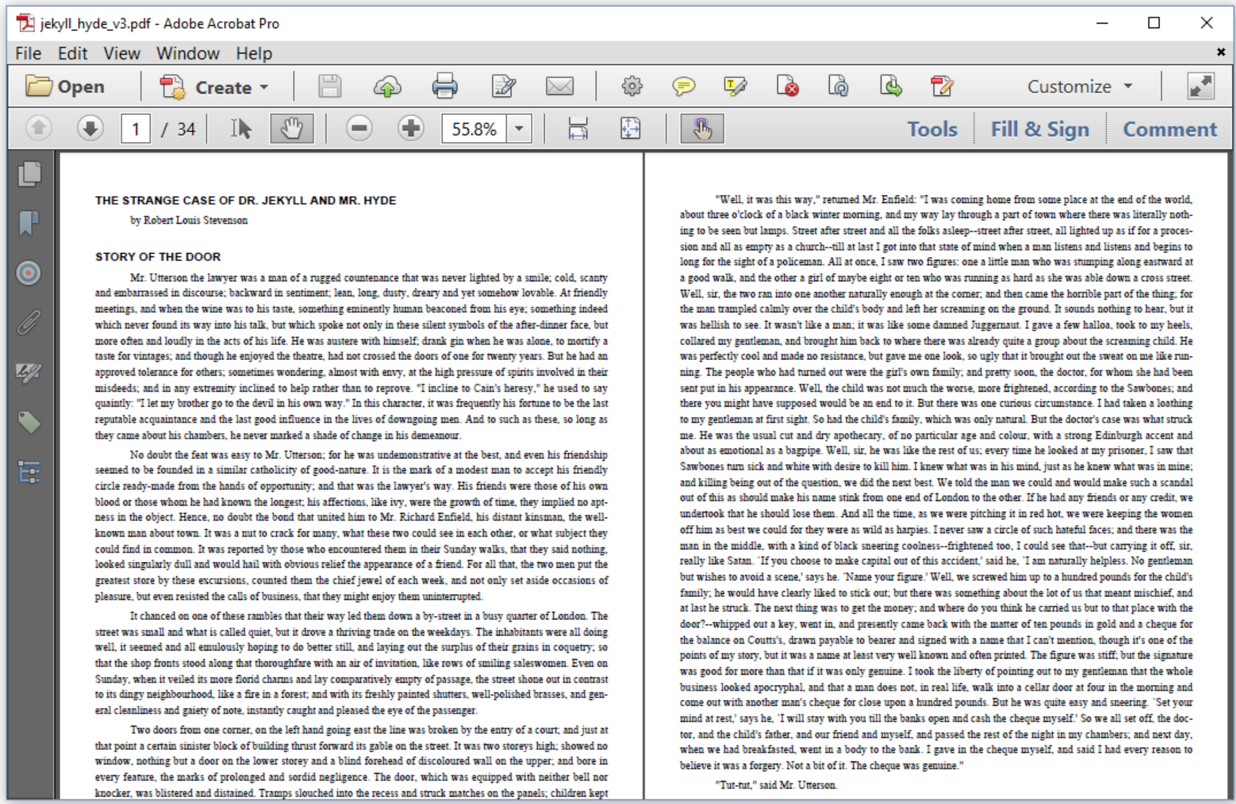
```
document.setTextAlignment(TextAlignment.JUSTIFIED)
    .setHyphenation(new HyphenationConfig("en", "uk", 3, 3));
```

We used the `setTextAlignment()` to change the default alignment at the `Document` level. We used the `setHyphenation()` method to define the hyphenation rules. In this case, we created a `HyphenationConfig` object to treat the text as British English. When splitting a word, we indicated that we want at least 3 characters before the hyphenation point and at least 3 characters after the hyphenation point. This means that the word "elephant" can't be hyphenated as "e-lephant" because "e" is shorter than 3 characters; we need to split the word like this instead: "ele-phant". The word "attitude" can't be hyphenated as "attitu-de" because "de" is shorter than 3 characters, in this case we need something like "atti-tude".

[21]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1896-c02e06_jekyllhydev2.java

Changing defaults at the `Document` level, such as the default alignment, the default hyphenation, or even the default font, wasn't possible in iText 5. You had to define all of these properties at the level of the separate building blocks. In iText 7, we introduced the inheritance of properties. The default font is still Helvetica, but we can now define a different font at the `Document` level.

Figure 2.8 shows our third attempt to convert the txt file to a PDF. We changed the font from 12 pt Helvetica to 11 pt Times-Roman. As a result, the page count was reduced from 42 pages to only 34.




THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and laying out the surplus of their grains in coquetry; so that the shop fronts stood along that thoroughfare with an air of invitation, like rows of smiling saleswomen. Even on Sunday, when it veiled its more florid charms and lay comparatively empty of passage, the street shone out in contrast to its dingy neighbourhood, like a fire in a forest; and with its freshly painted shutters, well-polished brasses, and general cleanliness and gaiety of note, instantly caught and pleased the eye of the passenger.

Two doors from one corner, on the left hand going east the line was broken by the entry of a court; and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the rest of us; every time he looked at my prisoner, I saw that Sawbones turn sick and white with desire to kill him. I knew what was in his mind, just as he knew what was in mine; and killing being out of the question, we did the next best. We told the man we could and would make such a scandal out of this as should make his name stink from one end of London to the other. If he had any friends or any credit, we undertook that he should lose them. And all the time, as we were pitching it in red hot, we were keeping the women off him as best we could for they were as wild as harpies. I never saw a circle of such hateful faces; and there was the man in the middle, with a kind of black sneering coolness--frightened too, I could see that--but carrying it off, sir, really like Satan. 'If you choose to make capital out of this accident,' said he, 'I am naturally helpless. No gentleman but wishes to avoid a scene,' says he. 'Name your figure.' Well, we screwed him up to a hundred pounds for the child's family; he would have clearly liked to stick out; but there was something about the lot of us that meant mischief, and at last he struck. The next thing was to get the money; and where do you think he carried us but to that place with the door?--whipped out a key, went in, and presently came back with the matter of ten pounds in gold and a cheque for the balance on Coutts's, drawn payable to bearer and signed with a name that I can't mention, though it's one of the points of my story, but it was a name at least very well known and often printed. The figure was stiff; but the signature was good for more than that if it was only genuine. I took the liberty of pointing out to my gentleman that the whole business looked apocryphal, and that a man does not, in real life, walk into a cellar door at four in the morning and come out with another man's cheque for close upon a hundred pounds. But he was quite easy and sneering. 'Set your mind at rest,' says he, 'I will stay with you till the banks open and cash the cheque myself.' So we all set off, the doctor, and the child's father, and our friend and myself, and passed the rest of the night in my chambers; and next day, when we had breakfasted, went in a body to the bank. I gave in the cheque myself, and said I had every reason to believe it was a forgery. Not a bit of it. The cheque was genuine."

"Tut-tut," said Mr. Utterson.

**Figure 2.8: Third attempt to convert txt to PDF**

When we look at the JekyllHydeV3[22] example, we see that two different fonts are used:

[22]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1897-c02e07_jekyllhydev3.java

```
1  Document document = new Document(pdf);
2  PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
3  PdfFont bold = PdfFontFactory.createFont(FontConstants.HELVETICA_BOLD);
4  document.setTextAlignment(TextAlignment.JUSTIFIED)
5      .setHyphenation(new HyphenationConfig("en", "uk", 3, 3))
6      .setFont(font)
7      .setFontSize(11);
```

Times-Roman is used as the default font, but we also define Helvetica-Bold for the titles. The txt file was conceived in such a way that the first line of the text file is the title of the book. Every other title in the story is preceded by an empty line. Every line that isn't a title, is a full paragraph. Knowing this, we can adapt the loop that reads the text file line by line.

```
1  BufferedReader br = new BufferedReader(new FileReader(SRC));
2  String line;
3  Paragraph p;
4  boolean title = true;
5  while ((line = br.readLine()) != null) {
6      p = new Paragraph(line);
7      p.setKeepTogether(true);
8      if (title) {
9          p.setFont(bold).setFontSize(12);
10         title = false;
11     }
12     else {
13         p.setFirstLineIndent(36);
14     }
15     if (line.isEmpty()) {
16         p.setMarginBottom(12);
17         title = true;
18     }
19     else {
20         p.setMarginBottom(0);
21     }
22     document.add(p);
23 }
```

This code snippet is a tad more complex than what we had before, but let's walk through it step by step:

- We introduce a Boolean `title` (line 4) which we initalize as `true` because we know that the first line in the text file is a title. We create a `Paragraph` for each line (line 6) and we use

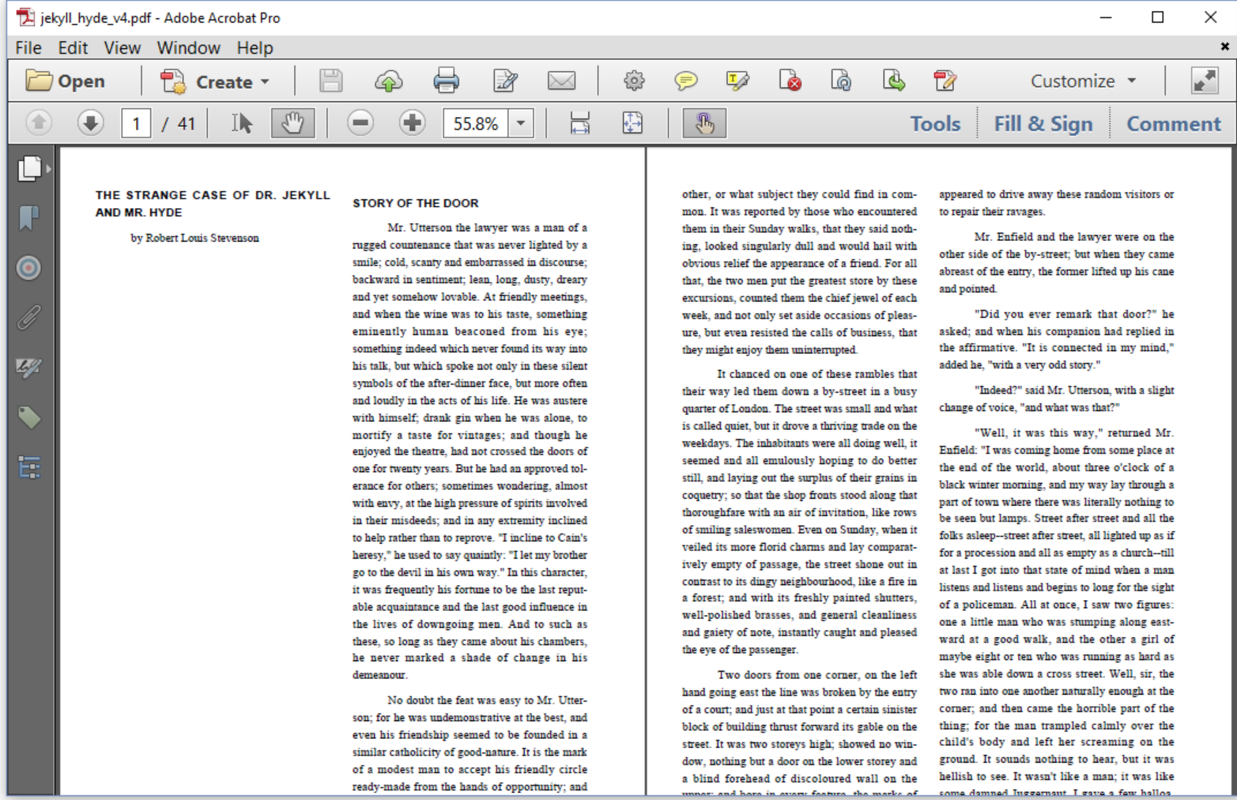
the `setKeepTogether()` method because we don't want iText to distribute paragraphs over different pages (line 7). If a `Paragraph` doesn't fit the current page, it will be forwarded to the next page unless the `Paragraph` doesn't fit the next page either. In that case will be split anyway: part of it will be added to the current page and the rest will be forwarded to the next page –or pages.

- If value of `title` is `true`, we change the default font that was defined at the `Document` level as 11 pt Times-Roman to 12 pt Helvetica-Bold. We know that the next line in the txt file will be normal content, so we set the value of `title` to `false` (line 9-11). For normal lines, we change the indentation of the first line so that we can easily distinguish the different paragraphs in the text (line 12-14).
- If the current line is an empty `String`, we define a bottom margin of `12` (line 16) and we change the value of `title` back to `true` (line 17), because we know that the next line will be a title; for all other the lines, we reduce the bottom margin of the `Paragraph` to `0` (line 20).
- Once all the properties for the `Paragraph` are set, we add it to the `Document` (line 22).

As you could tell from figure 2.8, iText has rendered the text to PDF page by page in quite a nice way. Now suppose that we want to render the text in two columns, organized side by side on one page. In that case, we need to introduce a `DocumentRenderer` instance.

## Changing the Document renderer

The text in figure 2.8 is rendered using exactly the same `Document` defaults and exactly the same `Paragraph` properties as in the previous example. There's one major difference: the text is now rendered in two columns per page.



THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and

other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and laying out the surplus of their grains in coquetry; so that the shop fronts stood along that thoroughfare with an air of invitation, like rows of smiling saleswomen. Even on Sunday, when it veiled its more florid charms and lay comparatively empty of passage, the street shone out in contrast to its dingy neighbourhood, like a fire in a forest; and with its freshly painted shutters, well-polished brasses, and general cleanliness and gaiety of note, instantly caught and pleased the eye of the passenger.

Two doors from one corner, on the left hand going east the line was broken by the entry of a court; and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the

appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like

**Figure 2.9: Rendering the text in two columns**

To achieve this, we used the `ColumnDocumentRenderer` class. This is a subclass of the `DocumentRenderer` class that is used by default. The JekyllHydeV4[23] example explains how the `ColumnDocumentRenderer` is created and applied.

```
1  float offSet = 36;
2  float gutter = 23;
3  float columnWidth = (PageSize.A4.getWidth() - offSet * 2) / 2 - gutter;
4  float columnHeight = PageSize.A4.getHeight() - offSet * 2;
5  Rectangle[] columns = {
6      new Rectangle(offSet, offSet, columnWidth, columnHeight),
7      new Rectangle(
8          offSet + columnWidth + gutter, offSet, columnWidth, columnHeight)};
9  document.setRenderer(new ColumnDocumentRenderer(document, columns));
```

We define an array of `Recangle` objects, and we use that array to create a `ColumnDocumentRenderer` object. We use the `setRenderer()` method to tell the `Document` to use this renderer instead of the default `DocumentRenderer` instance.

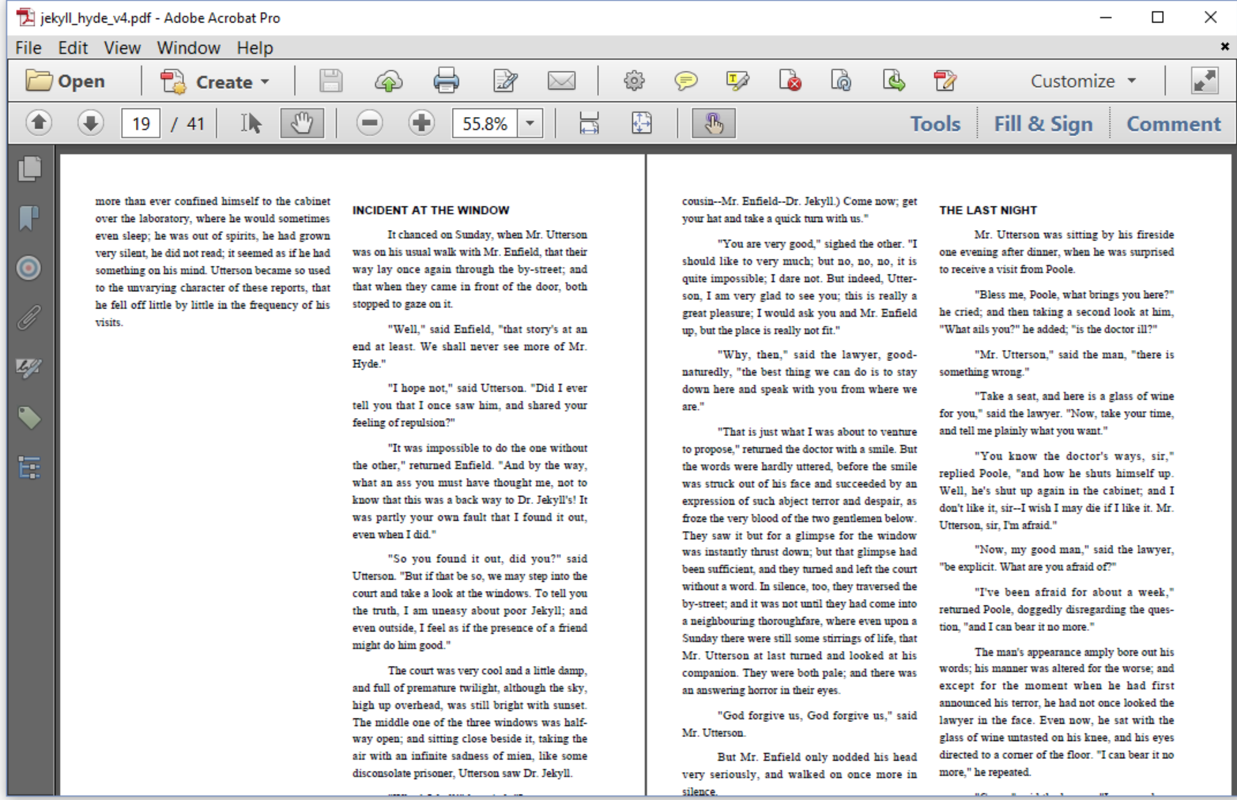
[23]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1898-c02e08_jekyllhydev4.java

If we wanted to organize content in columns in iText 5, we needed to use the `ColumnText` object. In iText 2, there was a `MultiColumnText` object that reduced the amount of code that needed to be written to distribute the code over different columns, but this class was removed in iText 5 because of the lack of robustness of `MultiColumnText`. With the `ColumnDocumentRenderer`, developers now have a reliable way to create columns without having to write as much code as was needed in iText 5.

While we were at it, we applied a small change to the code that parses the text:

```
1  BufferedReader br = new BufferedReader(new FileReader(SRC));
2  String line;
3  Paragraph p;
4  boolean title = true;
5  AreaBreak nextArea = new AreaBreak(AreaBreakType.NEXT_AREA);
6  while ((line = br.readLine()) != null) {
7      p = new Paragraph(line);
8      if (title) {
9          p.setFont(bold).setFontSize(12);
10         title = false;
11     }
12     else {
13         p.setFirstLineIndent(36);
14     }
15     if (line.isEmpty()) {
16         document.add(nextArea);
17         title = true;
18     }
19     document.add(p);
20 }
```

In line 5, we create an `AreaBreak` object. This is a layout object that terminates the current content area and creates a new one. In this case, we create an `AreaBreak` of type `NEXT_AREA` and we introduce it before the start of every new chapter. The effect of this area break is shown in figure 2.10.




more than ever confined himself to the cabinet over the laboratory, where he would sometimes even sleep; he was out of spirits, he had grown very silent, he did not read; it seemed as if he had something on his mind. Utterson became so used to the unvarying character of these reports, that he fell off little by little in the frequency of his visits.

INCIDENT AT THE WINDOW

It chanced on Sunday, when Mr. Utterson was on his usual walk with Mr. Enfield, that their way lay once again through the by-street; and that when they came in front of the door, both stopped to gaze on it.

"Well," said Enfield, "that story's at an end at least. We shall never see more of Mr. Hyde."

"I hope not," said Utterson. "Did I ever tell you that I once saw him, and shared your feeling of repulsion?"

"It was impossible to do the one without the other," returned Enfield. "And by the way, what an ass you must have thought me, not to know that this was a back way to Dr. Jekyll's! It was partly your own fault that I found it out, even when I did."

"So you found it out, did you?" said Utterson. "But if that be so, we may step into the court and take a look at the windows. To tell you the truth, I am uneasy about poor Jekyll; and even outside, I feel as if the presence of a friend might do him good."

The court was very cool and a little damp, and full of premature twilight, although the sky, high up overhead, was still bright with sunset. The middle one of the three windows was half-way open; and sitting close beside it, taking the air with an infinite sadness of mien, like some disconsolate prisoner, Utterson saw Dr. Jekyll.

cousin--Mr. Enfield--Dr. Jekyll.) Come now; get your hat and take a quick turn with us."

"You are very good," sighed the other. "I should like to very much; but no, no, no, it is quite impossible; I dare not. But indeed, Utterson, I am very glad to see you; this is really a great pleasure; I would ask you and Mr. Enfield up, but the place is really not fit."

"Why, then," said the lawyer, good-naturedly, "the best thing we can do is to stay down here and speak with you from where we are."

"That is just what I was about to venture to propose," returned the doctor with a smile. But the words were hardly uttered, before the smile was struck out of his face and succeeded by an expression of such abject terror and despair, as froze the very blood of the two gentlemen below. They saw it but for a glimpse for the window was instantly thrust down; but that glimpse had been sufficient, and they turned and left the court without a word. In silence, too, they traversed the by-street; and it was not until they had come into a neighbouring thoroughfare, where even upon a Sunday there were still some stirrings of life, that Mr. Utterson at last turned and looked at his companion. They were both pale; and there was an answering horror in their eyes.

"God forgive us, God forgive us," said Mr. Utterson.

But Mr. Enfield only nodded his head very seriously, and walked on once more in silence.

THE LAST NIGHT

Mr. Utterson was sitting by his fireside one evening after dinner, when he was surprised to receive a visit from Poole.

"Bless me, Poole, what brings you here?" he cried; and then taking a second look at him, "What ails you?" he added; "is the doctor ill?"

"Mr. Utterson," said the man, "there is something wrong."

"Take a seat, and here is a glass of wine for you," said the lawyer. "Now, take your time, and tell me plainly what you want."

"You know the doctor's ways, sir," replied Poole, "and how he shuts himself up. Well, he's shut up again in the cabinet; and I don't like it, sir--I wish I may die if I like it. Mr. Utterson, sir, I'm afraid."

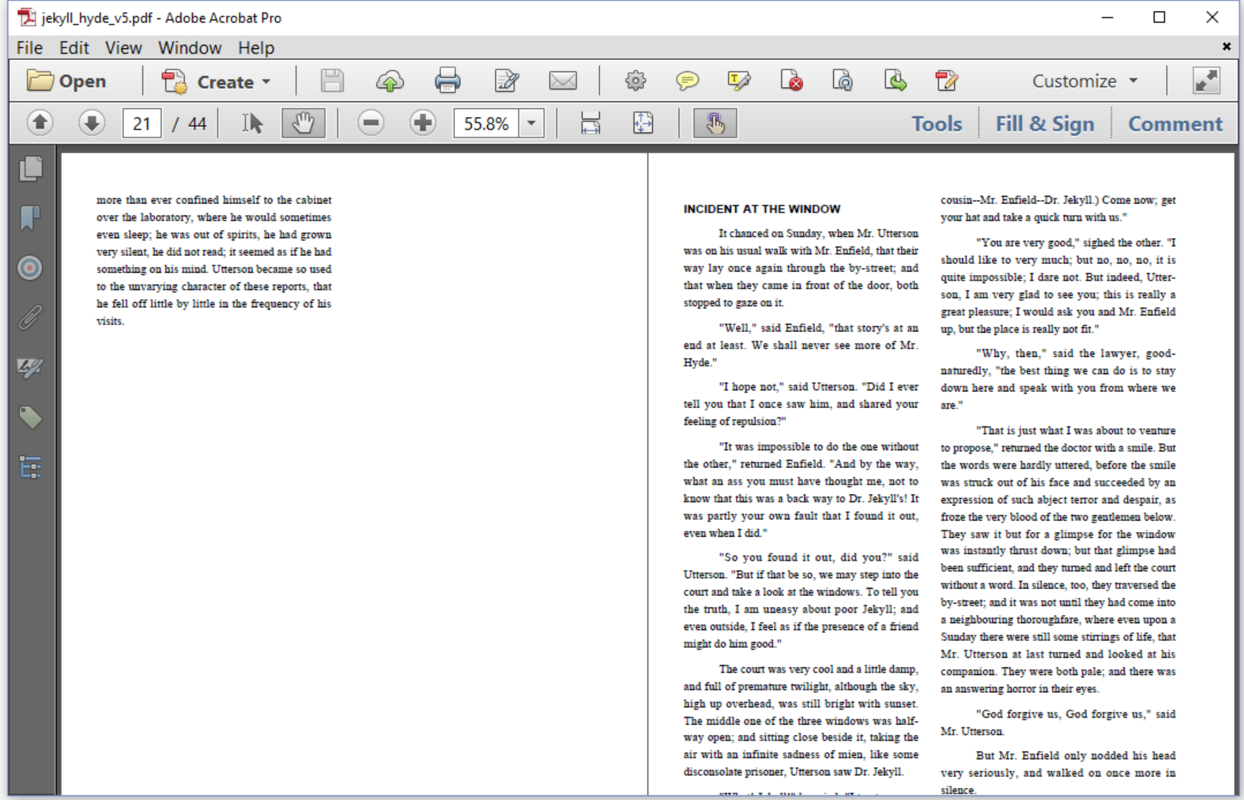
"Now, my good man," said the lawyer, "be explicit. What are you afraid of?"

"I've been afraid for about a week," returned Poole, doggedly disregarding the question, "and I can bear it no more."

The man's appearance amply bore out his words; his manner was altered for the worse; and except for the moment when he had first announced his terror, he had not once looked the lawyer in the face. Even now, he sat with the glass of wine untasted on his knee, and his eyes directed to a corner of the floor. "I can bear it no more," he repeated.

**Figure 2.10: The effect of an AreaBreak of type NEXT_AREA**

Without the `AreaBreak`, the chapter "INCIDENT AT THE WINDOW" would have started in the left column of page 19, right after the content of the previous chapter. By introducing the `AreaBreak`, the new chapter now starts in a new column. If we had used an `AreaBreak` of type `NEXT_PAGE`, a new page would have been started; see figure 2.11.




more than ever confined himself to the cabinet over the laboratory, where he would sometimes even sleep; he was out of spirits, he had grown very silent, he did not read; it seemed as if he had something on his mind. Utterson became so used to the unvarying character of these reports, that he fell off little by little in the frequency of his visits.

INCIDENT AT THE WINDOW

It chanced on Sunday, when Mr. Utterson was on his usual walk with Mr. Enfield, that their way lay once again through the by-street; and that when they came in front of the door, both stopped to gaze on it.

"Well," said Enfield, "that story's at an end at least. We shall never see more of Mr. Hyde."

"I hope not," said Utterson. "Did I ever tell you that I once saw him, and shared your feeling of repulsion?"

"It was impossible to do the one without the other," returned Enfield. "And by the way, what an ass you must have thought me, not to know that this was a back way to Dr. Jekyll's! It was partly your own fault that I found it out, even when I did."

"So you found it out, did you?" said Utterson. "But if that be so, we may step into the court and take a look at the windows. To tell you the truth, I am uneasy about poor Jekyll; and even outside, I feel as if the presence of a friend might do him good."

The court was very cool and a little damp, and full of premature twilight, although the sky, high up overhead, was still bright with sunset. The middle one of the three windows was half-way open; and sitting close beside it, taking the air with an infinite sadness of mien, like some disconsolate prisoner, Utterson saw Dr. Jekyll.

cousin--Mr. Enfield--Dr. Jekyll.) Come now; get your hat and take a quick turn with us."

"You are very good," sighed the other. "I should like to very much; but no, no, no, it is quite impossible; I dare not. But indeed, Utterson, I am very glad to see you; this is really a great pleasure; I would ask you and Mr. Enfield up, but the place is really not fit."

"Why, then," said the lawyer, good-naturedly, "the best thing we can do is to stay down here and speak with you from where we are."

"That is just what I was about to venture to propose," returned the doctor with a smile. But the words were hardly uttered, before the smile was struck out of his face and succeeded by an expression of such abject terror and despair, as froze the very blood of the two gentlemen below. They saw it but for a glimpse for the window was instantly thrust down; but that glimpse had been sufficient, and they turned and left the court without a word. In silence, too, they traversed the by-street; and it was not until they had come into a neighbouring thoroughfare, where even upon a Sunday there were still some stirrings of life, that Mr. Utterson at last turned and looked at his companion. They were both pale; and there was an answering horror in their eyes.

"God forgive us, God forgive us," said Mr. Utterson.

But Mr. Enfield only nodded his head very seriously, and walked on once more in silence.

**Figure 2.11: The effect of an AreaBreak of type NEXT_PAGE**

In the JekyllHydeV5[24] example, we changed a single line:

```
1 AreaBreak nextPage = new AreaBreak(AreaBreakType.NEXT_PAGE);
```

Instead of skipping to the next column, iText now skips to the next page.

By default, the newly created page will have the same page size as the current page. If you want iText to create a page of another size, you can use the constructor that accepts a `PageSize` object as a parameter. For instance: `new AreaBreak(PageSize.A3)`.

There's also an `AreaBreak` of type `LAST_PAGE`. This `AreaBreakType` is to be used when switching between different renderers.

# Switching between different renderers

Figure 2.12 shows a document for which we use the default `DocumentRenderer` for the first page. Starting with the second page, we introduce a `ColumnDocumentRenderer` with two colums.

[24]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1899-c02e09_jekyllhydev5.java

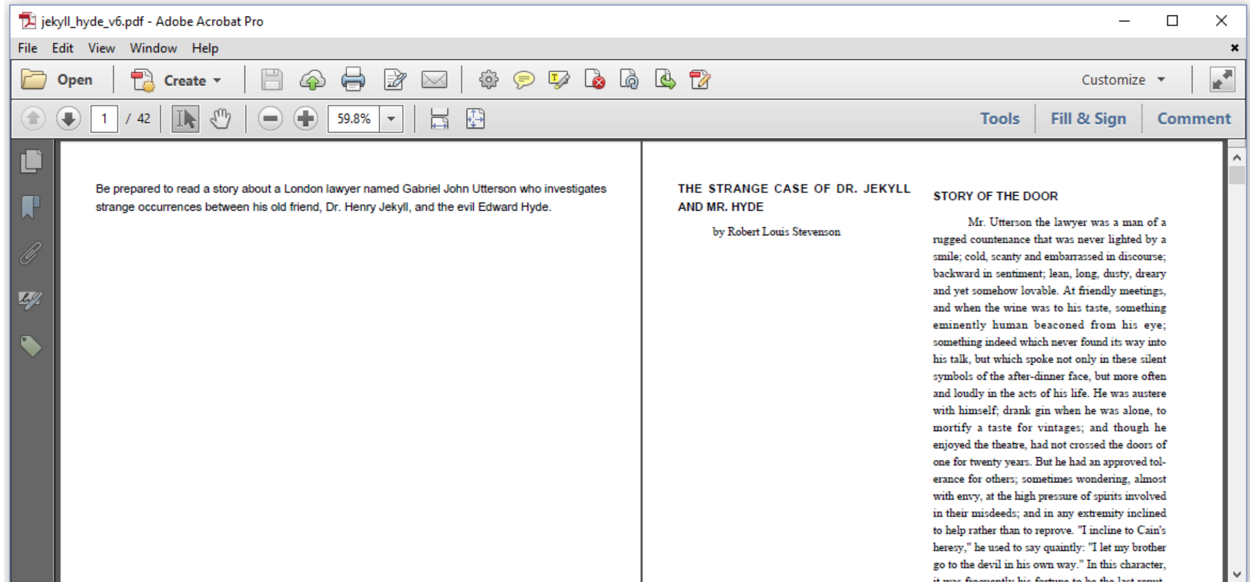



**Figure 2.12: Different renderers in the same document**

If we look closely at the JekyllHydeV6[25] example, we see that we swith renderers two times.

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
    Document document = new Document(pdf);
    Paragraph p = new Paragraph()
        .add("Be prepared to read a story about a London lawyer "
        + "named Gabriel John Utterson who investigates strange "
        + "occurrences between his old friend, Dr. Henry Jekyll, "
        + "and the evil Edward Hyde.");
    document.add(p);
    document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
    ... // Define column areas
    document.setRenderer(new ColumnDocumentRenderer(document, columns));
    document.add(new AreaBreak(AreaBreakType.LAST_PAGE));
    ... // Add novel in two columns
    document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
    document.setRenderer(new DocumentRenderer(document));
    document.add(new AreaBreak(AreaBreakType.LAST_PAGE));
    p = new Paragraph()
        .add("This was the story about the London lawyer "
        + "named Gabriel John Utterson who investigates strange "
        + "occurrences between his old friend, Dr. Henry Jekyll, "
        + "and the evil Edward Hyde. THE END!");
```

[25]http://developers.itextpdf.com/node/3186/draft#1900-c02e10_jekyllhydev6.java

```
23      document.add(p);
24      document.close();
25  }
```

We add a long `Paragraph` to the first page (line 4-9). As we didn't define any renderer, the default `DocumentRenderer` is used. We introduce a page break (line 10) and change the renderer to a `ColumnDocumentRenderer` with two columns. Right after we set this new renderer, we introduce an `AreaBreak` that jumps to the last page. Why is this necessary?

Whenever you create a new `DocumentRenderer`, iText starts returns to the top of the document –that is: from the first page. This allows you to use different renderers on the same document next to each other on the same page. If that is needed, we'll have to instruct iText not to flush the content to the `OutputStream`; otherwise we won't have access to previous pages. In this case, we don't need to change anything on previous pages. We just want to switch to another renderer on the next page. Introducing a page break that goes to the last page will avoid that new content overwrites old content.

If we would omit `document.add(new AreaBreak(AreaBreakType.LAST_PAGE));`, then the new content, organized in columns, would be added on the first page, overwriting the long paragraph.

We introduce another page break after we've finished adding the novel (line 15). We change the renderer back to the standard `DocumentRenderer` (line 17), but we add another `LAST_PAGE` area break (line 17) before we add another paragraph (line 18-23).

The `AreaBreak` examples explaining the difference between `NEXT_AREA`, `NEXT_PAGE`, and `LAST_PAGE` have led us somewhat astray. We overlooked an important question we need to ask when rendering PDF: when do we flush the content to the `OutputStream`?

# Flushing the Document renderer

If you look at the API documentation for `Canvas`, `Document`, `CanvasRenderer`, `DocumentRenderer` and `ColumnDocumentRenderer`, you'll notice that all of these objects have at least one constructor that accepts a Boolean parameter named `immediateFlush`. So far, we've never used one of these constructors. As a result, iText always used the default value: `true`. All the content we've added was always flushed immediately.

In the next three examples, we'll set the value to `false`. In these three examples, we'll postpone flushing the content for three very specific reasons: to change the layout after content was added, to change the content of objects after they were added, and to add content to previous pages.

In iText 5, content added to a `Document` was flushed to the `OutputStream` as soon as a page was full. Once content was added to a page, there was no way to change (the layout of) that content. With iText 7, there's a way to postpone the actual rendering of the content, allowing us to apply changes after the content was added to the `Canvas` or `Document`.

Let's return to the example in which we converted text to a PDF document with two columns, more specifically to the example in which we introduced page breaks before every new chapter. These page breaks result in different pages having only one column. As we can tell from figure 2.11, this column is on the right side of the page.

Now suppose that we want to move these solitary columns to the middle of the page as shown in figure 2.13.

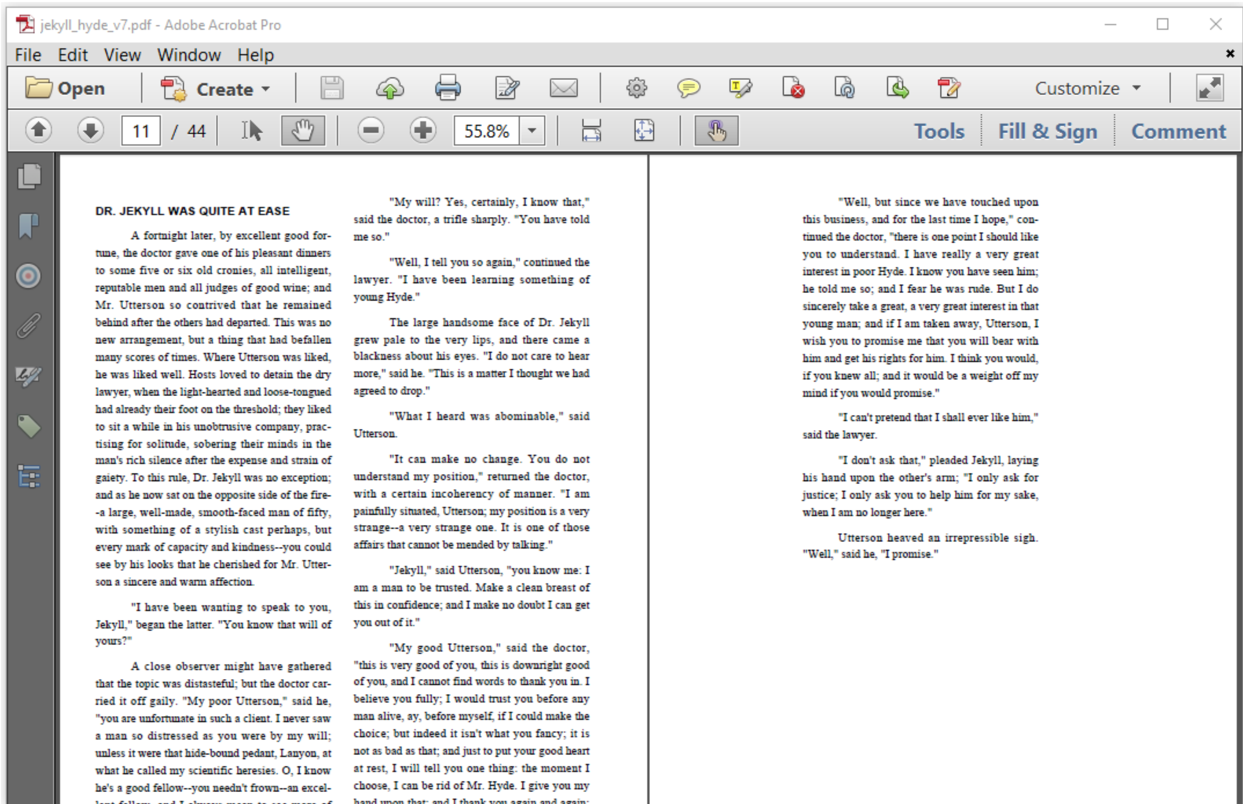



**Figure 2.13: Moving a column to the middle of the page**

We can't tell in advance when this situation will occur. We parse the text line by line, and we don't know what the next line will bring us when we add a `Paragraph` to the document. It could be another `Paragraph` or a `LineBreak`. This means that we shouldn't render the content right away. If we did, we couldn't move it to the middle if a chapter ends somewhere in the left column. We need to postpone flushing. We can do so in the renderer as demonstrated in the JekyllHyderV7[26] example.

In this example, we took the code of the `ColumnDocumentRenderer` class and we adapted it to our specific needs.

[26] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1901-c02e11_jekyllhydev7.java

```
class MyColumnRenderer extends DocumentRenderer {
    protected int nextAreaNumber;
    protected final Rectangle[] columns;
    protected int currentAreaNumber;
    protected Set<Integer> moveColumn = new HashSet<>();

    public MyColumnRenderer(Document document, Rectangle[] columns) {
        super(document, false);
        this.columns = columns;
    }

    @Override
    protected LayoutArea updateCurrentArea(LayoutResult overflowResult) {
        if (overflowResult != null
            && overflowResult.getAreaBreak() != null
            && overflowResult.getAreaBreak().getType()
                != AreaBreakType.NEXT_AREA) {
            nextAreaNumber = 0;
        }
        if (nextAreaNumber % columns.length == 0) {
            super.updateCurrentArea(overflowResult);
        }
        currentAreaNumber = nextAreaNumber + 1;
        return (currentArea = new LayoutArea(currentPageNumber,
            columns[nextAreaNumber++ % columns.length].clone()));
    }

    @Override
    protected PageSize addNewPage(PageSize customPageSize) {
        if (currentAreaNumber != nextAreaNumber
            && currentAreaNumber % columns.length != 0)
            moveColumn.add(currentPageNumber - 1);
        return super.addNewPage(customPageSize);
    }

    @Override
    protected void flushSingleRenderer(IRenderer resultRenderer) {
        int pageNum = resultRenderer.getOccupiedArea().getPageNumber();
        if (moveColumn.contains(pageNum)) {
            resultRenderer.move(columns[0].getWidth() / 2, 0);
        }
        super.flushSingleRenderer(resultRenderer);
```

```
43      }
44  }
```

Let's take a closer look at this custom `DocumentRenderer`:

- Line 2-5: we reuse two member-variables from the `ColumnDocumentRenderer`: the `nextAreaNumber` integer keeps track of the column count; the `columns` array stores the position and dimension of each column. We add an extra integer `currentAreaNumber` that remembers the current column count and a `moveColumn` collection in which we'll store the page numbers of the pages with a single column.
- Line 7-9: we construct a `MyColumnRenderer` instance. We call the constructor of the `DocumentRenderer` superclass and set the `immediateFlush` parameter to `false`: content will not be flushed immediately.
- Line 12-26: the `updateCurrentArea()` method is identical to the method with the same name in the `ColumnDocumentRenderer` class, except for one tiny difference: we set the value of `currentAreaNumber` to `nextAreaNumber + 1`. This method is called each time a new column is started. Note that the `currentAreaNumber` is set to `0` each time a page break is introduced.
- Line 28-34: we override the `newPage()` method. This method is triggered every time a new page is started. Whether or not the content was rendered to the previous page, depends on the value of `immediateFlush`. We use this method to check if the previous page consisted of only one column. This is the case if `currentAreaNumber` and `nextAreaNumber` aren't equal and if the value of `currentAreaNumber` is odd (this assumes that `columns` is an array with only two elements). If there's only one column in the previous page, we add the page number of that page (`currentPageNumber - 1`) to the `moveColumns` collection.
- Line 36-43: we override the `flushSingleRenderer()` method. This is the method that renders the content. If `immediateFlush` is `true`, this method is called automatically. If `immediateFlush` is `false`, we have to trigger the rendering process ourselves. We override this method because we want to move the coordinates of the `IRenderer` to the right with half a column width for every page we registered as a single-column page in the `newPage()` method.

Now let's take a look at how we can use this custom column renderer.

```
1  Rectangle[] columns = {
2      new Rectangle(offSet, offSet, columnWidth, columnHeight),
3      new Rectangle(
4          offSet + columnWidth + gutter, offSet, columnWidth, columnHeight)};
5  DocumentRenderer renderer = new MyColumnRenderer(document, columns);
6  document.setRenderer(renderer);
```

We define an array with two `Rectangle` objects. We use this array to create an instance of our custom `MyColumnRenderer` object. We use this instance as the renderer for our `Document`. The rest

of our code is identical to what we had before: we set the default values for the `Document`; then we parse the text file and we add content while doing so.

If we would close the `document` object after adding all the content, we'd end up with a document that consists of nothing but empty pages. In our renderer, we jump from area to area, and we create new page after new page, but we aren't rendering anything because the `flushSingleRenderer()` method is never called. We have to trigger this method ourselves, and we can do so like this:

```
1 renderer.flush();
2 document.close();
```

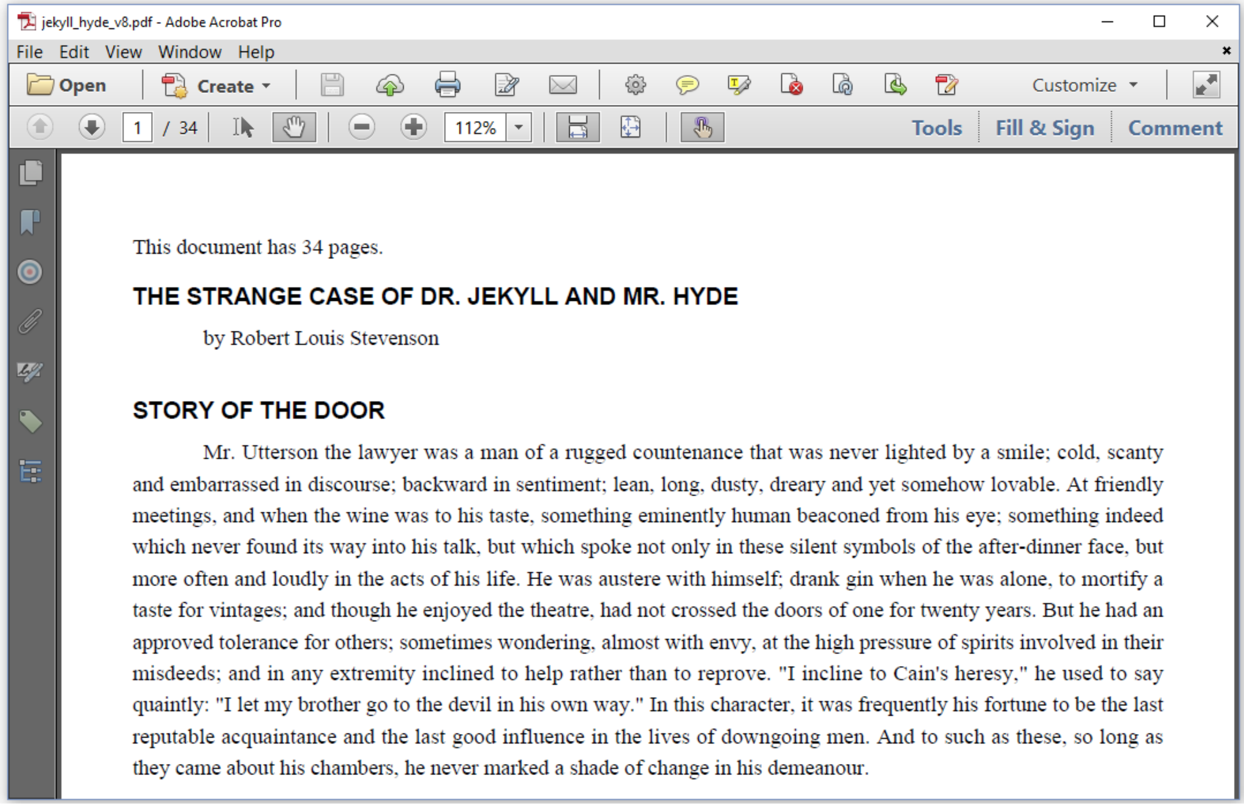
When we `flush()` the renderer, all the content we've been adding without flushing will be rendered. The `flushSingleRenderer()` method will be called as many times as there are objects added to the `Document`. Every time it's called on a page marked as a single-column page, the content will be moved to the right so that the column appears in the middle of the page.

This is one of the more complex examples in this book. Writing your own `RootRenderer` implementation isn't easy, but this functionality gives you a lot of power to create PDF documents the way *you* want to, as opposed to the way iText wants to.

Let's continue with a couple of examples in which we use the `immediateFlush` parameter when creating a `Document` instance.

# Changing content that was previously added

Take a close look at figure 2.14. At first sight, it isn't all that different from examples we've seen before, but there's something special about the first line of text.


This document has 34 pages.

THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

**Figure 2.14: Start by showing the total number of pages**

The first line of text says "This document has 34 pages." From previous examples, we know that we're building a document as we go, reading a text file line by line. When we parse the first lines of text, there is no way we can predict how many pages will be needed for the full document. How did we guess that we'd end up with 34 pages?

Truth be told, we didn't have to guess; we used a little trick. The JekyllHydeV8[27] example reveals the magic we used. We created a `Document` instance with the `immediateFlush` parameter set to false.

```
1 Document document = new Document(pdf, PageSize.A4, false);
```

The first object we add to this `document` is some text saying "This document has {totalpages} pages."

```
1 Text totalPages = new Text("This document has {totalpages} pages.");
2 IRenderer renderer = new TextRenderer(totalPages);
3 totalPages.setNextRenderer(renderer);
4 document.add(new Paragraph(totalPages));
```

As you can see, we used a placeholder `{totalpages}` for the total number of pages. We created a `TextRenderer` instance and added this `renderer` as the next renderer for the `Text` object. We wrap

[27] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1902-c02e12_jekyllhydev8.java

the Text object in a Paragraph and add this paragraph to the document. Then we add all story of Dr. Jekyll and Mr. Hyde. Because of the fact that immediateFlush is false, no text will be rendered until at the very last moment. This very last moment could be when we close the document, in which case the first line would still read "This document has {totalpages} pages."

Obviously, that's not what we want. We want to change {totalpage} into the actual number of pages before the text is rendered. This can be achieved using the TextRenderer object.

```
1 String total = renderer.toString().replace("{totalpages}",
2     String.valueOf(pdf.getNumberOfPages()));
3 ((TextRenderer)renderer).setText(total);
4 ((Text)renderer.getModelElement()).setNextRenderer(renderer);
5 document.relayout();
6 document.close();
```

In line 1-2, we change the String "This document has {totalpages} pages." to "This document has 34 pages." As you can see, we can retrieve the original content of the Text object from the renderer and we replace the placeholder with pdf.getNumberOfPages(). In line 3-4, we change the text of the TextRenderer and we add this altered text renderer to the Text object.

If we would close the document after line 4, the PDF would still show "This document has {totalpages} pages." For the change to take effect, we need to re-layout the document. This is done using the relayout() method in line 5. Only after the layout has been recreated, we can close the document, as is done in line 6.

In iText 5, we could have achieved more or less the same result by adding a placeholder with fixed dimensions. Once the complete document was rendered, we could then fill out the total number of pages on the placeholder. We will use the same approach with iText 7 in chapter 7, but iText 7 now also provides an alternative solution by allowing us to change the content of a Text object and then recreate the layout.

Changing the content of a Text object is still somewhat complex. There are many cases where we don't need to recreate the layout. In those cases, the complexity can be reduced substantially as demonstrated in the next example.

## Adding a Page X of Y footer

In figure 2.15, we see that each page has a footer that indicates the current number and the total number of pages.

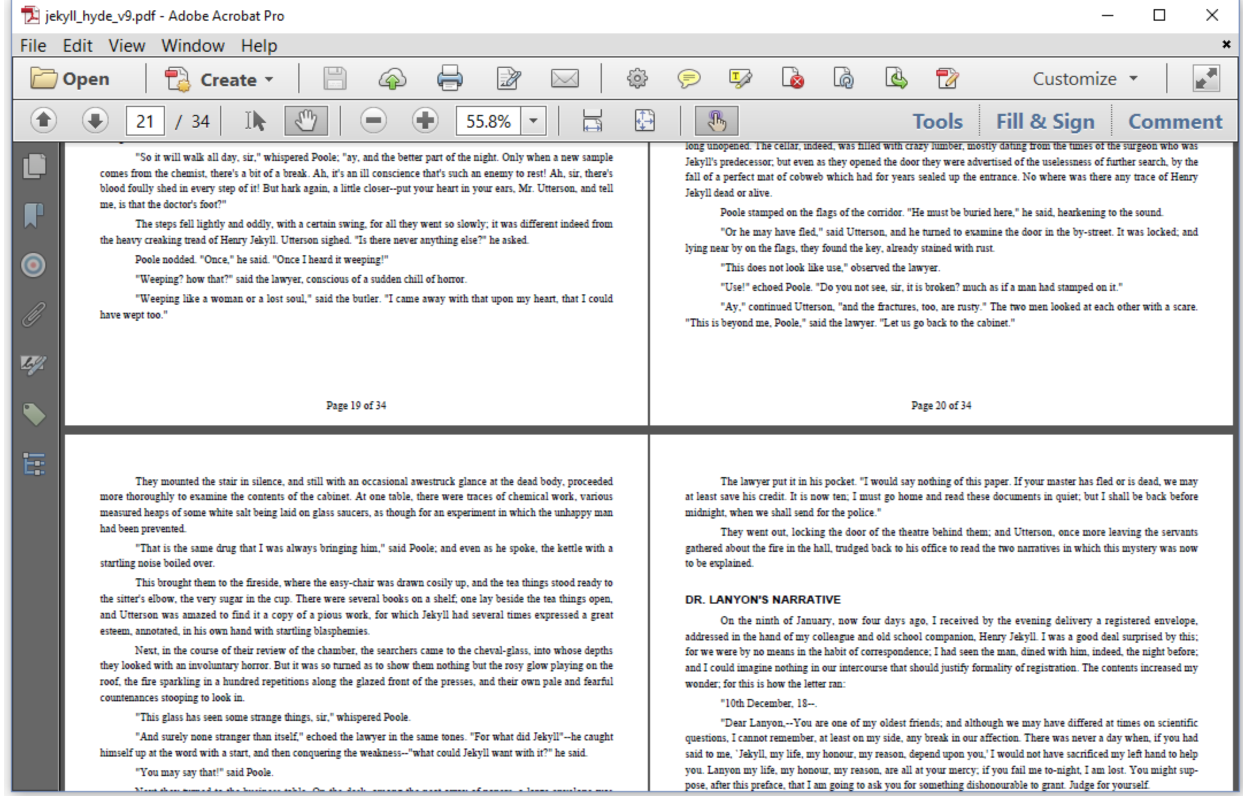



**Figure 2.15: Page X of Y footer**

To achieve this, we used a much easier approach than what we did in the previous example. Let's take a look at the JekyllHydeV9[28] example.

Once more, we tell the `Document` that it shouldn't flush its content immediately.

```
1 Document document = new Document(pdf, PageSize.A4, false);
```

After adding the complete text of the short story by Robert Louis Stevenson, we loop over every page in the document and we add a `Paragraph` to each page.

[28] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1903-c02e13_jekyllhydev9.java

```
int n = pdf.getNumberOfPages();
Paragraph footer;
for (int page = 1; page <= n; page++) {
    footer = new Paragraph(String.format("Page %s of %s", page, n));
    document.showTextAligned(footer, 297.5f, 20, page,
        TextAlignment.CENTER, VerticalAlignment.MIDDLE, 0);
}
document.close();
```

The `showTextAligned()` method can be used to add text at an absolute position on any page, using a specific horizontal and vertical alignment with respect to the chosen coordinate, and using a specific angle.

In this case, we loop over all the pages (from 1 to 34) and we add a line of text centered vertically and horizontally at position `x = 297.5f` and `y = 20` on every page. We didn't need to change the layout of any of the content that was already added, hence we don't need to use the `relayout()` method. All of the content is rendered at the moment we `close()` the document.

This example only works if you set `immediateFlush` to `false`. If you forget setting this parameter, you'll encounter the following exception:

> Exception in thread "main" java.lang.NullPointerException at com.itextpdf.kernel.pdf.PdfDictionary.get(PdfDictionary.java)

This exception occurs because you are trying to change the contents of a page dictionary that has already been flushed to the `OutputStream`. iText still has a reference to that page dictionary, but the dictionary as such is no longer there, hence the `NullPointerException`.

### Why didn't we get a NullPointerException in our low-level CanvasReturn example?

In the CanvasReturn[29] example, we created `PdfPage` objects. As we are using low-level functionality, it is our responsibility to manage the resources. We can use the `flush()` method on a `PdfPage` object of a finished page to flush its content to the `OutputStream`. Once this is done, we can no longer add anything to that page. We'll get a `NullPointerException` if we try to get (one of) its content stream(s).

Let's take a look at some more `showTextAligned()` examples.

[29] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1894-c02e04_canvasreturn.java

# Adding text with showTextAligned

Different `showTextAligned()` methods are available in the `RootElement` class. These methods can be used in the `Canvas` and the `Document` object to put a single line of text at an absolute position. If this line of text doesn't fit the `Canvas` or if it doesn't fit the current page of the `Document`, it won't be split into different lines. It might even run off the page, outside the visible area of that page.

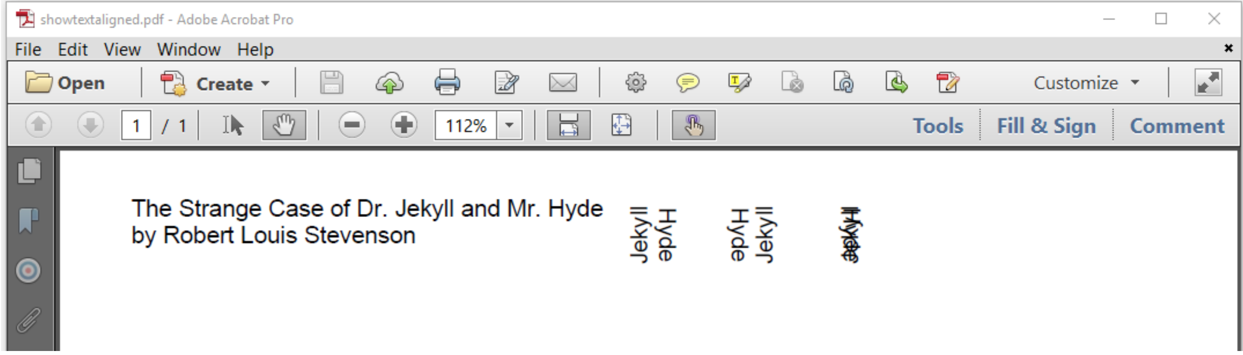



**Figure 2.16: Text added at absolute positions**

The PDF shown in figure 2.16 was created with the ShowTextAligned[30] example.

```
1  Paragraph title = new Paragraph("The Strange Case of Dr. Jekyll and Mr. Hyde");
2  document.showTextAligned(title, 36, 806, TextAlignment.LEFT);
3  Paragraph author = new Paragraph("by Robert Louis Stevenson");
4  document.showTextAligned(author, 36, 806,
5      TextAlignment.LEFT, VerticalAlignment.TOP);
6  document.showTextAligned("Jekyll", 300, 800,
7      TextAlignment.CENTER, 0.5f * (float)Math.PI);
8  document.showTextAligned("Hyde", 300, 800,
9      TextAlignment.CENTER, -0.5f * (float)Math.PI);
10 document.showTextAligned("Jekyll", 350, 800,
11     TextAlignment.CENTER, VerticalAlignment.TOP, 0.5f * (float)Math.PI);
12 document.showTextAligned("Hyde", 350, 800,
13     TextAlignment.CENTER, VerticalAlignment.TOP, -0.5f * (float)Math.PI);
14 document.showTextAligned("Jekyll", 400, 800,
15     TextAlignment.CENTER, VerticalAlignment.MIDDLE, 0.5f * (float)Math.PI);
16 document.showTextAligned("Hyde", 400, 800,
17     TextAlignment.CENTER, VerticalAlignment.MIDDLE, -0.5f * (float)Math.PI);
```

In line 1 and 3, we create two `Paragraph` objects. We add these objects to the current page using the `showTextAligned()` method.

[30]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1904-c02e14_showtextaligned.java

- In line 2, we add the `Paragraph` at position `x = 36; y = 806` and we align the content to the left of this coordinate. We didn't define a vertical alignment. The default `VerticalAlignment.BOTTOM` will be used, which means that the coordinate will be considered as the bottom coordinate of the content.
- In line 4-5, we add the content at the exact same coordinate, but we define a different value for the vertical alignment: `VerticalAlignment.TOP`. Now the coordinate is considered as the top coordinate of the content.

In lines 6 to 17, we add text as a `String` instead of as a `Paragraph`. We also introduce rotation values of 90 degrees (`0.5f * (float)Math.PI`) and -90 degrees

- In lines 6-9, we add two names at the same coordinate, but with a different rotation angle. We do the same in lines 10-13. Notice the difference between the apparent order in which the names "Jekyll" and "Hyde" appear depending on the value of the `VerticalAlignment` (as we introduce a rotation of 90 degrees, vertical becomes horizontal, and vice-versa).
- In lines 14-17, we add both names at the same coordinate with a different angle, but with `VerticalAlignment.MIDDLE`. The names are written on top of each other and have become almost illegible.

This example demonstrates the different variations of `showTextAligned()` methods. There's also a `showTextAlignedKerned()` method, but we need to learn more about using iText 7 add-ons before we can use that method in an example.

## Using iText 7 add-ons

The core libraries of iText 7 are available as open source software under the AGPL license. This means that you can use iText in your applications without having to pay a license fee *as long as* you distribute your own software based on iText under the same license. To put it simple: you can use iText for free if you also make your own source code available for free. The moment you distribute your code under another license –for instance: you work for a customer who uses your code in a closed source environment–, you or your customer have to purchase a commercial license.

There's more to it, but it would lead us too far to discuss the AGPL; this is a technical tutorial, not a book of law.

Many developers aren't aware of the implications of using AGPL software. This can be very annoying for many different reasons. These are some examples of such annoyances:

- Companies at the verge of getting funding or being acquired, fail the due diligence process because they don't have a commercial license for their use of iText.
- iText Group successfully sued a company for blatant abuse of our intellectual property as an example proving that the AGPL can be enforced. The case was won in about one and a half month. That was fast, but at iText Group, we all agree that there are better ways to spend our time than by going to court because some company wrongly assumes that open source software is software that is free of obligations and free of charge.
- Some companies ignore the implications of the AGPL license deliberately. This leads to unfair competition between customers who buy a commercial license, allowing us to invest in further development, and users who benefit from the further development, refusing to contribute in any way.

To create more awareness and to avoid misunderstandings, we decided to make part of iText closed source. We've defined a series of valuable add-ons that won't be available as open source software. We used to work with a *dual licensing* model and we'll continue to do so, but now we're also using the *open core* model. If developers want to use the functionality that is only available in a closed source add-on, a commercial license will have to be purchased.

The pdfCalligraph[31] module (aka the typography jar) is one example of such a closed source add-on. We've spent a lot of time and effort into improving the typography. With pdfCalligraph, iText finally supports Indic writing systems such as Devanagari and Tamil. iText now also supports special features such as the visualization of vowels in Arabic. All of this functionality is available in a separate typography jar.

You can use the pdfCalligraph add-on by introducing the following dependency:

```
1 <dependency>
2     <groupId>com.itextpdf</groupId>
3     <artifactId>typography</artifactId>
4     <version>1.0.0</version>
5     <scope>compile</scope>
6 </dependency>
```

When importing a closed source add-on, you need a license-key in order to use that add-on. You need the itext-licensekey jar to import that key into your code. This is the dependency for the license-key jar:

[31] http://itextpdf.com/itext7/pdfcalligraph

```
<dependency>
    <groupId>com.itextpdf</groupId>
    <artifactId>itext-licensekey</artifactId>
    <version>2.0.0</version>
    <scope>compile</scope>
</dependency>
```

Loading the license key into your code is done like this:

```
LicenseKey.loadLicenseFile(new FileInputStream(KEY));
```

In my case, the `KEY` value is a constant with the path to my personal license key for using the typography jar.

If you introduce an add-on, but you forget adding the line using the `loadLicenseFile()` method, you'll run into the following exception:

> Exception in thread "main" java.lang.RuntimeException: java.lang.reflect.InvocationTargetException ... Caused by: com.itextpdf.licensekey.LicenseKeyException: License file not loaded.

If you try to load the license key, but it's missing, the following exception will be thrown:

> Exception in thread "main" java.io.FileNotFoundException:itextkey.xml (The system cannot find the path specified)

If the key was found at this location, but it was corrupted, you'll get this `LicenseKeyException`:

> Exception in thread "main" com.itextpdf.licensekey.LicenseKeyException: Signature was corrupted.

If you are using a license key that is expired, you'll get yet another message:

> Exception in thread "main" com.itextpdf.licensekey.LicenseKeyException: License expired.

These are the most common exceptions that can occur. Usually, a verbose message will tell you what went wrong. In the next example, we're going to use the typography jar to introduce *kerning*.

# Improving the typography

Figure 2.17 shows the difference between text *without* kerning and text *with* kerning.

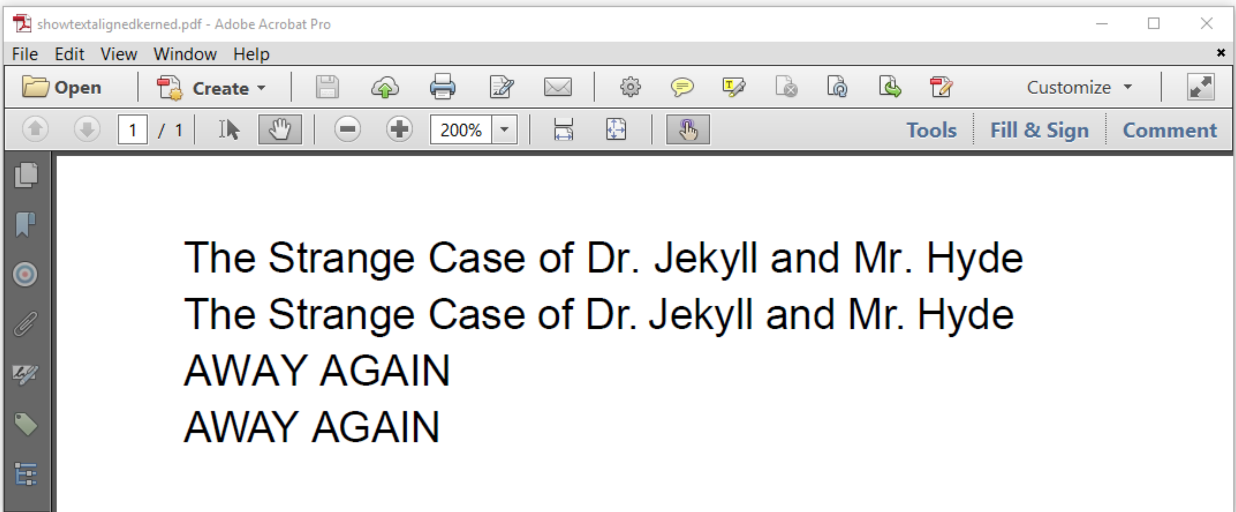



**Figure 2.17: Kerned text**

The kerning mechanism isn't that obvious in the title of Stevenson's short story. The devil is in the details: the `.` after `Dr` and `Mr` has been slightly moved in the kerned line. When kerning is active, the font program is consulted for kerning information. In this case, the font program knows that when a combination of `r` and `.` is encountered, the `.` should moved closer to the `r`.

The mechanism is easier to spot in the word `"AWAY"`. In the kerned version, the `A` characters move closer to the `W` on both sides. The distance between the `A` and the `Y` has also been reduced. The ShowTextAlignedKerned[32] example demonstrates how we used the `showTextAlignedKerned()` method to achieve this.

[32] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1905-c02e15_showtextalignedkerned.java

```
1 document.showTextAligned(
2     "The Strange Case of Dr. Jekyll and Mr. Hyde", 36, 806, TextAlignment.LEFT);
3 document.showTextAlignedKerned(
4     "The Strange Case of Dr. Jekyll and Mr. Hyde", 36, 790,
5     TextAlignment.LEFT, VerticalAlignment.BOTTOM, 0);
6 document.showTextAligned("AWAY AGAIN", 36, 774, TextAlignment.LEFT);
7 document.showTextAlignedKerned("AWAY AGAIN", 36, 758,
8     TextAlignment.LEFT, VerticalAlignment.BOTTOM, 0);
```

The pdfCalligraph add-on is made an optional because improved typography requires more extensive processing power to examine character combinations and to look up if the font program contains kerning or ligature information for these combinations.

In iText 5, R2L script was supported, but only in the context of `ColumnText` and `PdfPCell`. You had to change the writing system explicitly. Ligatures were supported, but only in Arabic text. There was no support for Hindi or other Indic writing systems whatsoever. With iText 7, it's sufficient to add the typography jar to the CLASSPATH. As soon as iText 7 detects the pdfCalligraph add-on, The writing system will be automatically changed from *left to right* (L2R) to *right to left* (R2L) if Hebrew or Arabic is detected. When Devanagari or Tamil content is detected, ligatures will be made automatically.

All of this extra work may be overkill for straightforward English text, in which case you don't really need the pdfCalligraph add-on.

## I have tried using kerning / support for Arabic, Indic languages / ligatures, but it doesn't work. Why not?

The `showTextAlignedKerned()` method won't have any effect if you don't have the typography jar in your CLASSPATH. If the typography jar is missing, there will be no difference between the *normal* text and the *kerned* text. If you want to render Hindi or Arabic, the text will be rendered incorrectly without the typography jar. Ligatures won't be made unless you add the typography jar to your CLASSPATH.

Currently not all writing systems are supported. We started with Arabic, Devanagari and Tamil. Support for other writing systems will follow depending on what iText customers request.

We could continue with many more examples involving pdfCalligraph and typography, but we'll leave that for another tutorial. This chapter was about the `RootElement` objects `Canvas` and `Document`, and we've covered quite some ground.

# Summary

In this chapter, we discussed the `Canvas` and the `Document` object, both subclasses of the abstract `RootElement` class. We also made some examples with the corresponding `RootRenderer` classes, `CanvasRenderer` and `DocumentRenderer`. While doing so, we discovered that we can easily render content in columns using the `ColumnDocumentRenderer`. The column examples allowed us to learn more about the `AreaBreak` object, which is a subclass of the abstract `AbstractElement` class.

We rendered the text of the short story “The Strange Case of Dr. Jekyll and Mr. Hyde” many times tweaking different properties of the `Document` object. We learned that content is flushed to the `OutputStream` as soon as possible by default, but that we can ask iText to postpone the rendering of elements so that we can change their content or layout afterwards.

Finally, we discussed the mechanism of closed source add-ons for iText 7. These add-ons require a license key that needs to be purchased from iText Software. We’ve experimented with the pdfCalligraph add-on also known as the typography jar. In the next chapter, we’ll dig into the `ILeafElement` implementations. We’ve already used the `Text` object many times, but in the next chapter, we’ll also take a look at the `Link`, `Tab` and `Image` object.

# Chapter 3: Using ILeafElement implementations

The `ElementPropertyContainer` has three direct subclasses: `Style`, `RootElement`, and `AbstractElement`. We've briefly discussed `Style` at the end of chapter 1. We've discussed the `RootElement` subclasses `Canvas` and `Document` in the previous chapter. We'll deal with the `AbstractElement` class in the next three chapters:

- We'll start with the `ILeafElement` implementations `Tab`, `Link`, `Text`, and `Image` in this chapter.
- We'll continue with the `BlockElement` objects `Div`, `LineSeparator`, `List`, `ListItem`, and `Paragraph` in the next chapter.
- We'll conclude with the `BlockElement` objects `Table` and `Cell` in chapter 5.

Note that we've already discussed the `AreaBreak` object in chapter 2. We'll have covered all of the basic building blocks by the end of chapter 5.

In the previous chapter, we've used a txt file as a resource to create a PDF document. In this chapter, and in the chapters that follow, we'll also use a CSV file, jekyll_hyde.csv[33], as data source. See figure 3.1.

[33]http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/highlevel/src/main/resources/data/jekyll_hyde.csv

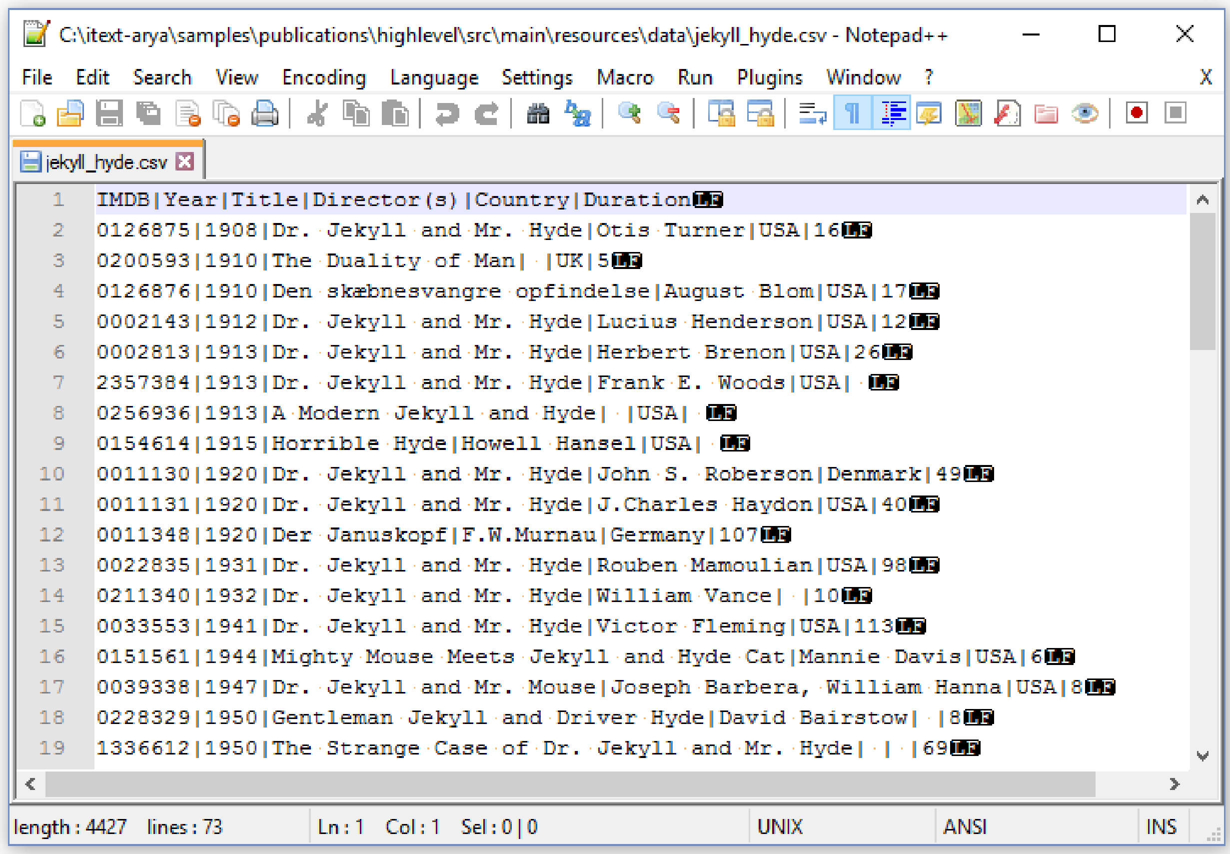



**Figure 3.1: A simple CSV file that will be used as data source**

As you can see, this CSV file could be interpreted as a database table containing records that consist of 6 fields:

1. *An IMDB number*– the ID of an entry in the Internet Movie Database (IMDB) that was based on the Jekyll and Hyde story by Robert Louis Stevenson.
2. *A year*– the year the corresponding movie, short film, cartoon, or video was produced.
3. *A title*– the title of the movie, short film, cartoon, or video.
4. *Director or directors*– the director or directors who made the movie, short film, cartoon, or video.
5. *A country*– the country where the movie, short film, cartoon, or video was produced.
6. *A run length*– the number of minutes of the movie, short film, cartoon, or video.

We will use the CsvTo2DList[34] utilities class to read this CSV file, that was stored using UTF-8 encoding, into a two-dimensional `List<List<String>>` list.

[34] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/util#1919-csvto2dlist.java

```
public static final List<List<String>> convert(String src, String separator)
    throws IOException {
    List<List<String>> resultSet = new ArrayList<List<String>>();
    BufferedReader br = new BufferedReader(
        new InputStreamReader(new FileInputStream(src), "UTF8"));
    String line;
    List record;
    while ((line = br.readLine()) != null) {
        StringTokenizer tokenizer = new StringTokenizer(line, separator);
        record = new ArrayList<String>();
        while (tokenizer.hasMoreTokens()) {
            record.add(tokenizer.nextToken());
        }
        resultSet.add(record);
    }
    return resultSet;
}
```

In this chapter, we'll render this two-dimensional list to a PDF using `Tab` elements.

## Working with Tab elements

Let's take a look at the JekyllHydeTabsV1[35] example:

```
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
for (List<String> record : resultSet) {
    Paragraph p = new Paragraph();
    p.add(record.get(0).trim()).add(new Tab())
        .add(record.get(1).trim()).add(new Tab())
        .add(record.get(2).trim()).add(new Tab())
        .add(record.get(3).trim()).add(new Tab())
        .add(record.get(4).trim()).add(new Tab())
        .add(record.get(5).trim());
    document.add(p);
}
```

In line 1, we use our CsvTo2DList[36] utilities class to create a `resultSet` of type `List<List<String>>`. In line 2, we loop over the *rows* of this result set, and we create a `Paragraph` containing all the fields in the `record` list. In-between, we add `Tab` objects.

[35]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1906-c03e01_jekyllhydetabsv1.java
[36]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/util#1919-csvto2dlist.java

Figure 3.2 shows the resulting PDF.

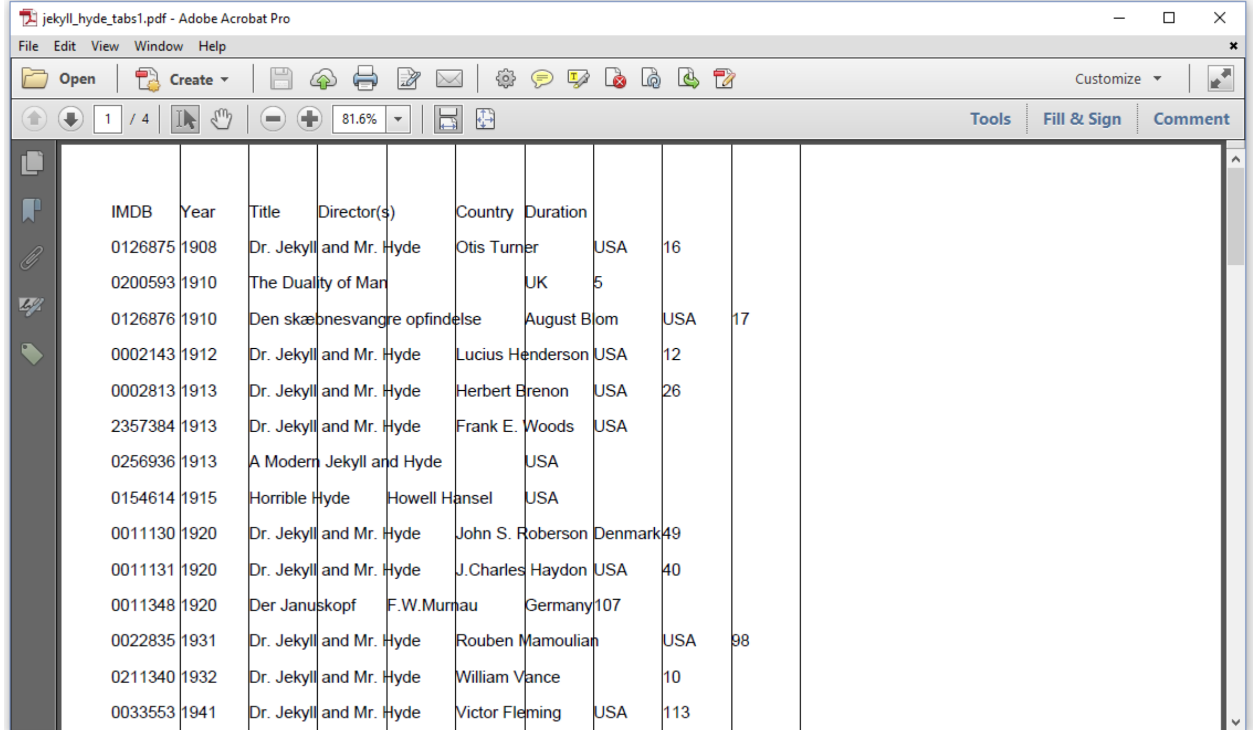
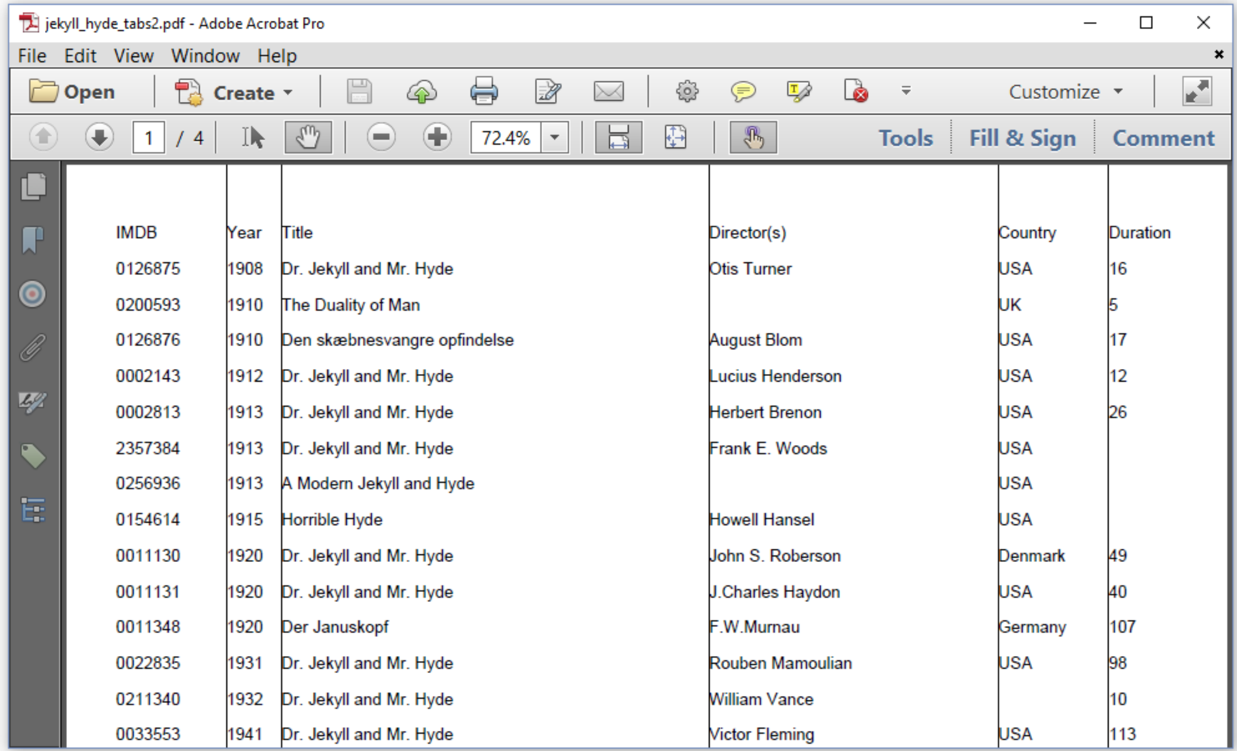



**Figure 3.2: default tab positions**

As you can see, we've added extra lines to show the default tab positions.

```
1  PdfCanvas pdfCanvas = new PdfCanvas(pdf.addNewPage());
2  for (int i = 1; i <= 10; i++) {
3      pdfCanvas.moveTo(document.getLeftMargin() + i * 50, 0);
4      pdfCanvas.lineTo(document.getLeftMargin() + i * 50, 595);
5  }
6  pdfCanvas.stroke();
```

By default, each tab position is a multiple of 50 user units (which, by default, equals 50 pt), starting at the left margin of the page. Those tab positions work quite well for the first three fields ("IMDB", "Year", and "Title"), but the "Director(s)" field starts at different positions, depending on the length of the "Title" field. Let's fix this and try to get the result shown in figure 3.3.



| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |

**Figure 3.3: defining tab positions**

In the JekyllHydeTabsV2[37] example, we define specific tab positions using the `TabStop` class.

```
float[] stops = new float[]{80, 120, 430, 640, 720};
List<TabStop> tabstops = new ArrayList();
PdfCanvas pdfCanvas = new PdfCanvas(pdf.addNewPage());
for (int i = 0; i < stops.length; i++) {
    tabstops.add(new TabStop(stops[i]));
    pdfCanvas.moveTo(document.getLeftMargin() + stops[i], 0);
    pdfCanvas.lineTo(document.getLeftMargin() + stops[i], 595);
}
pdfCanvas.stroke();
```

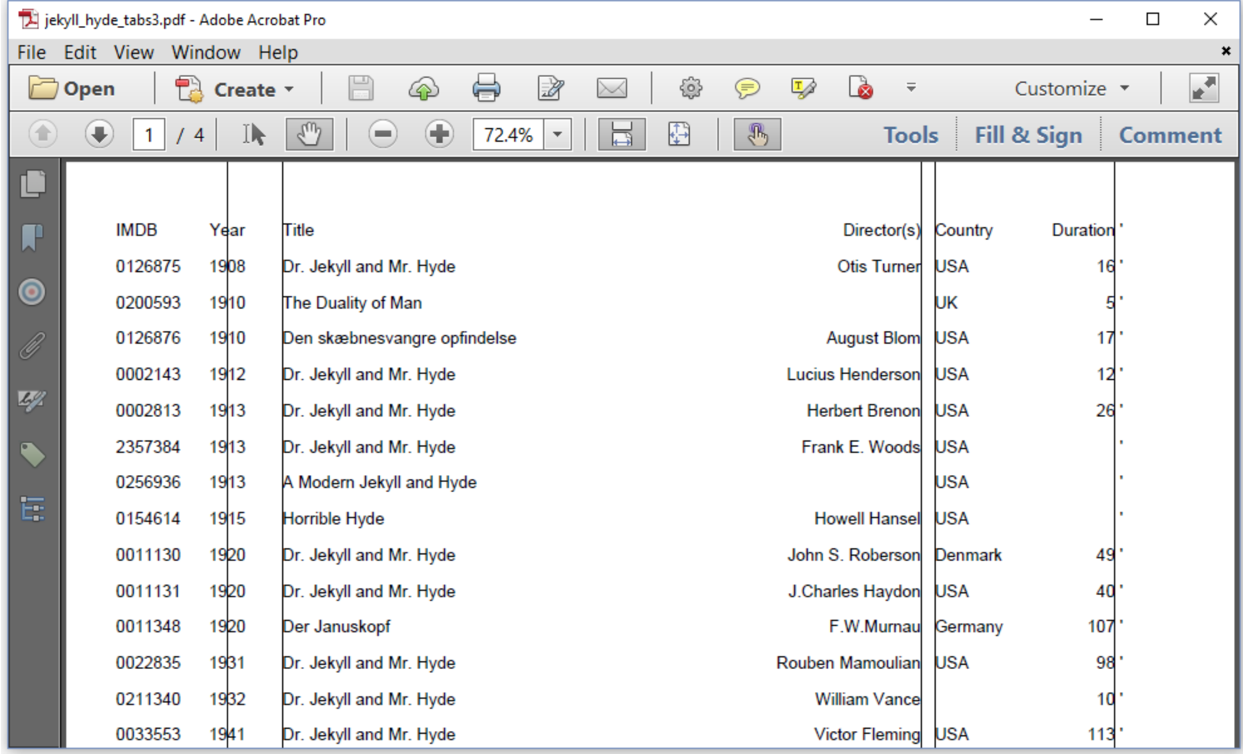
We've stored 5 tab stops in a `float` array in line 1, we create a `List` of `TabStop` objects in line 2, we loop over the different `float` values in line 4 and add the 5 tab stops to the `TabStop` list in line 5. While we are at it, we also draw lines that will show us the position of each tab stop, so that we have a visual reference to check if iText positioned our content correctly.

The next code snippet is almost an exact copy of what we had before.

[37] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1907-c03e02_jekyllhydetabsv2.java

```
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
for (List<String> record : resultSet) {
    Paragraph p = new Paragraph();
    p.addTabStops(tabstops);
    p.add(record.get(0).trim()).add(new Tab())
        .add(record.get(1).trim()).add(new Tab())
        .add(record.get(2).trim()).add(new Tab())
        .add(record.get(3).trim()).add(new Tab())
        .add(record.get(4).trim()).add(new Tab())
        .add(record.get(5).trim());
    document.add(p);
}
```

Line 4 is the only difference: we use the `addTabStops()` method to add the `List<TabStop>` object to the `Paragraph`. The different fields are now aligned in such a way that the content starts at the position defined by the tab stop; the tab stop is to the left of the content. We can change this alignment as shown in figure 3.4.




| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |

**Figure 3.4: different tab stop alignments**

The JekyllHydeTabsV3[38] example shows how this is done:

[38] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1908-c03e03_jekyllhydetabsv3.java

```
float[] stops = new float[]{80, 120, 580, 590, 720};
List<TabStop> tabstops = new ArrayList();
tabstops.add(new TabStop(stops[0], TabAlignment.CENTER));
tabstops.add(new TabStop(stops[1], TabAlignment.LEFT));
tabstops.add(new TabStop(stops[2], TabAlignment.RIGHT));
tabstops.add(new TabStop(stops[3], TabAlignment.LEFT));
TabStop anchor = new TabStop(stops[4], TabAlignment.ANCHOR);
anchor.setTabAnchor(' ');
tabstops.add(anchor);
```

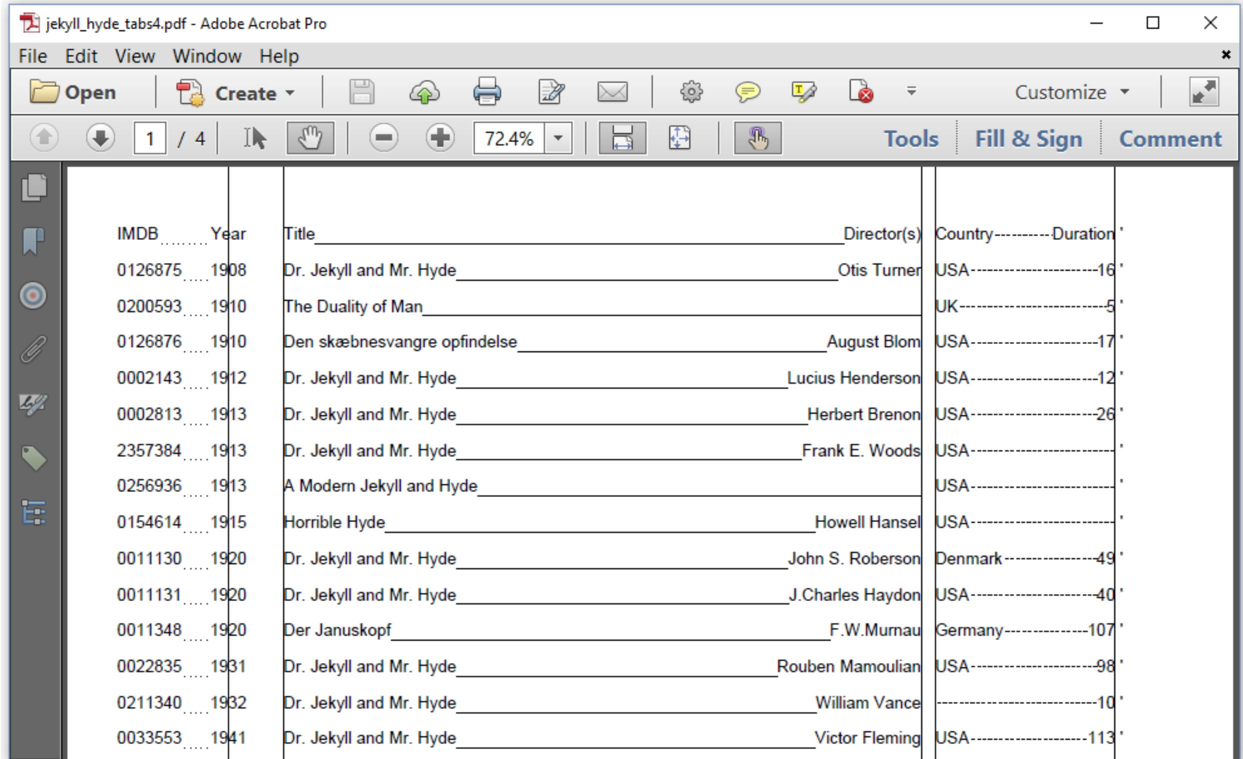
We have 5 tabstops:

- The first tab stop will center the Year at position 80; for this we use `TabAlignment.CENTER`.
- The second tab stop will make sure that the title starts at position 120; for this we use `TabAlignment.LEFT`.
- The third tab stop will make sure that the name(s) of the director(s) ends at position 580; for this we use `TabAlignment.RIGHT`.
- The fourth tab stop will make sure that the country starts at position 590.
- The fifth tab stop will align the content based on the position of the space character; for this we use `TabAlignment.ANCHOR` and we define a tab anchor using the `setTabAnchor()` method.

If you look at the CSV file, you see that we don't have any space characters in the "Run length" field, so let's add adapt our code and add `" \'"` to that field. See line 10 in the following snippet.

```
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
for (List<String> record : resultSet) {
    Paragraph p = new Paragraph();
    p.addTabStops(tabstops);
    p.add(record.get(0).trim()).add(new Tab())
        .add(record.get(1).trim()).add(new Tab())
        .add(record.get(2).trim()).add(new Tab())
        .add(record.get(3).trim()).add(new Tab())
        .add(record.get(4).trim()).add(new Tab())
        .add(record.get(5).trim() + " \'");
    document.add(p);
}
```

Figure 3.5 shows yet another variation on this example.



| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |

**Figure 3.5: tab stops with tab leaders**

In the JekyllHydeTabsV4[39] example, we add tab leaders.

```java
float[] stops = new float[]{80, 120, 580, 590, 720};
List<TabStop> tabstops = new ArrayList();
tabstops.add(new TabStop(stops[0], TabAlignment.CENTER, new DottedLine()));
tabstops.add(new TabStop(stops[1], TabAlignment.LEFT));
tabstops.add(new TabStop(stops[2], TabAlignment.RIGHT, new SolidLine(0.5f)));
tabstops.add(new TabStop(stops[3], TabAlignment.LEFT));
TabStop anchor = new TabStop(stops[4], TabAlignment.ANCHOR, new DashedLine());
anchor.setTabAnchor(' ');
tabstops.add(anchor);
```

A tab leader is defined using a class that implements the `ILineDrawer` interface. We add a dotted line between the IMDB id and the year, a solid line between the title and the director(s), and a dashed line between the country and the run length.

[39] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1909-c03e04_jekyllhydetabsv4.java

You could implement the `ILineDrawer` interface to draw any kind of line, but iText ships with three implementations that are ready to use: `SolidLine`, `DottedLine`, and `DashedLine`. Each of these classes allows you to change the line width and color. The `DottedLine` class also allows you to change the gap between the dots. In the next chapter, we'll also use these classes to draw line separators with the `LineSeparator` class.

At first sight, using the `Tab` object seems to be a great way to render content in a tabular form, but there are some serious limitations.

# Limitations of the Tab functionality

The previous screen shots looked nice because we chose our tab stops wisely. We rendered our data on an A4 page with landscape orientation, leaving sufficient space to render all the data. This won't always be possible. In figure 3.6, we try to add the same content on an A4 page with portrait orientation.

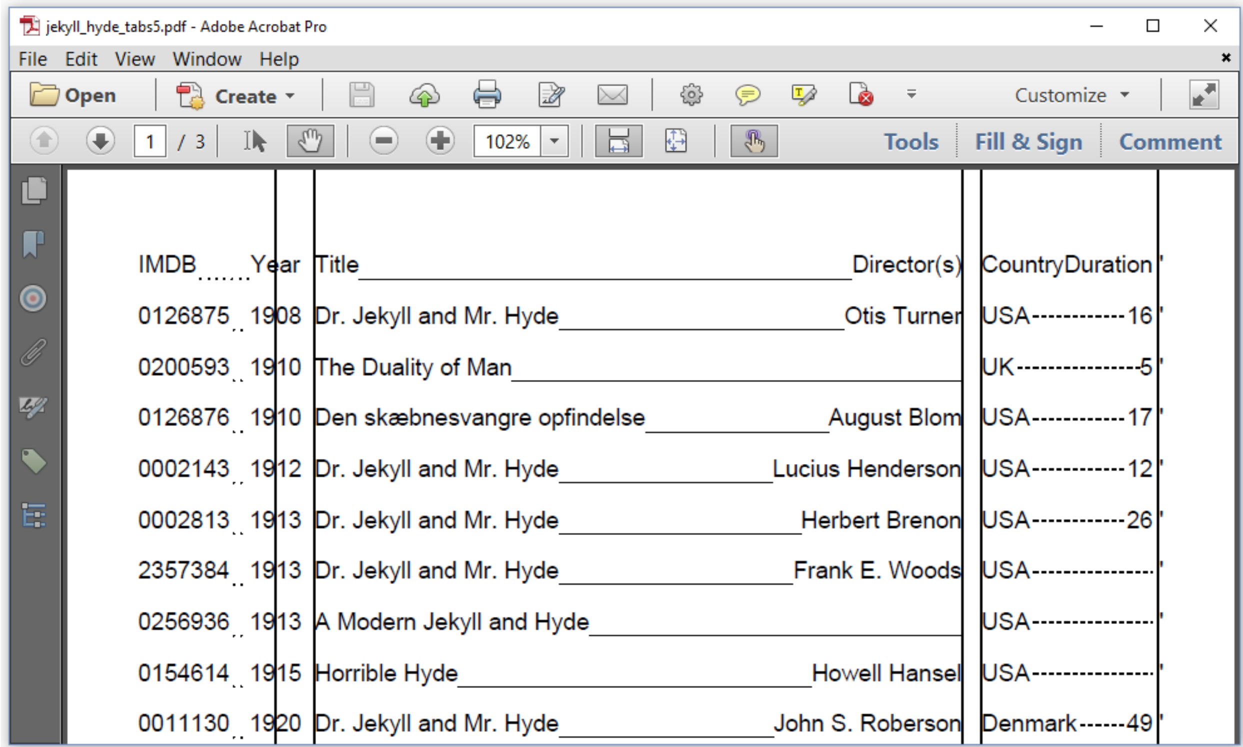



**Figure 3.6: using portrait orientation**

This PDF was made with the JekyllHydeV5[40] example. As you can see, this still looks quite nice, apart from the fact that "Country" and "Duration" stick together on the first line.

[40] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1910-c03e05_jekyllhydetabsv5.java

The `Tab` functionality contained some errors in iText 7.0.0. Due to rounding errors, some text wasn't aligned correctly in a seemingly random way. This problem was fixed in iText 7.0.1.

Another bug that was fixed in iText 7.0.1 is related to the `SolidLine` class. In iText 7.0.0, the line width of a `SolidLine` was ignored.

When we scroll down in the document, we see a more serious problem when there's no sufficient space to fit the title and the director next to each other. Director "Charles Lamont" pushes the country to the "Duration" column and the number of minutes gets shown on a second row.

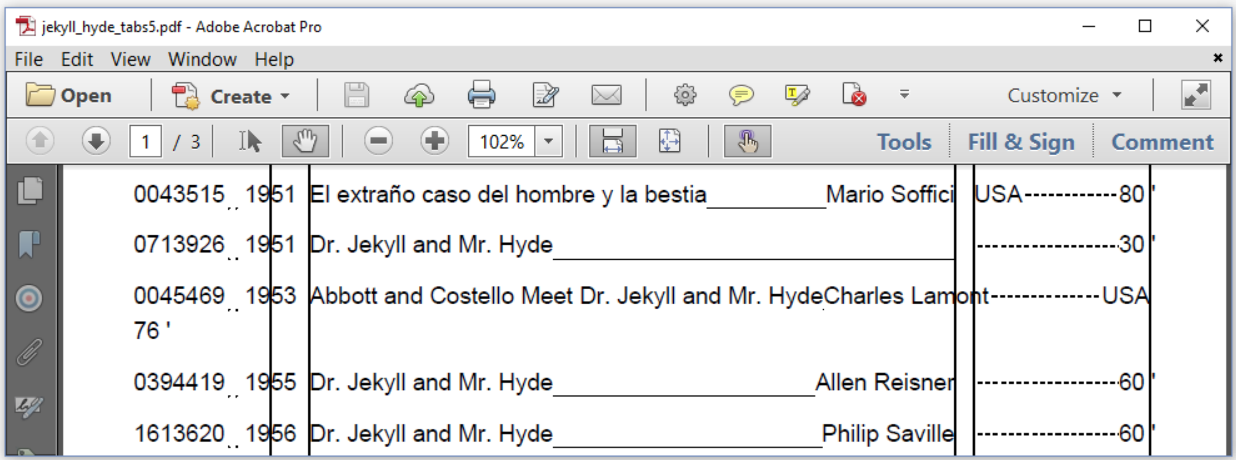


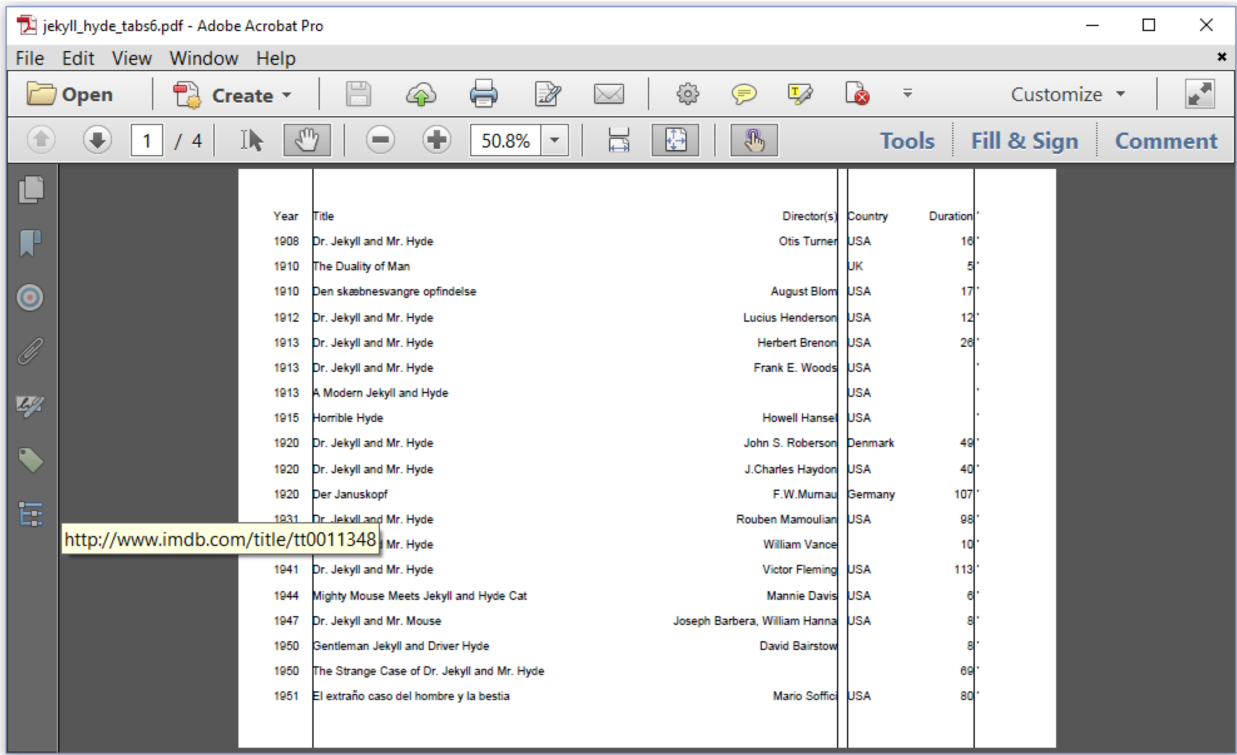

**Figure 3.7: trying to fit the data on a page with portrait orientation**

We can solve these problems by using the `Table` and `Cell` class to organize data in a tabular form. These objects will be discussed in chapter 5 of this tutorial. For now, we'll continue with some more `ILeafElement` implementations.

# Adding links

In the previous examples, we've added the ID of the movie, short film, cartoon, or video as actual content. This ID can help us find the movie on the Internet Movie Database (IMDB)[41]. In Figure 3.8, we don't show the ID, but when we click on the title of a movie, we can jump to the corresponding page on IMDB.

[41]http://imdb.com




| Year | Title | Director(s) | Country | Duration' |
|---|---|---|---|---|
| 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16' |
| 1910 | The Duality of Man | | UK | 5' |
| 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17' |
| 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12' |
| 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26' |
| 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | ' |
| 1913 | A Modern Jekyll and Hyde | | USA | ' |
| 1915 | Horrible Hyde | Howell Hansel | USA | ' |
| 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49' |
| 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40' |
| 1920 | Der Januskopf | F.W.Murnau | Germany | 107' |
| 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98' |
| [illegible] | [illegible] Mr. Hyde | William Vance | | 10' |
| 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113' |
| 1944 | Mighty Mouse Meets Jekyll and Hyde Cat | Mannie Davis | USA | 6' |
| 1947 | Dr. Jekyll and Mr. Mouse | Joseph Barbera, William Hanna | USA | 8' |
| 1950 | Gentleman Jekyll and Driver Hyde | David Bairstow | | 8' |
| 1950 | The Strange Case of Dr. Jekyll and Mr. Hyde | | | 69' |
| 1951 | El extraño caso del hombre y la bestia | Mario Soffici | USA | 80' |

**Figure 3.8: introducing links to IMDB**

We create these links in the JekyllHydeTabsV6[42] example.

```
1  List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
2  for (List<String> record : resultSet) {
3      Paragraph p = new Paragraph();
4      p.addTabStops(tabstops);
5      PdfAction uri = PdfAction.createURI(
6          String.format("http://www.imdb.com/title/tt%s", record.get(0)));
7      Link link = new Link(record.get(2).trim(), uri);
8      p.add(record.get(1).trim()).add(new Tab())
9          .add(link).add(new Tab())
10         .add(record.get(3).trim()).add(new Tab())
11         .add(record.get(4).trim()).add(new Tab())
12         .add(record.get(5).trim() + " \'");
13     document.add(p);
14 }
```

In line 5-6, we create a `PdfAction` object that links to an URL. This URL is composed of `http://www.imdb.com/title/tt/` and the IMDB ID. In line 7, we create a `Link` object using a `String`

[42] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1911-c03e06_jekyllhydetabsv6.java

containing the title of the movie, and the `PdfAction`. As a result, you will be able to jump to the corresponding IMDB page when clicking a title.

Interactivity in PDF is achieved by using *annotations*. Annotations aren't part of the real content. They are objects added on top of the content. In this case, a *link annotation* is used. There are many other types of annotations, but that's outside the scope of this tutorial. There are also many types of actions. For now, we've only used a URI action. We'll use some more in chapter 6.

The `Link` class extends the `Text` class. Appendix A[43] lists a series of methods that are available for the `Link` as well as for the `Text` class to change the font, to change the background color, to add borders, and so on.

## Extra methods available in the Text class

We've already worked with `Text` objects on many occasions in the previous chapters, but let's take a closer look at some `Text` functionality we haven't discussed yet.

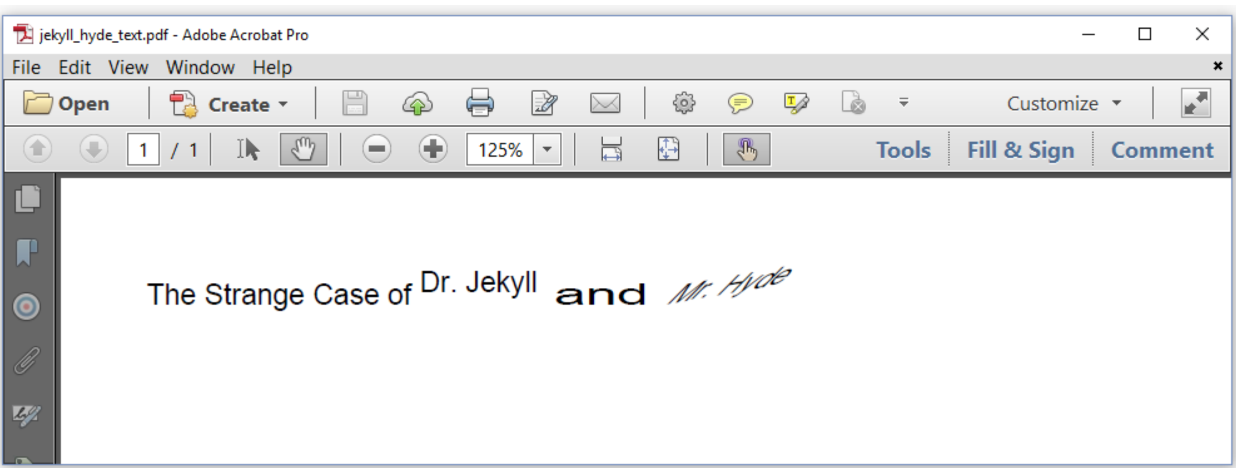



**Figure 3.9: extra text methods**

The first `Text` object shown in figure 3.9 is what text normally looks like. For the words "Dr. Jekyll", we defined a *text rise*. We scaled the word "and" horizontally. And we skewed the words "Mr. Hyde." The methods used to achieve this can be found in the TextExample[44] example.

[43] http://developers.itextpdf.com/content/itext-7-building-blocks/appendix/appendix-abstractelement-methods
[44] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1920-c03e07_textexample.java

```
Text t1 = new Text("The Strange Case of ");
Text t2 = new Text("Dr. Jekyll").setTextRise(5);
Text t3 = new Text(" and ").setHorizontalScaling(2);
Text t4 = new Text("Mr. Hyde").setSkew(10, 45);
document.add(new Paragraph(t1).add(t2).add(t3).add(t4));
```

We distinguish three new methods:

- The parameter passed to the `setTextRise()` method is the number of user units above the baseline of the text. You can also use a negative value if you want the text to appear below the base line.
- The parameter of the `setHorizontalScaling()` method is the horizontal scaling factor we want to use. In this case, the word `" and "` will be rendered double as wide as normal.
- The parameters of the `setSkew()` method define two angles in degrees. The first parameter is the angle between the text and its baseline. The second parameter is the angle that will be used to skew the characters. The `setSkew()` method is used to mimic an italic font (see chapter 1).

We'll continue using the `Text` object explicitly or implicitly in every example that involves text. The second half of this chapter will be dedicated entirely to the `Image` class.

# Introducing images

In 1996, Stephen Frears made a movie with Julia Roberts in the role of Mary Reilly, a maid in the household of Dr. Jekyll. Let's take an image of the poster of this movie and add it to a document as done in figure 3.10.

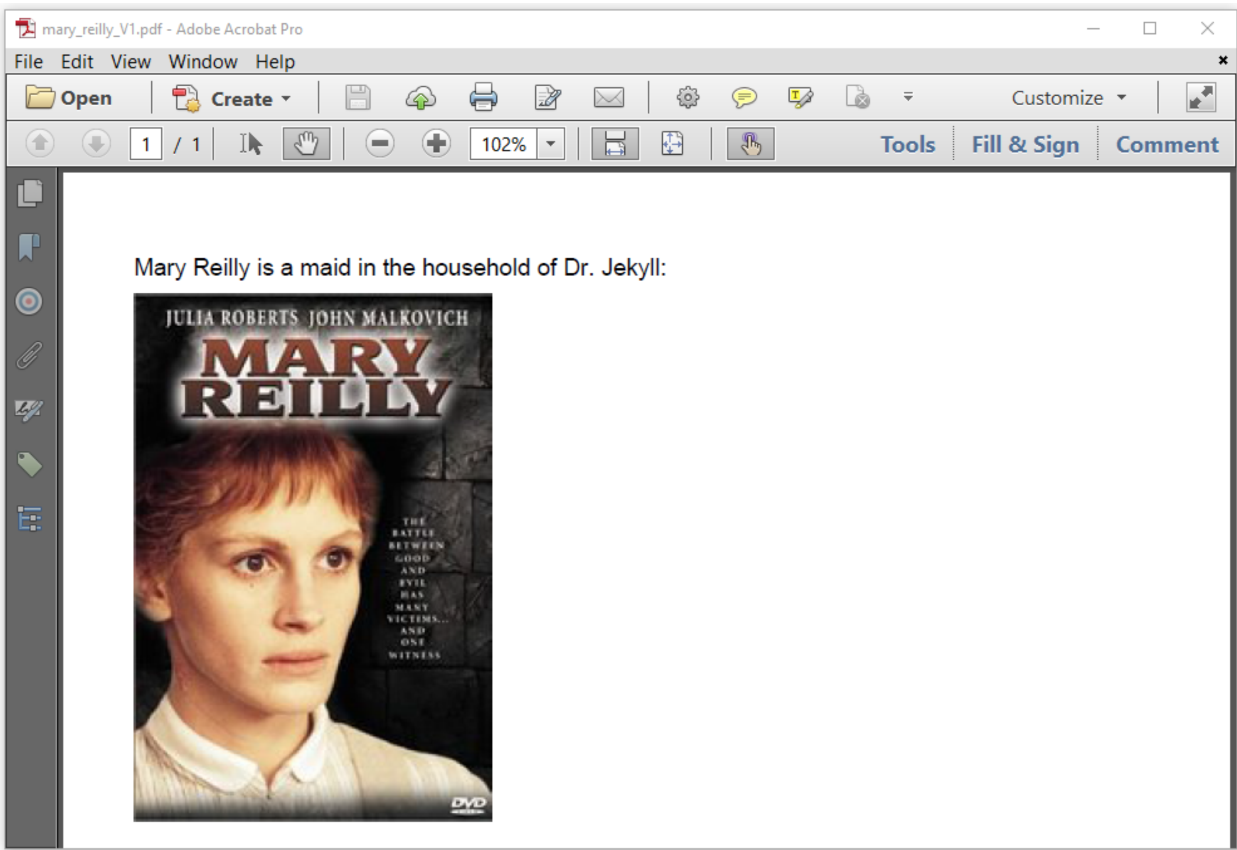



**Figure 3.10: an image added to a document**

The code to achieve this, is very simple. See the MaryReillyV1[45] example.

```
1  public static final String MARY = "src/main/resources/img/0117002.jpg";
2  public void createPdf(String dest) throws IOException {
3      PdfDocument pdf = new PdfDocument(
4          new PdfWriter(new FileOutputStream(dest)));
5      Document document = new Document(pdf);
6      Paragraph p = new Paragraph(
7          "Mary Reilly is a maid in the household of Dr. Jekyll: ");
8      document.add(p);
9      Image img = new Image(ImageDataFactory.create(MARY));
10     document.add(img);
11     document.close();
12 }
```

We have the path to our image in line 1. The `ImageDataFactory` uses this path in line 9 to get the image bytes and to convert them into an `ImageData` object that can be used to create an `Image` object.

[45] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1921-c03e08_maryreillyv1.java

In this case, we are passing a JPEG image, and we add that image straight to the `document` object in line 10.

JPEG images are stored inside a PDF as-is. It isn't necessary for iText to convert the image bytes into another image format. PNG for instance, isn't supported in PDF, hence iText will have to convert each PNG image we pass into a compressed bitmap.

Images are stored outside the content stream of the page in an object named an *image XObject*. XObject stands for eXternal Object. The bytes of the image are stored in a separate object *outside* the content stream. Now suppose that we would add the same image twice as is done in figure 3.11.

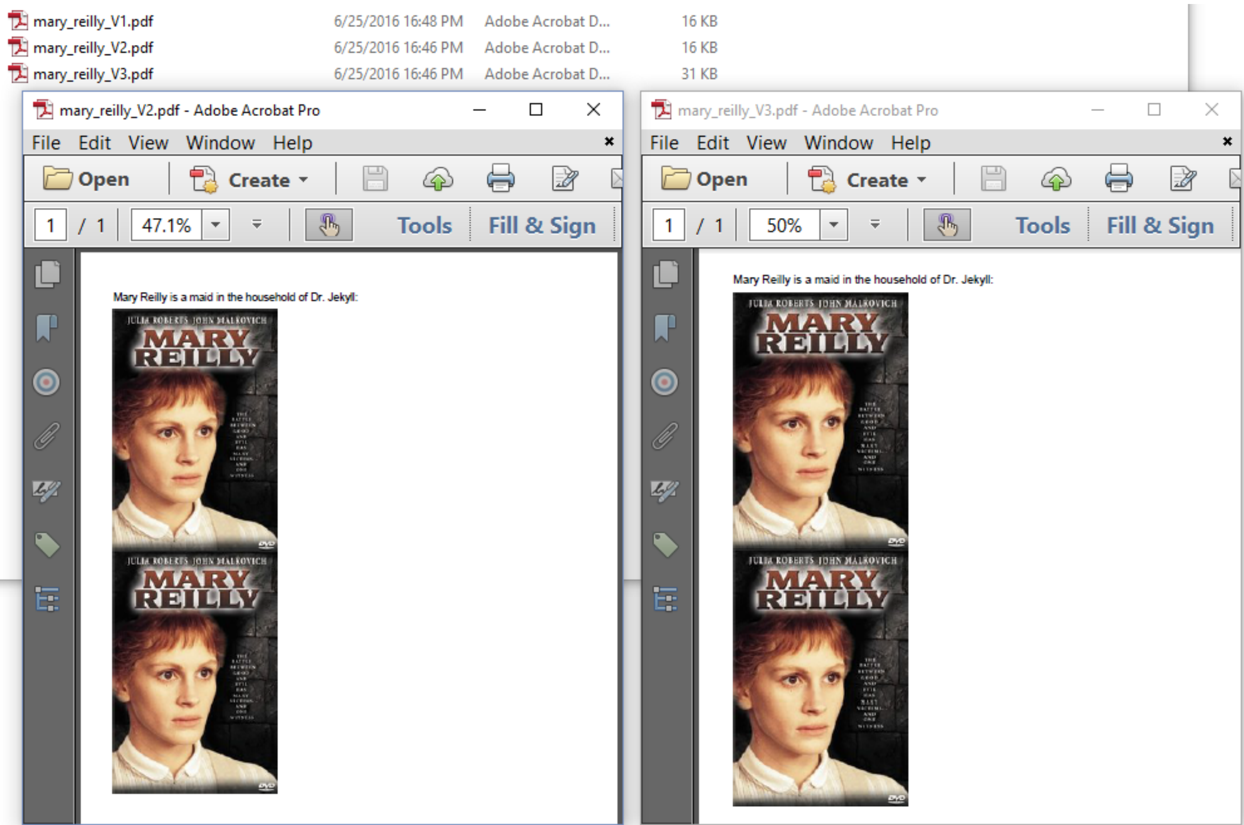



**Figure 3.11: adding the same figure twice**

When we compare the files mary_reilly_V2.pdf[46] and mary_reilly_V3.pdf[47], they look exactly the same to the naked eye. When we look at the file size of the files, we notice something strange:

- The file marked as V2 has the same file size as the file marked as V1. In other words, the file with two images has more or less the same file size as the file with a single image. This is

[46]http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/highlevel/cmpfiles/chapter03/cmp_mary_reilly_V2.pdf
[47]http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/highlevel/cmpfiles/chapter03/cmp_mary_reilly_V3.pdf

consistent with what we said before: the file is stored inside the document only once as an external object. We refer to this XObject twice.

- The file marked as V3 looks identical to the file marked as V2, but its file size is almost double the size of the file marked as V2. It's as if the image bytes of our JPEG are added twice to the PDF document.

The code we used to create the file marked as V2 can be found in the MaryReillyV2[48] example:

```
1 Image img = new Image(ImageDataFactory.create(MARY));
2 document.add(img);
3 document.add(img);
```

We create one `img` object; we add this image twice to the same `document`. As a result, the image is shown twice, but the image bytes are stored in a single image XObject.

Now let's take a look at the MaryReillyV3[49] example:

```
1 Image img1 = new Image(ImageDataFactory.create(MARY));
2 document.add(img1);
3 Image img2 = new Image(ImageDataFactory.create(MARY));
4 document.add(img2);
```

In this snippet, we create two `Image` instances for the same image, and we add both of these instances to the same `document`. Once more the image is shown twice, but now it's also stored twice (redundantly) inside the `document`.

There's a direct relationship between an `Image` object in iText and an image XObject inside the PDF. Every new `Image` object that is created and added to a document, results in a separate image XObject inside the PDF. If you create two or more `Image` objects of the same image, you'll end up with a bloated PDF file with too many redundant image XObjects. This is clearly something you want to avoid.

In these first examples, we added `Image` objects without defining a location. The first image was added right under our first paragraph. The second image was added right under the first one. We can also choose to add the `Image` at specific coordinates.

[48] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1925-c03e09_maryreillyv2.java
[49] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1926-c03e10_maryreillyv3.java

# Changing the position and width of an image

The two PDFs in figure 3.12 look identical, yet there were created in slightly different ways.

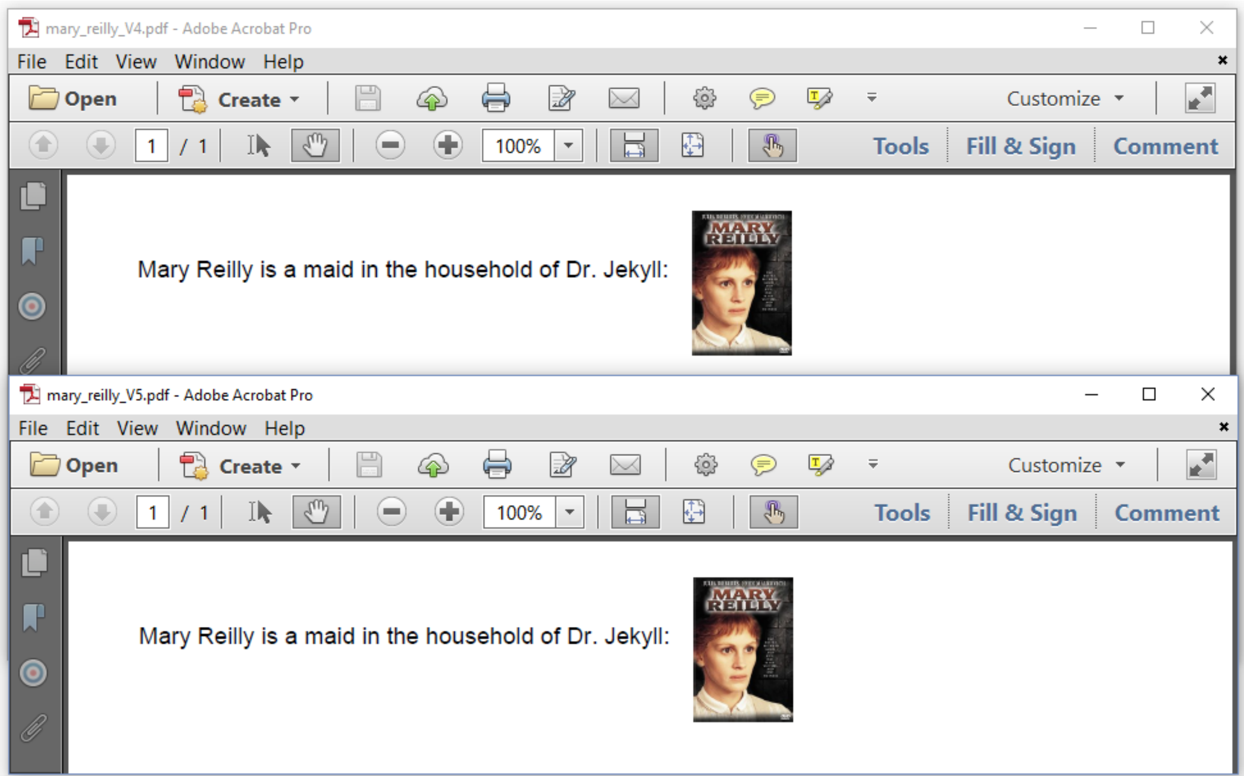



**Figure 3.12: adding an image at absolute positions**

The top PDF was created using the MaryReillyV4[50] example:

```
1 Image img = new Image(ImageDataFactory.create(MARY), 320, 750, 50);
2 document.add(img);
```

In this example, we define the position and the size of the image in the `Image` constructor. We define the position as `x = 320; y = 750`, and we define a width of 50 user units (which is, by default, a width of 50 pt). The height of the image will be adjusted accordingly, preserving the aspect ratio of the image.

The second PDF was created using the MaryReillyV5[51] example.

[50]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1929-c03e11_maryreillyv4.java

[51]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1930-c03e12_maryreillyv5.java

```
1 Image img = new Image(ImageDataFactory.create(MARY));
2 img.setFixedPosition(320, 750, UnitValue.createPointValue(50));
3 document.add(img);
```

In this case, we use the `setFixedPosition()` method to define the position and size of the image. Note that we use the `UnitValue` to define that 50 is a value expressed in pt. The other option is to define the width as a percentage.

There are different variations available for the `Image` constructor and the `setFixedPosition()` method. For instance, you can also define a page number as is done in the MaryReillyV6[52] example.

```
1 Image img = new Image(ImageDataFactory.create(MARY));
2 img.setFixedPosition(2, 300, 750, UnitValue.createPointValue(50));
3 document.add(img);
```

In this example, adding the image on page 2, triggers the creation of a new page. See figure 3.13.

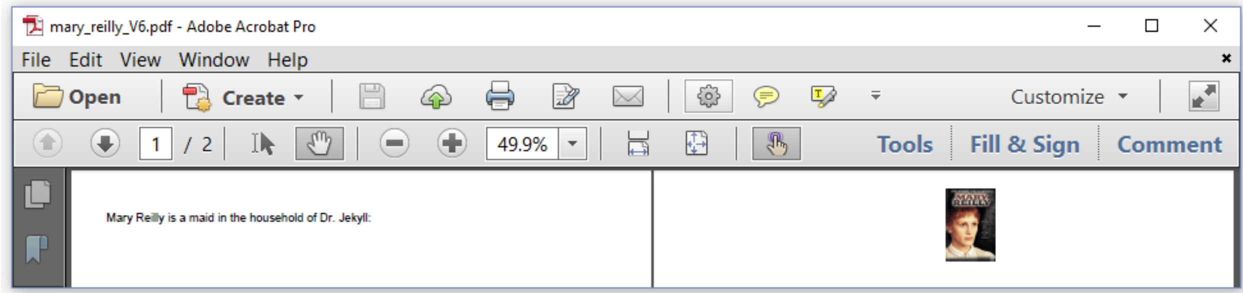



**Figure 3.13: adding an image on a specific page**

If we had been adding the image on page 200, 199 new pages would have been added in order to make sure that the image is actually on page 200. I'm not sure if there's an actual use case for the `setFixedPosition()` method that accepts a page number as a parameter when creating a document from scratch, but that method can also be used when adding content to an existing document.

# Adding an image to an existing PDF

In the iText 7: Jump-Start tutorial[53], we've been working with existing documents. We can import an existing document into iText with a `PdfReader` instance and create a new PDF based on the original document.

In iText 5, we would have worked with a `PdfStamper` object to add content to an existing PDF. This `PdfStamper` object no longer exists in iText 7. Content is always added using either a `PdfDocument` instance (low-level content), or a `Document` instance (high-level content).

[52] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1931-c03e13_maryreillyv6.java
[53] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial

Let's take a look how it's done in the MaryReillyV7[54] example.

```
1  public void manipulatePdf(String src, String dest) throws IOException {
2      PdfReader reader = new PdfReader(src);
3      PdfWriter writer = new PdfWriter(dest);
4      PdfDocument pdfDoc = new PdfDocument(reader, writer);
5      Document document = new Document(pdfDoc);
6      Image img = new Image(ImageDataFactory.create(MARY));
7      img.setFixedPosition(1, 350, 750, UnitValue.createPointValue(50));
8      document.add(img);
9      document.close();
10 }
```

We create a PdfDocument instance using a PdfReader and a PdfWriter object. We use the PdfDocument instance to create a Document. We add an Image to that document using specific coordinates and a specific width on page 1. The result is shown in figure 3.14.

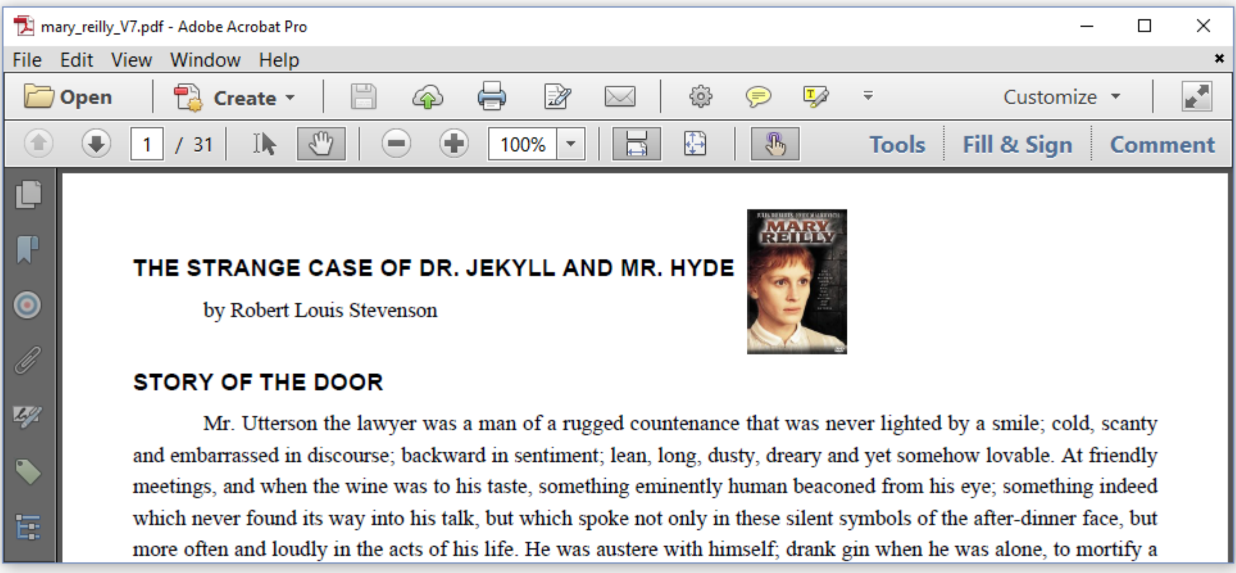



**Figure 3.14: adding an image to an existing PDF**

There are different ways to resize an image.

# Resizing and rotating an image

We already changed the dimensions by defining a width in points, in the MaryReillyV8[55] example, we use a percentage.

[54] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1932-c03e14_maryreillyv7.java
[55] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1933-c03e15_maryreillyv8.java

```
1 Image img = new Image(ImageDataFactory.create(MARY));
2 img.setHorizontalAlignment(HorizontalAlignment.CENTER);
3 img.setWidthPercent(80);
4 document.add(img);
```

As shown in figure 3.15, the image is now centered on the page (using the `setHorizontalAlignment()` method) and it takes 80% of the available width on the page (using the `setWidthPercent()` method).

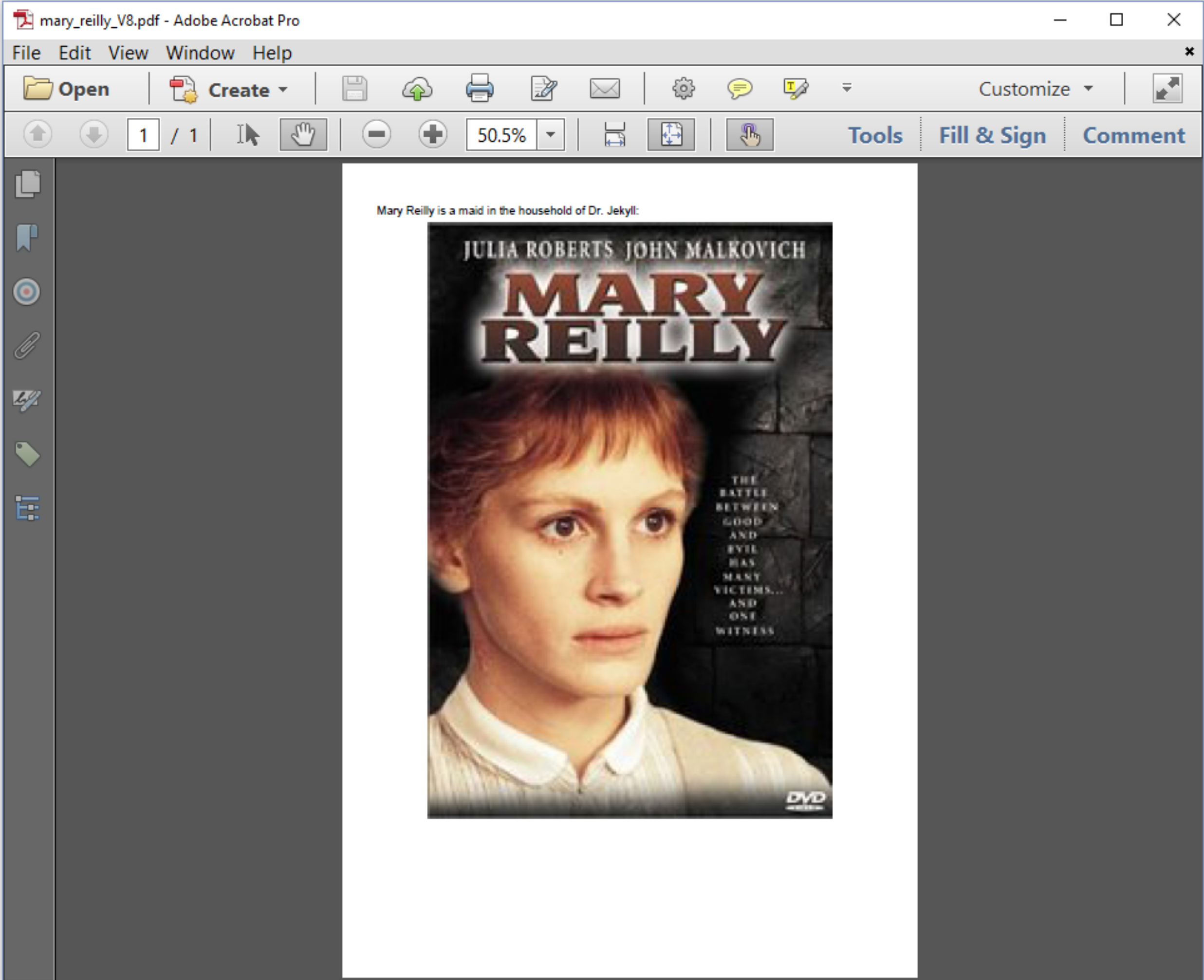



**Figure 3.15: defining the width as a percentage**

Note that iText will automatically scale the image to 100% of the available width when you're trying to add an image that doesn't fit.

Resizing an image doesn't change anything to the original quality of the image. The number of pixels in the image remains identical; iText doesn't change a single pixel in your image. This doesn't mean the resolution doesn't change when you resize an image. If an image is 720 pixels by 720 pixels and you render this image as a 720 pt by 720 pt image, the resolution will be 72 dots per inch. If you change the dimension to 72 pt by 72 pt, you will have a resolution of 720 dots per inch.

So far, we've been adding `Image` objects straight to the document. You can also add `Image` objects to `BlockElement` objects. In the MaryReillyV9[56] example, we add an `Image` to a `Paragraph`.

```
1 Paragraph p = new Paragraph(
2     "Mary Reilly is a maid in the household of Dr. Jekyll: ");
3 Image img = new Image(ImageDataFactory.create(MARY));
4 p.add(img);
5 document.add(p);
```

The result is shown in figure 5.16.

[56]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1934-c03e16_maryreillyv9.java

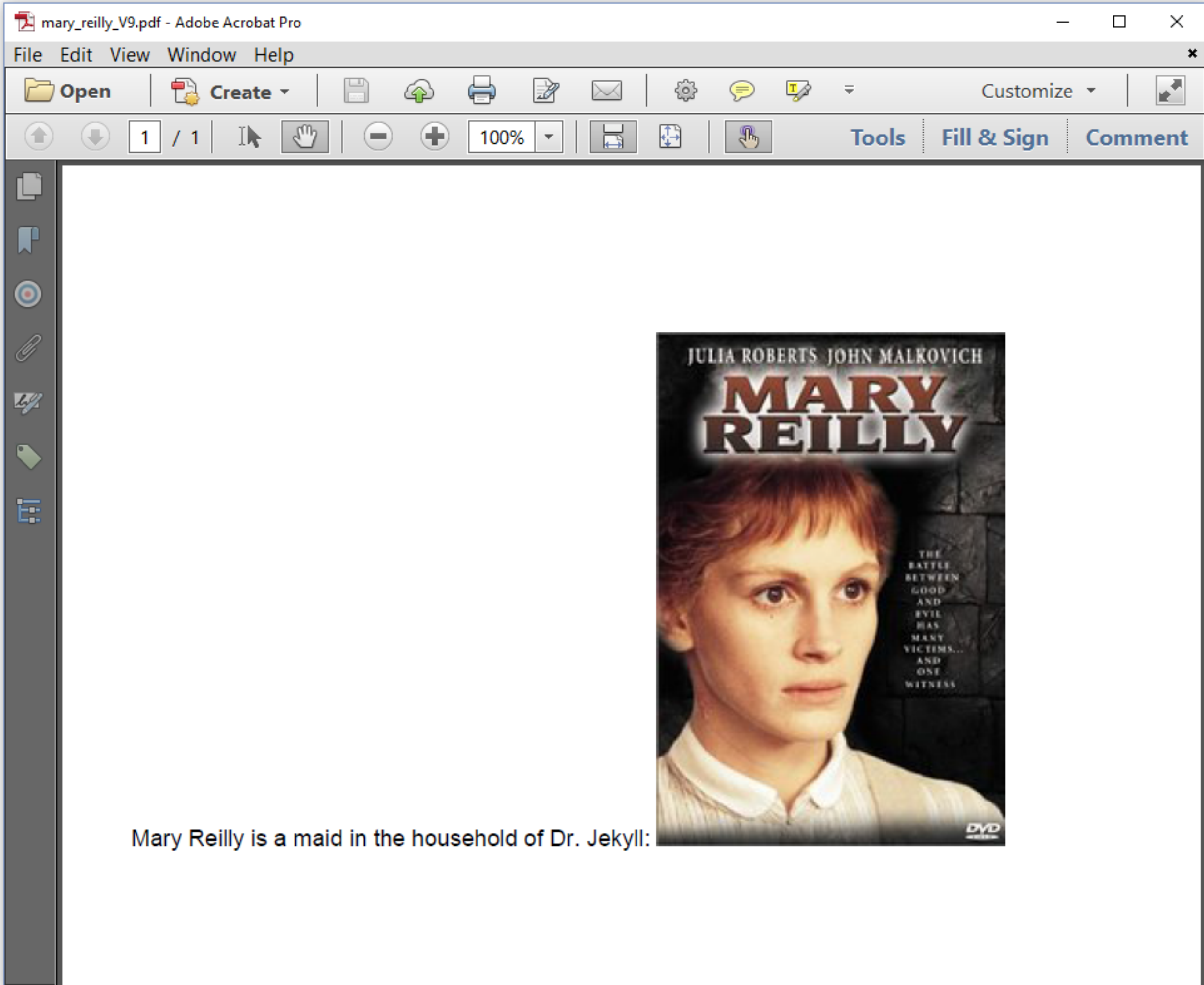



**Figure 3.16: adding an image to a Paragraph**

We see that the leading has been adjusted automatically, but also that the image is somewhat big. The Mary Reilly poster is 182 by 268 pixels in size. In this case, iText will use the same size in user units. As a result, the image shown in figure 3.16 measures 182 by 268 pt. iText may scale images automatically depending on the context. We already mentioned the situation where the image doesn't fit the width of the page; in chapter 5, we'll see how images behave in the context of tables.

There are also different `scale()` methods that allow us to scale an image programmatically. In the MaryReillyV10[57] example, we scale the image to 50% in X- as well as in Y-direction.

[57] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1935-c03e17_maryreillyv10.java

```
Paragraph p = new Paragraph(
    "Mary Reilly is a maid in the household of Dr. Jekyll: ");
Image img = new Image(ImageDataFactory.create(MARY));
img.scale(0.5f, 0.5f);
img.setRotationAngle(-Math.PI / 6);
p.add(img);
document.add(p);
```

We also set a rotation angle of -30 degrees, which results in the PDF shown in figure 3.17.

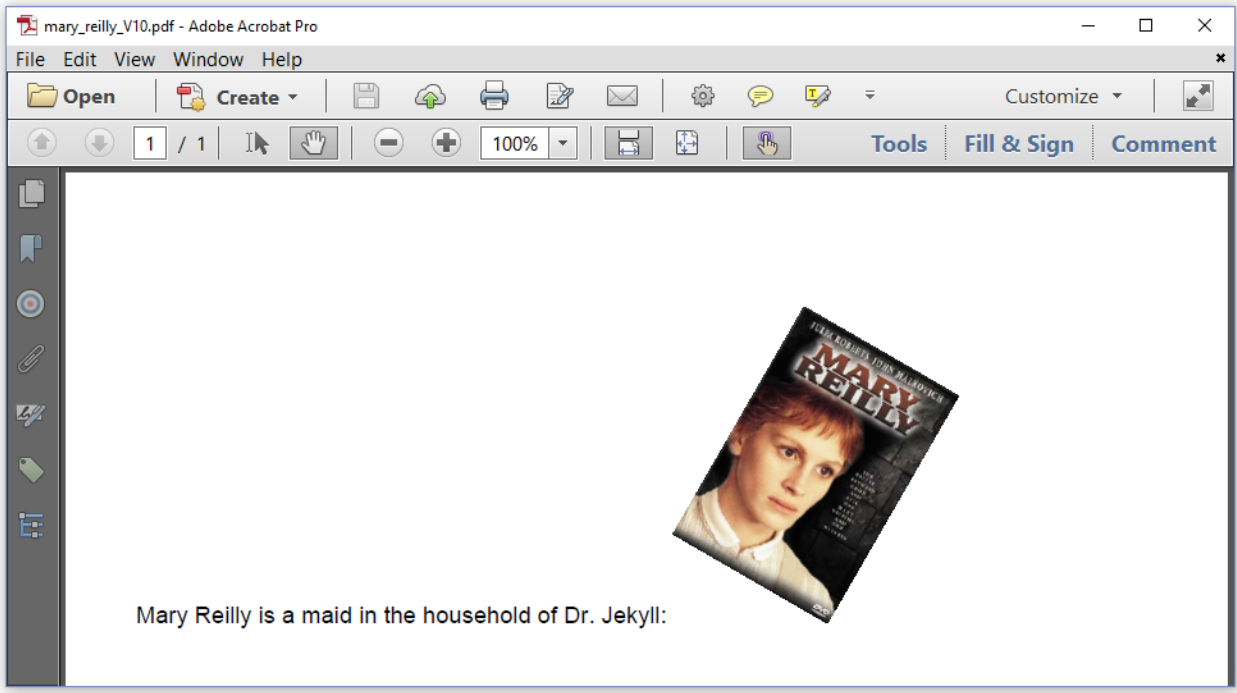



**Figure 3.17: scaled and rotated image**

These are the most common ways to change the dimensions of an `Image` object:

- the `scale()` method: accepts two parameters. The first one is the factor that will be used in the X-direction; the second one is the factor that will be used in the Y-direction. For instance: if you pass a value of `1f` for the X-direction and `0.5f` for the Y-direction, the image will be as wide as initially, but the height will be reduced to 50% of the original height.
- the `scaleAbsolute()` method: also accepts two parameters. The first one is the absolute width in user units; the second one is the absolute height in user units. For instance: if you use a value of `72f` for both the width and the height, the image will, by default, be rendered as an image of 1 inch by 1 inch.
- the `scaleToFit()` method: also accepts two parameters. Using the `scaleAbsolute()` method can lead to awkward results if you don't take the aspect ratio of the image into account.

> The first parameter of the `scaleToFit()` method defines the maximum width of the image; the second one defined the maximum height. The image will be scaled preserving the aspect ratio. This means that the resulting image may be smaller than expected.

So far, we've only been using JPEG images, but iText supports many other image types.

# Image types supported by iText

iText supports the following image formats: JPEG, JPEG2000, BMP, PNG, GIF, JBIG2, TIFF, and WMF. iText also supports raw image data (if you provide the pixels or the CCITT bytes). If you consider PDF to be an image format –which it isn't–, you can even import PDF pages as if it were images.

We've already covered JPEG sufficiently, we'll cover all the other formats in the next couple of examples, starting with the ImageTypes[58] example.

## Raw image data

When we use the `ImageDataFactory`, iText will examine the image that is provided. It will check which image type is encountered, and it will create an `ImageData` object for that specific image type. Most of the times, we'll import an existing image, but we can also create the raw image data on the fly. In figure 3.18, we see an image of a gradient that evolves from yellow to blue.

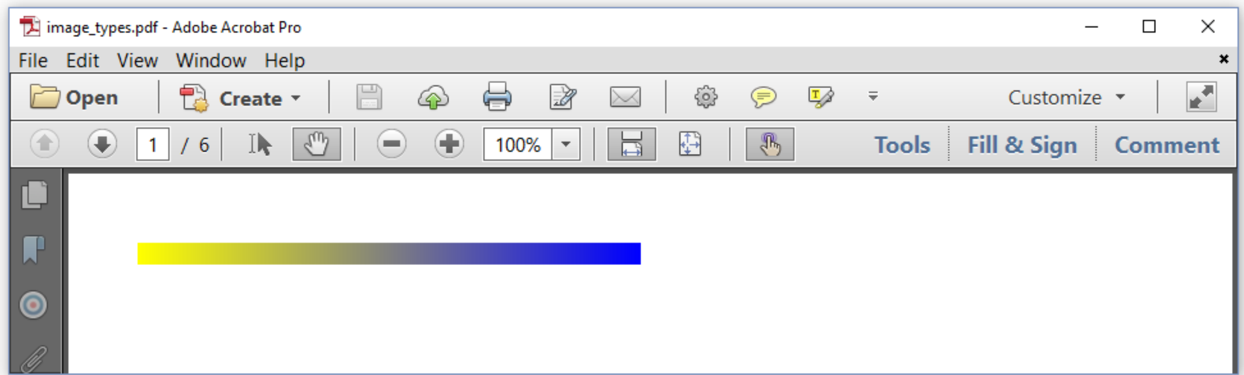



**Figure 3.18: raw image**

The RGB code for yellow is `#FFFF00`; the RGB code for blue is `#0000FF`. If we want to create an RGB images that shows gradient from yellow to blue, we could create an image with 256 pixels that is 256 pixels wide and 1 pixel high. We could then loop from 0 (`0x00`) to 255 (0xFF) creating pixels that vary from [Red = 255, Green = 255, Blue = 0] to [Red = 0, Green = 0, Blue = 255]. The total byte size of that image would be the number of pixels multiplied with the number of values needed to describe the color of each pixel. The following code snippet shows how this is done:

[58] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1936-c03e18_imagetypes.java

```
byte data[] = new byte[256 * 3];
for (int i = 0; i < 256; i++) {
    data[i * 3] = (byte) (255 - i);
    data[i * 3 + 1] = (byte) (255 - i);
    data[i * 3 + 2] = (byte) i;
}
ImageData raw = ImageDataFactory.create(256, 1, 3, 8, data, null);
Image img = new Image(raw);
img.scaleAbsolute(256, 10);
document.add(img);
```

In this snippet, we ask the `ImageDataFactory` to create the `ImageData` for an image of 256 pixels by 1 pixel. We are using 3 components for each pixel. Each component is expressed using 8 bits per component (bpc); that's 1 byte. The fourth parameter of the `create()` method is the `data[]`. The fifth parameter is an array we can use to define *transparency*. We don't need this parameter in our simple example. We use the `ImageData` to create a new `Image` and we scale this image in the Y-direction. If we didn't scale the image, we'd only see a very thin line that is 1 user unit high.

## Which values are valid for the number of components?

You can work with 1, 3, or 4 components.

- *1 component*– means that you define the color of each pixel using one value. We typically call this a gray value, although it's actually a black / white (or rather color / no color) value if you only use 1 bit per component. If you have 8 bits per component, you can define gray values with an intensity varying between 0 (black) and 255 (white).
- *3 components*– means that you define RGB colors using three values: Red, Green, and Blue.
- *4 components*– means that you define CMYK colors using four values: Cyan, Magenta, Yellow, and blacK.

Usually, we don't have to worry about all of this, we can just pass a reference to an image or a `byte[]` containing an existing image, and we let iText do all the low-level work.

Let's take this first batch of image files and see what happens what we add them to a `Document`.

```
public static final String TEST1 = "src/main/resources/img/test/map.jp2";
public static final String TEST2 = "src/main/resources/img/test/butterfly.bmp";
public static final String TEST3 = "src/main/resources/img/test/hitchcock.png";
public static final String TEST4 = "src/main/resources/img/test/info.png";
public static final String TEST5 = "src/main/resources/img/test/hitchcock.gif";
public static final String TEST6 = "src/main/resources/img/test/amb.jb2";
public static final String TEST7 = "src/main/resources/img/test/marbles.tif";
```

We start with the `.jp2` file which is an image in JPEG2000 format.

## JPEG / JPEG2000

The code to add a JPEG2000 image doesn't look any different than the code to add a JPEG image.

```
Image img1 = new Image(ImageDataFactory.create(TEST1));
document.add(img1);
```

The result is shown in figure 3.19.

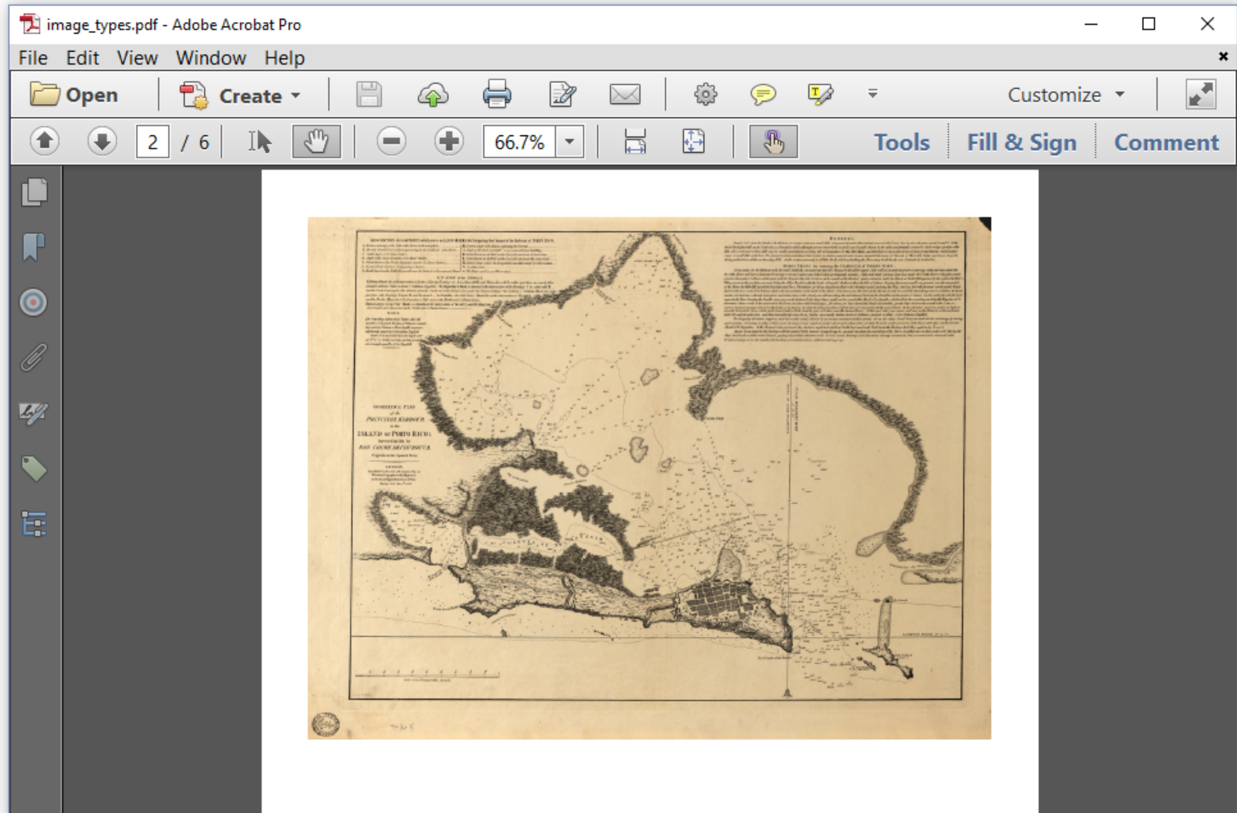



**Figure 3.19: JPEG2000**

JPEG and JPEG200 are supported natively in PDF, this isn't the case for PNG.

## BMP / PNG / GIF

GIF is supported in PDF (it's called LZW), but whenever iText encounters a BMP file, a PNG file, or a GIF file, that file gets converted into a raw image that consists of a bytes that define pixels. These pixels are then compressed and stored in the PDF.

Figure 20 shows one BMP (the butterfly), two PNG files (the first Hitchcock image and the information sign) and one GIF file that is added twice (a second Hitchcock image).

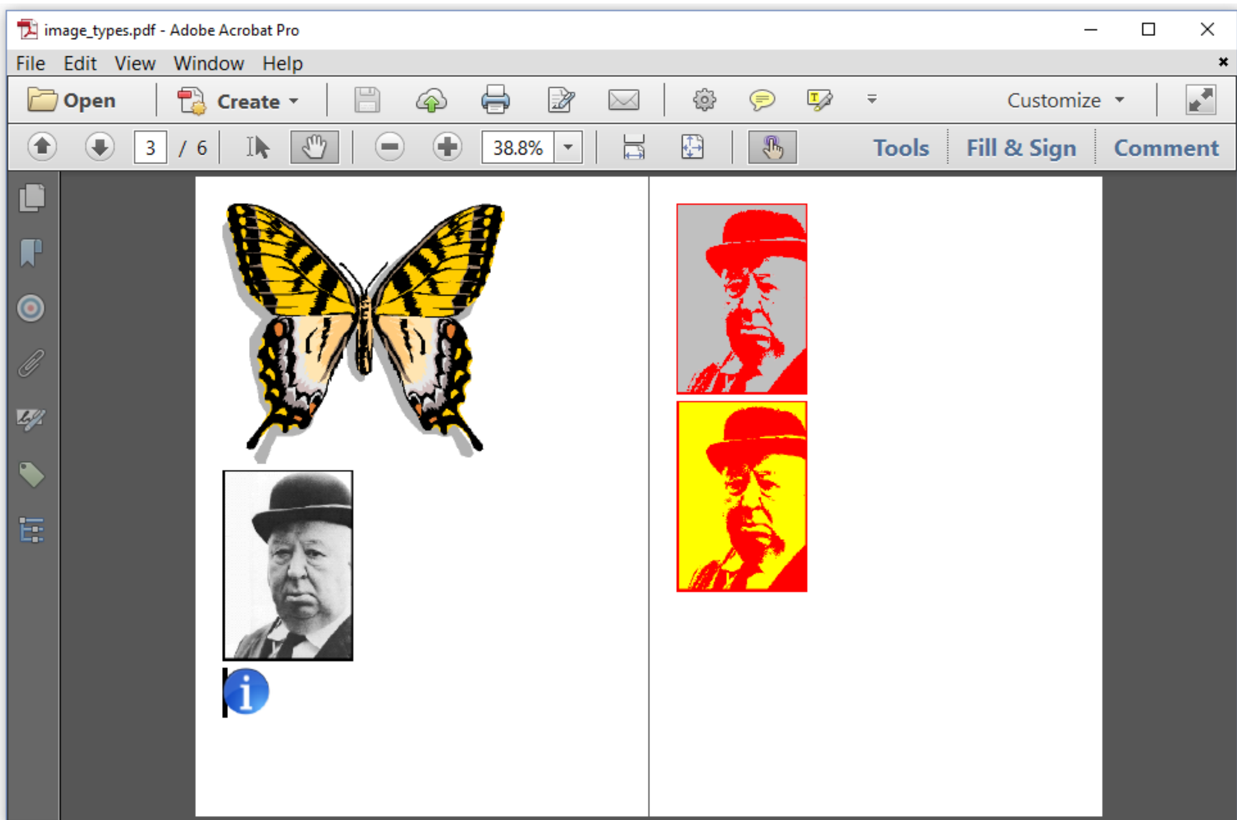



**Figure 3.20: BMP, PNG, GIF**

The code for the page to the left looks like this:

```
// BMP
Image img2 = new Image(ImageDataFactory.create(TEST2));
img2.setMarginBottom(10);
document.add(img2);
// PNG
Image img3 = new Image(ImageDataFactory.create(TEST3));
img3.setMarginBottom(10);
document.add(img3);
// Transparent PNG
```

```
10 Image img4 = new Image(ImageDataFactory.create(TEST4));
11 img4.setBorderLeft(new SolidBorder(6));
12 document.add(img4);
```

As you can see, we're using the `setMarginBottom()` method for `img2` and `img3` to introduce 10 user units of white space between the images. There is something special with `img4`; `info.png` is partly transparent. We introduce a left border with a thickness of 6 user units. We see that border, because the image is transparent. If the image were opaque, that border would have been invisible because it would have been covered by the image.

Transparent images aren't supported in PDF, at least not in the way you'd expect. When you add an image with transparent parts to a PDF, iText will add two images:

- An opaque image: for instance, an image where the transparent part consists of black pixels,
- An image mask: this is an image with 1 component that defines the transparency.

A PDF viewer will use both images to compose the transparent image.

If the image mask has 1 bpc, we talk about a *hard mask*. The pixel of the opaque image underneath the mask is either visible or invisible. If the image mask has more than 1 bpc, we talk about a *soft mask*. The pixel underneath the mask can be partly transparent.

The same is true for background colors. We define a gray background for the first Hitchcock image in the page on the right:

```
1 Image img5 = new Image(ImageDataFactory.create(TEST5));
2 img5.setBackgroundColor(Color.LIGHT_GRAY);
3 document.add(img5);
```

We only see this background, because `hitchcock.gif` is a GIF file with transparency. The second Hitchcock image is added in a completely different way.

## AWT images

If you're working in a Java environment, you may have to work with the AWT image class `java.awt.Image`; iText also support these images.

```
java.awt.Image awtImage =
    Toolkit.getDefaultToolkit().createImage(TEST5);
Image awt =
    new Image(ImageDataFactory.create(awtImage, java.awt.Color.yellow));
awt.setMarginTop(10);
document.add(awt);
```

We read the `hitchcock.gif` image into a `java.awt.Image` object in line 1-2. We get an `ImageData` object from the `ImageDataFactory` in line 4. The first parameter is the AWT image, the second parameter defines the color that needs to be used for the transparent part (if there is any). You can also add a Boolean as third parameter. If that parameter is `true`, the image will be converted to a black and white image.

## JBIG2 / TIFF

Figure 3.21 shows a JBIG2 image and a TIFF image.

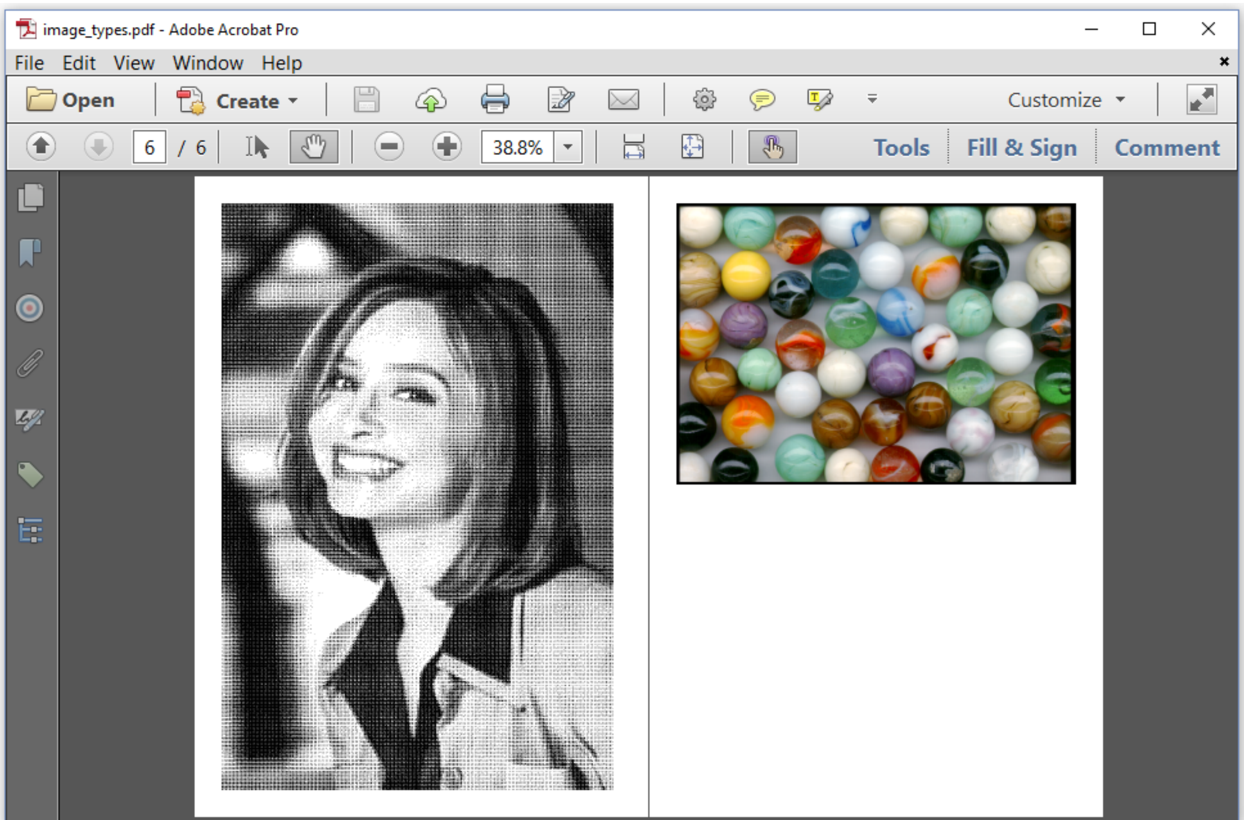



Figure 3.21: JBIG2, TIFF

The code is pretty straightforward:

```
// JBIG2
Image img6 = new Image(ImageDataFactory.create(TEST6));
document.add(img6);
// TIFF
Image img7 = new Image(ImageDataFactory.create(TEST7));
document.add(img7);
```

It isn't always that easy to convert the full JBIG2 or TIFF image to PDF though. A JBIG2 image and a TIFF image can contain different pages. In that case, we need to loop over the pages and extract every page as a separate image. The same is true for animated GIF images that consist of different frames.

## Animated GIFs / Paged images

In the PagedImages[59] example, we define three new constants that refer to three different images.

```
public static final String TEST1 =
    "src/main/resources/img/test/animated_fox_dog.gif";
public static final String TEST2 = "src/main/resources/img/test/amb.jb2";
public static final String TEST3 = "src/main/resources/img/test/marbles.tif";
```

Figure 3.22 shows the different frames of an animated GIF that shows an animation of a fox jumping over a dog.

[59] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1937-c03e19_pagedimages.java

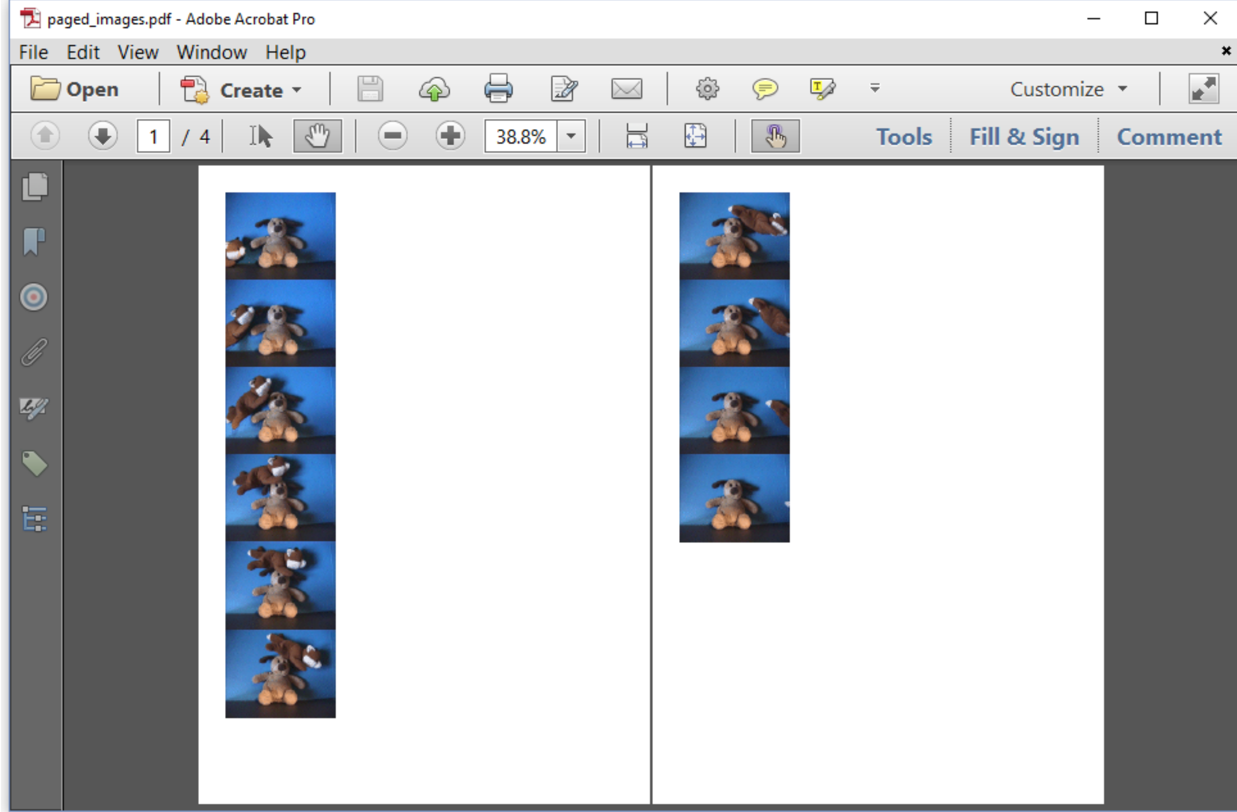



**Figure 3.22: frames from an animated GIF**

Animated GIFs aren't supported in PDF, so you can't add the animation as-is to the document. We can only add every frame to the document as a separate image. That's what we do in the next code snippet.

```
1 URL url1 = UrlUtil.toURL(TEST1);
2 List<ImageData> list = ImageDataFactory.createGifFrames(url1);
3 for (ImageData data : list) {
4     img = new Image(data);
5     document.add(img);
6 }
```

We create an `URL` object that uses the path to the file as input. We then create a `List` of `ImageData` objects containing the `ImageData` of every frame in the animated GIF. Finally, we add each frame as a separate `Image` to the `Document`.

The code to read the different pages from a JBIG2 and a TIFF file is more complex.

```
1     // JBIG2
2     URL url2 = UrlUtil.toURL(TEST2);
3     IRandomAccessSource ras2 =
4         new RandomAccessSourceFactory().createSource(url2);
5     RandomAccessFileOrArray raf2 = new RandomAccessFileOrArray(ras2);
6     int pages2 = Jbig2ImageData.getNumberOfPages(raf2);
7     for (int i = 1; i <= pages2; i++) {
8         img = new Image(ImageDataFactory.createJbig2(url2, i));
9         document.add(img);
10    }
11    // TIFF
12    URL url3 = UrlUtil.toURL(TEST3);
13    IRandomAccessSource ras3 =
14        new RandomAccessSourceFactory().createSource(url3);
15    RandomAccessFileOrArray raf3 = new RandomAccessFileOrArray(ras3);
16    int pages3 = TiffImageData.getNumberOfPages(raf3);
17    for (int i = 1; i <= pages3; i++) {
18        img = new Image(
19            ImageDataFactory.createTiff(url3, true, i, true));
20        document.add(img);
21    }
22    document.close();
```

We first need to know the number of pages in the JBIG2 or TIFF file. This requires us to create a `RandomAccessFileOrArray` object. With this object, we can ask the `Jbig2ImageData` or the `TiffImageData` class for the number of pages in the JBIG2 or TIFF file. We can then loop over the number of pages in that file, and we use the `createJbig2()` or `createTiff()` method to get the `ImageData` object needed to create an `Image`.

Up until now, all the `Image` objects that we have created, resulted in an image XObject stored in the PDF document. In the next example, we'll create a different type of XObject.

## WMF / PDF

All the image types we've worked with so far were *raster images*. Raster images consist of pixels of a certain color put next to each other in a grid. In the XObjectTypes[60] example, we have the following source files:

```
1 public static final String WMF = "src/main/resources/img/test/butterfly.wmf";
2 public static final String SRC = "src/main/resources/pdfs/jekyll_hyde.pdf";
```

[60] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-3#1938-c03e20_xobjecttypes.java

WMF is a vector image format. Vector images don't have pixels. They are made up of basic geometric shapes such as lines and curves. These lines and curves are expressed as a mathematical equation, which means that you can easily scale them without losing any quality.

The concept of resolution doesn't exist in the context of vector images. The resolution only comes into play when you render the image to a device. The resolution of the device –a printer, a screen– will determine the resolution you perceive when looking at the vector image.

A PDF can contain raster images, and each of these raster images will have its own resolution, but the PDF itself doesn't have a resolution. The content of the PDF is also made up of geometric shapes defined using PDF syntax.

In figure 3.23, you see a WMF file representing a butterfly and a page from an existing PDF file that were added to a `Document` using the `Image` object.

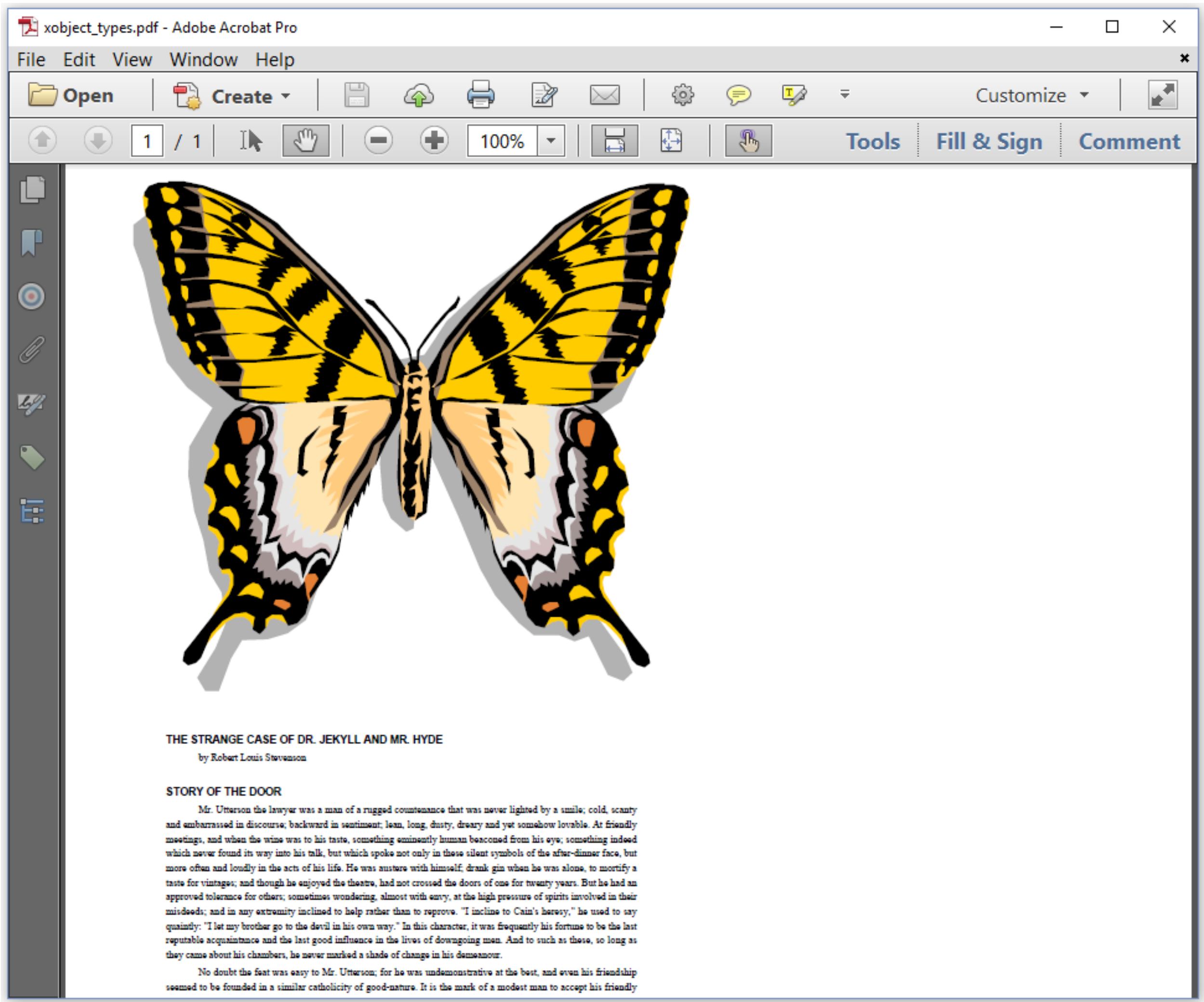



**Figure 3.23: WMF, PDF**

If you look inside this PDF file, you won't find any image XObject; instead you'll discover two *form XObjects*. A form XObject uses the same mechanism as an image XObject, except that a form XObject doesn't consist of pixels; it's a snippet of PDF syntax that is external to the page content.

If we want to add a WMF file to a `Document` using the `Image` class, we need to create a `PdfFormXObject` first. The `WmfImageData` object will help us create the `ImageData` that is needed to create this form XObject. We can use that `xObject1` to create an `Image` instance.

```
1 PdfFormXObject xObject1 =
2     new PdfFormXObject(new WmfImageData(WMF), pdf);
3 Image img1 = new Image(xObject1);
4 document.add(img1);
```

We need to do something similar to import a page from an existing PDF file if we want to import that page as if it were an image.

```
PdfReader reader = new PdfReader(SRC);
PdfDocument existing = new PdfDocument(reader);
PdfPage page = existing.getPage(1);
PdfFormXObject xObject2 = page.copyAsFormXObject(pdf);
Image img2 = new Image(xObject2);
img2.scaleToFit(400, 400);
document.add(img2);
```

We start by creating a `PdfReader` object (line 1) and a `PdfDocument` based on that `reader` (line 2). We obtain a `PdfPage` from that `existing` document (line 3), and we copy that `page` as a `PdfXFormObject`. We can use that `xObject2` to create an `Image` instance.

The content of the existing page will be added as if it were a vector image. All interactive features that may exist in the original page, such as links, form fields, and other annotations, will be lost.

This concludes the overview of the objects that implement the `ILeafElement` interface.

# Summary

In this chapter, we've covered the building blocks that implement the `ILeafElement` interface. These elements are atomic building blocks; they aren't composed of other elements.

- `Tab`– is an element that allows you to put some space between two other building blocks, either using white space, or by introducing a leader. You can also use the `Tab` element to align an element.
- `Text`– is an element that contains a snippet of text using a single font, single font size, single font color. It's the atomic text building block.
- `Link`– is a `Text` element for which we can define a `PdfAction`, for instance: an action that opens a web site when we click on the text. We'll discuss more examples of links and actions in chapter 6.
- `Image`– is an element that can be used to create an image XObject so that you can use raster images in your PDF. For reasons of convenience, we also allow developers to wrap a `PdfFormXObject` inside an `Image` object so that they can use form XObjects using the same functionality that is available for image XObjects.

We haven't finished talking about these objects. We'll continue using them in the chapters that follow, starting with the next chapter that will discuss the `Div`, `LineSeparator`, `Paragraph`, `List`, and `ListItem` object.

# Chapter 4: Adding AbstractElement objects (part 1)

In previous chapters, we've already discussed five classes that implement the AbstractElement class. We've discussed the AreaBreak class in chapter 2, and we've discussed the four classes implementing the ILeafElement –Tab, Link, Text, and Image– in chapter 3. In this chapter, we'll start with a first series of AbstractElement implementations. We'll take a look at the Div class to group elements and at the LineSeparator to draw lines between elements. We've already used the Paragraph class many times in previous chapters, but we'll revisit it in this chapter. Finally, we'll introduce the List and the ListItem class. We'll save the Table and Cell class for the next chapter.

## Grouping elements with the Div class

The Div class is a BlockElement implementation that can be used to group different elements. In Figure 4.1, we see an overview of movies based on the Jekyll and Hyde story. Each entry consists of at most three elements:

- a Paragraph showing the title of the movie,
- a Paragraph showing the director, the country, and a year,
- an Image showing the movie poster (if any).

We combined these three elements in a Div and we defined a left border, left padding and bottom margin for that Div.

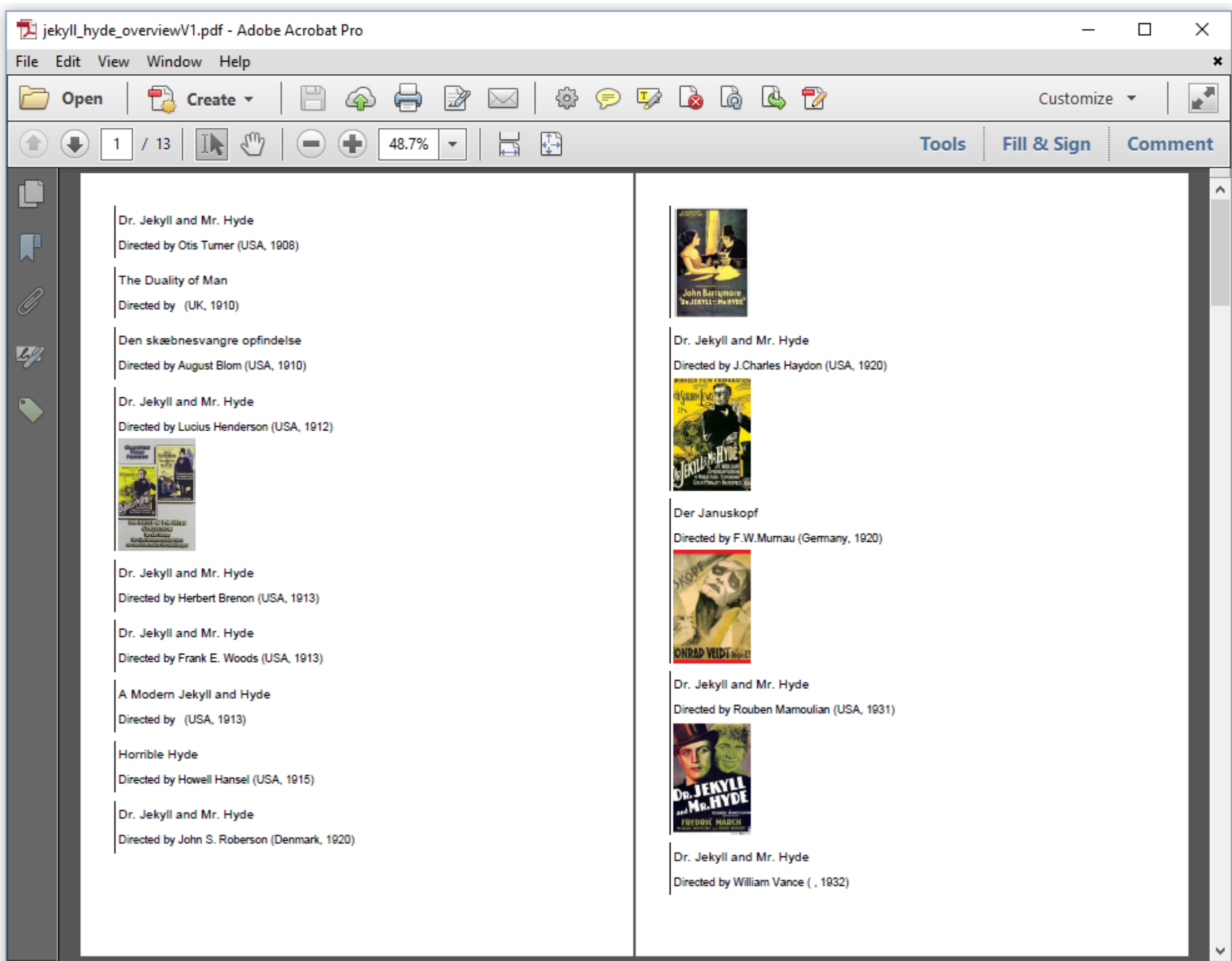



**Figure 4.1: Grouping elements in a Div**

The DivExample1[61] example shows how this is done:

```
1  public void createPdf(String dest) throws IOException {
2      PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
3      Document document = new Document(pdf);
4      List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
5      resultSet.remove(0);
6      for (List<String> record : resultSet) {
7          Div div = new Div()
8              .setBorderLeft(new SolidBorder(2))
9              .setPaddingLeft(3)
10             .setMarginBottom(10);
11         String url = String.format(
```

[61] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1962-c04e01_divexample1.java

```
12             "http://www.imdb.com/title/tt%s", record.get(0));
13         Link movie = new Link(record.get(2), PdfAction.createURI(url));
14         div.add(new Paragraph(movie.setFontSize(14)))
15             .add(new Paragraph(String.format(
16                 "Directed by %s (%s, %s)",
17                 record.get(3), record.get(4), record.get(1))));
18         File file = new File(String.format(
19             "src/main/resources/img/%s.jpg", record.get(0)));
20         if (file.exists()) {
21             Image img = new Image(
22                 ImageDataFactory.create(file.getPath()));
23             img.scaleToFit(10000, 120);
24             div.add(img);
25         }
26         document.add(div);
27     }
28     document.close();
29 }
```

As usual, we create a `PdfDocument` and a `Document` instance (line 2-3). We reuse the CSV file that was introduced in the previous chapter, and we loop over all the movies listed in that CSV file, excluding the header row (line 4-6). We create a new `Div` object (line 7) and we define the left border as a solid border with a thickness of 2 user units (line 8), we set the left padding to 3 user units (line 9), and we introduce a bottom margin of 10 user units (line 10). We add the title `Paragraph` to this `Div` (line 14), as well as a `Paragraph` with additional info (line 15 - 17). If we find a movie poster, we add it as an `Image` (line 24). We add each `Div` to the document (line 26) and we close the document (line 28).

If we look at the bottom of the first page and at the top of the second page in Figure 4.1, we see that the `Div` containing the information about the movie "Dr. Jekyll and Mr. Hyde" directed by John S. Roberson, is distributed over two pages. The movie poster didn't fit on the first page, so it was forwarded to the second page. Maybe this isn't the behavior we desire. Maybe we want to keep the elements added to the same `Div` together as shown in figure 4.2.

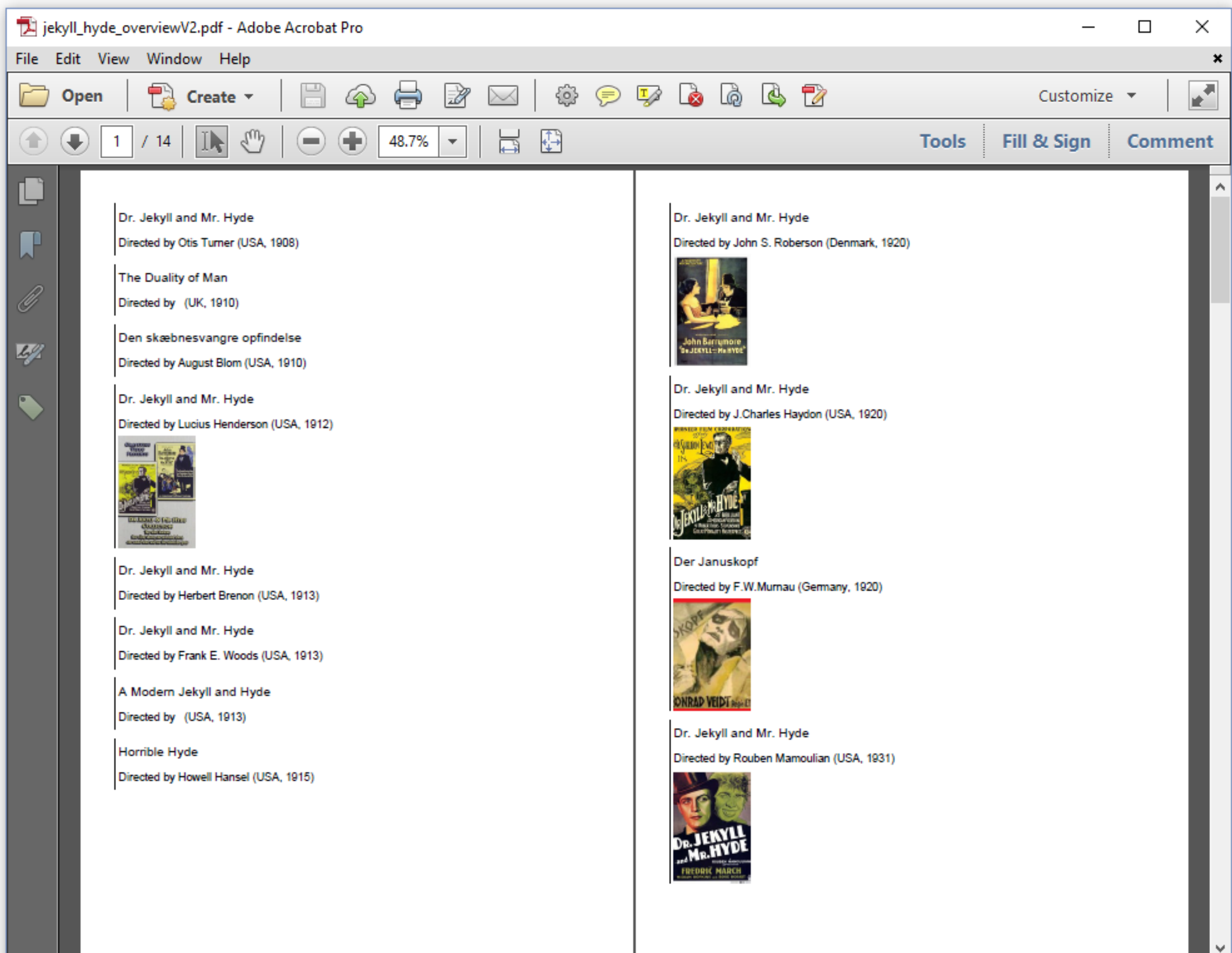



**Figure 4.2: Keeping a Div on one page**

We use only one extra method to achieve this; see the DivExample2[62] example.

```
1  Div div = new Div()
2      .setKeepTogether(true)
3      .setBorderLeft(new SolidBorder(2))
4      .setPaddingLeft(3)
5      .setMarginBottom(10);
```

By adding `setKeepTogether(true)`, we tell iText to try to keep the content of a `Div` on the same page. If the content of that `Div` fits on the next page, all the elements in the `Div` will be forwarded to the next page. This is the case in figure 4.2 where the title and the info about the 1920 movie "Dr. Jekyll and Mr. Hyde" directed by John S. Roberson is no longer added on the first page. Instead it's forwarded to the next page.

[62]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1963-c04e02_divexample2.java

This approach won't work if the content of a Div doesn't fit on the next page. In that case, the elements are distributed over the current page and subsequent pages as if the setKeepTogether() method wasn't used. There's a workaround in case you really want to keep one element on the same page as the next element. We'll look at an example demonstrating this workaround after we've discussed the LineSeparator object.

# Drawing horizontal lines with the LineSeparator object

The building blocks created for iText are inspired by the tags that are available for HTML. That's not a secret. The Text object roughly corresponds with <span>, Paragraph corresponds with <p>, Div corresponds with <div>, and so on. The best way to explain what the LineSeparator is about, is to say that it corresponds with the <hr> tag. Figure 4.3 shows a horizontal rule consisting of a red line, 1 user unit thick, that takes 50% of the available width, for which a top margin of 5 user units was defined.

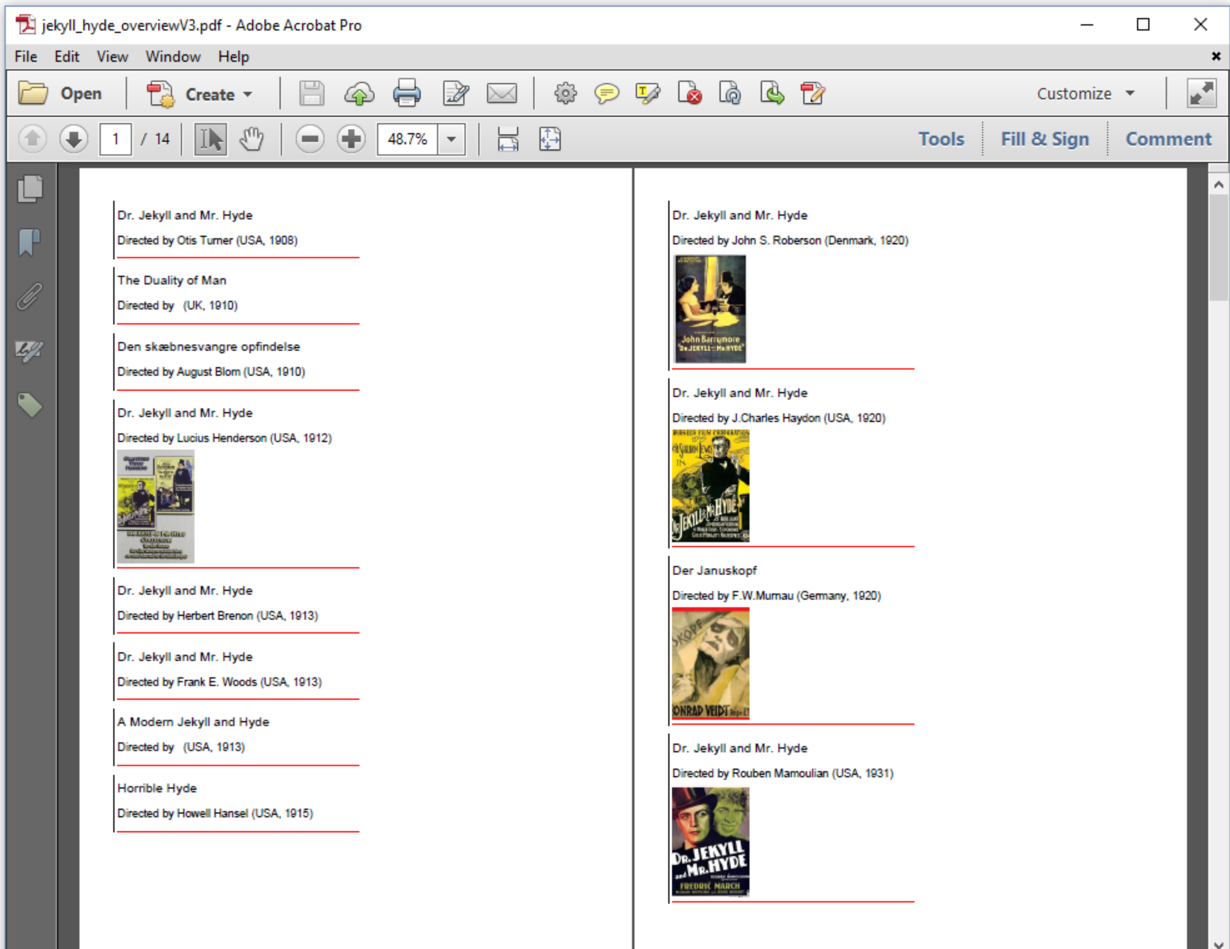



Figure 4.3: Using a LineSeparator

The LineSeparatorExample[63] example shows how it's done.

```
1 SolidLine line = new SolidLine(1f);
2 line.setColor(Color.RED);
3 LineSeparator ls = new LineSeparator(line);
4 ls.setWidthPercent(50);
5 ls.setMarginTop(5);
```

We create a `SolidLine` object, passing a parameter that defines the thickness. We remember from the previous chapter that `SolidLine` is one of the implementations of the `ILineDrawer` interface. We set its color to red and we use this `ILineDrawer` to create a `LineSeparator` instance. In this case, we define the width of the line using the `setWidthPercent()` method. We could also have used the `setWidth()` method to define an absolute width expressed in user units. Finally, we set the top margin to 5 user units.

In the LineSeparatorExample[64] example, we add the `ls` object to our `Div` element containing information about a movie.

```
1 div.add(ls);
```

There isn't much more to be said about `LineSeparator`. Just make sure that you use the right methods to set properties. For instance: you can't change the color of a line at the level of the `LineSeparator`, you have to set it at the level of the `ILineDrawer`. The same goes for the thickness of the line. Check Appendix B[65] to find out which `AbstractElement` methods are implemented for the `LineSeparator` class, and which methods are ignored.

## Keeping content together

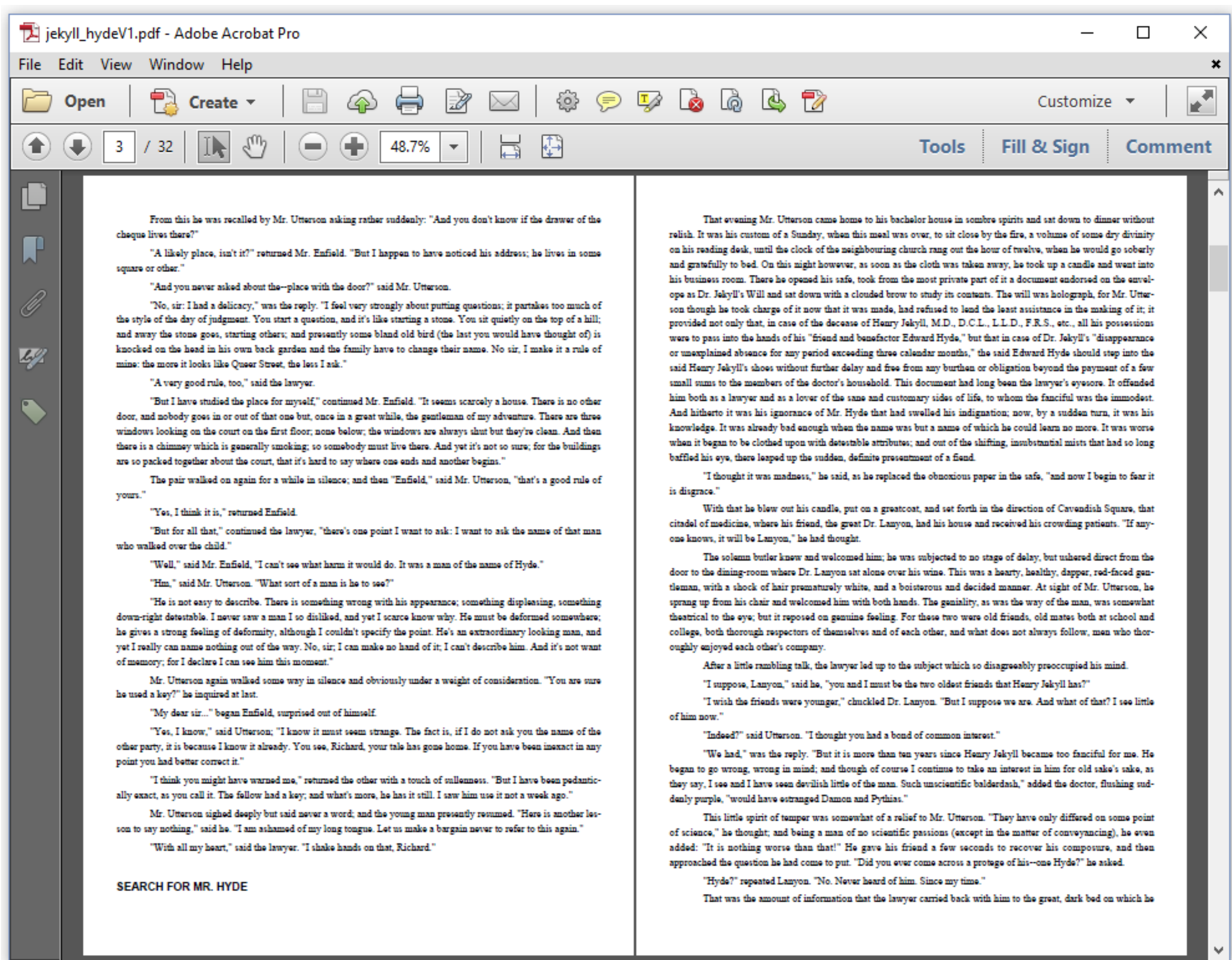
We've been working with the `Paragraph` class many times in previous examples. For instance: in chapter 2, we've used the `Paragraph` class to convert a text file to PDF by creating a `Paragraph` object for each line in the text file, and by adding all of these `Paragraph` objects to a `Document` instance one way or another. The screen shots in the previous chapters showed that we can make some really nice PDF documents, but there's always room for improvement.

Figure 4.4 demonstrates one of the flaws that we still need to fix: we have the title of a chapter on page 3, but the content of that chapter starts on page 4.

[63]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1964-c04e03_lineseparatorexample.java

[64]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1964-c04e03_lineseparatorexample.java

[65]http://developers.itextpdf.com/content/itext-7-building-blocks/appendix/b-blockelement




From this he was recalled by Mr. Utterson asking rather suddenly: "And you don't know if the drawer of the cheque lives there?"

"A likely place, isn't it?" returned Mr. Enfield. "But I happen to have noticed his address; he lives in some square or other."

"And you never asked about the--place with the door?" said Mr. Utterson.

"No, sir: I had a delicacy," was the reply. "I feel very strongly about putting questions; it partakes too much of the style of the day of judgment. You start a question, and it's like starting a stone. You sit quietly on the top of a hill; and away the stone goes, starting others; and presently some bland old bird (the last you would have thought of) is knocked on the head in his own back garden and the family have to change their name. No sir, I make it a rule of mine: the more it looks like Queer Street, the less I ask."

"A very good rule, too," said the lawyer.

"But I have studied the place for myself," continued Mr. Enfield. "It seems scarcely a house. There is no other door, and nobody goes in or out of that one but, once in a great while, the gentleman of my adventure. There are three windows looking on the court on the first floor; none below; the windows are always shut but they're clean. And then there is a chimney which is generally smoking; so somebody must live there. And yet it's not so sure; for the buildings are so packed together about the court, that it's hard to say where one ends and another begins."

The pair walked on again for a while in silence; and then "Enfield," said Mr. Utterson, "that's a good rule of yours."

"Yes, I think it is," returned Enfield.

"But for all that," continued the lawyer, "there's one point I want to ask: I want to ask the name of that man who walked over the child."

"Well," said Mr. Enfield, "I can't see what harm it would do. It was a man of the name of Hyde."

"Hm," said Mr. Utterson. "What sort of a man is he to see?"

"He is not easy to describe. There is something wrong with his appearance; something displeasing, something down-right detestable. I never saw a man I so disliked, and yet I scarce know why. He must be deformed somewhere; he gives a strong feeling of deformity, although I couldn't specify the point. He's an extraordinary looking man, and yet I really can name nothing out of the way. No, sir; I can make no hand of it; I can't describe him. And it's not want of memory; for I declare I can see him this moment."

Mr. Utterson again walked some way in silence and obviously under a weight of consideration. "You are sure he used a key?" he inquired at last.

"My dear sir..." began Enfield, surprised out of himself.

"Yes, I know," said Utterson; "I know it must seem strange. The fact is, if I do not ask you the name of the other party, it is because I know it already. You see, Richard, your tale has gone home. If you have been inexact in any point you had better correct it."

"I think you might have warned me," returned the other with a touch of sullenness. "But I have been pedantic-ally exact, as you call it. The fellow had a key; and what's more, he has it still. I saw him use it not a week ago."

Mr. Utterson sighed deeply but said never a word; and the young man presently resumed. "Here is another les-son to say nothing," said he. "I am ashamed of my long tongue. Let us make a bargain never to refer to this again."

"With all my heart," said the lawyer. "I shake hands on that, Richard."

SEARCH FOR MR. HYDE

That evening Mr. Utterson came home to his bachelor house in sombre spirits and sat down to dinner without relish. It was his custom of a Sunday, when this meal was over, to sit close by the fire, a volume of some dry divinity on his reading desk, until the clock of the neighbouring church rang out the hour of twelve, when he would go soberly and gratefully to bed. On this night however, as soon as the cloth was taken away, he took up a candle and went into his business room. There he opened his safe, took from the most private part of it a document endorsed on the envel-ope as Dr. Jekyll's Will and sat down with a clouded brow to study its contents. The will was holograph, for Mr. Utter-son though he took charge of it now that it was made, had refused to lend the least assistance in the making of it; it provided not only that, in case of the decease of Henry Jekyll, M.D., D.C.L., L.L.D., F.R.S., etc., all his possessions were to pass into the hands of his "friend and benefactor Edward Hyde," but that in case of Dr. Jekyll's "disappearance or unexplained absence for any period exceeding three calendar months," the said Edward Hyde should step into the said Henry Jekyll's shoes without further delay and free from any burthen or obligation beyond the payment of a few small sums to the members of the doctor's household. This document had long been the lawyer's eyesore. It offended him both as a lawyer and as a lover of the sane and customary sides of life, to whom the fanciful was the immodest. And hitherto it was his ignorance of Mr. Hyde that had swelled his indignation; now, by a sudden turn, it was his knowledge. It was already bad enough when the name was but a name of which he could learn no more. It was worse when it began to be clothed upon with detestable attributes; and out of the shifting, insubstantial mists that had so long baffled his eye, there leaped up the sudden, definite presentment of a fiend.

"I thought it was madness," he said, as he replaced the obnoxious paper in the safe, "and now I begin to fear it is disgrace."

With that he blew out his candle, put on a greatcoat, and set forth in the direction of Cavendish Square, that citadel of medicine, where his friend, the great Dr. Lanyon, had his house and received his crowding patients. "If any-one knows, it will be Lanyon," he had thought.

The solemn butler knew and welcomed him; he was subjected to no stage of delay, but ushered direct from the door to the dining-room where Dr. Lanyon sat alone over his wine. This was a hearty, healthy, dapper, red-faced gen-tleman, with a shock of hair prematurely white, and a boisterous and decided manner. At sight of Mr. Utterson, he sprang up from his chair and welcomed him with both hands. The geniality, as was the way of the man, was somewhat theatrical to the eye; but it reposed on genuine feeling. For these two were old friends, old mates both at school and college, both thorough respectors of themselves and of each other, and what does not always follow, men who thor-oughly enjoyed each other's company.

After a little rambling talk, the lawyer led up to the subject which so disagreeably preoccupied his mind.

"I suppose, Lanyon," said he, "you and I must be the two oldest friends that Henry Jekyll has?"

"I wish the friends were younger," chuckled Dr. Lanyon. "But I suppose we are. And what of that? I see little of him now."

"Indeed?" said Utterson. "I thought you had a bond of common interest."

"We had," was the reply. "But it is more than ten years since Henry Jekyll became too fanciful for me. He began to go wrong, wrong in mind; and though of course I continue to take an interest in him for old sake's sake, as they say, I see and I have seen devilish little of the man. Such unscientific balderdash," added the doctor, flushing sud-denly purple, "would have estranged Damon and Pythias."

This little spirit of temper was somewhat of a relief to Mr. Utterson. "They have only differed on some point of science," he thought; and being a man of no scientific passions (except in the matter of conveyancing), he even added: "It is nothing worse than that!" He gave his friend a few seconds to recover his composure, and then approached the question he had come to put. "Did you ever come across a protege of his--one Hyde?" he asked.

"Hyde?" repeated Lanyon. "No. Never heard of him. Since my time."

That was the amount of information that the lawyer carried back with him to the great, dark bed on which he

**Figure 4.4: a widowed title**

We'd like to avoid this kind of behavior. We'd like the title to be on the same page as the start of the content of the chapter. We do a first attempt to fix this problem in the ParagraphAndDiv1[66] example.

```
 1 public void createPdf(String dest) throws IOException {
 2     PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
 3     Document document = new Document(pdf);
 4     PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
 5     PdfFont bold = PdfFontFactory.createFont(FontConstants.HELVETICA_BOLD);
 6     document.setTextAlignment(TextAlignment.JUSTIFIED)
 7         .setHyphenation(new HyphenationConfig("en", "uk", 3, 3));
 8     BufferedReader br = new BufferedReader(new FileReader(SRC));
 9     String line;
10     Div div = new Div();
```

[66] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1965-c04e04_paragraphanddiv1.java

```
11      while ((line = br.readLine()) != null) {
12          Paragraph title = new Paragraph(line)
13              .setFont(bold).setFontSize(12)
14              .setMarginBottom(0);
15          div = new Div()
16              .add(title)
17              .setFont(font).setFontSize(11)
18              .setMarginBottom(18);
19          while ((line = br.readLine()) != null) {
20              div.add(
21                  new Paragraph(line)
22                      .setMarginBottom(0)
23                      .setFirstLineIndent(36)
24              );
25              if (line.isEmpty()) {
26                  document.add(div);
27                  break;
28              }
29          }
30      }
31      document.add(div);
32      document.close();
33  }
```

This example is very similar to the examples we made in chapter 2. The main difference is that we no longer add the `Paragraph` objects straight to the `Document`. Instead, we store the `Paragraph` objects in a `Div` object, and we add the `Div` object to the `Document` at the end of each chapter.

We could add `.setKeepTogether(true)` between line 15 and 16, but that wouldn't have any effect as the full content of the `Div` doesn't fit on a single page. As documented before, the `setKeepTogether()` method is ignored. We've had long discussions at iText on how to solve this problem. We decided that the most elegant way to avoid widowed objects consisted of introducing a `setKeepWithNext()` method.

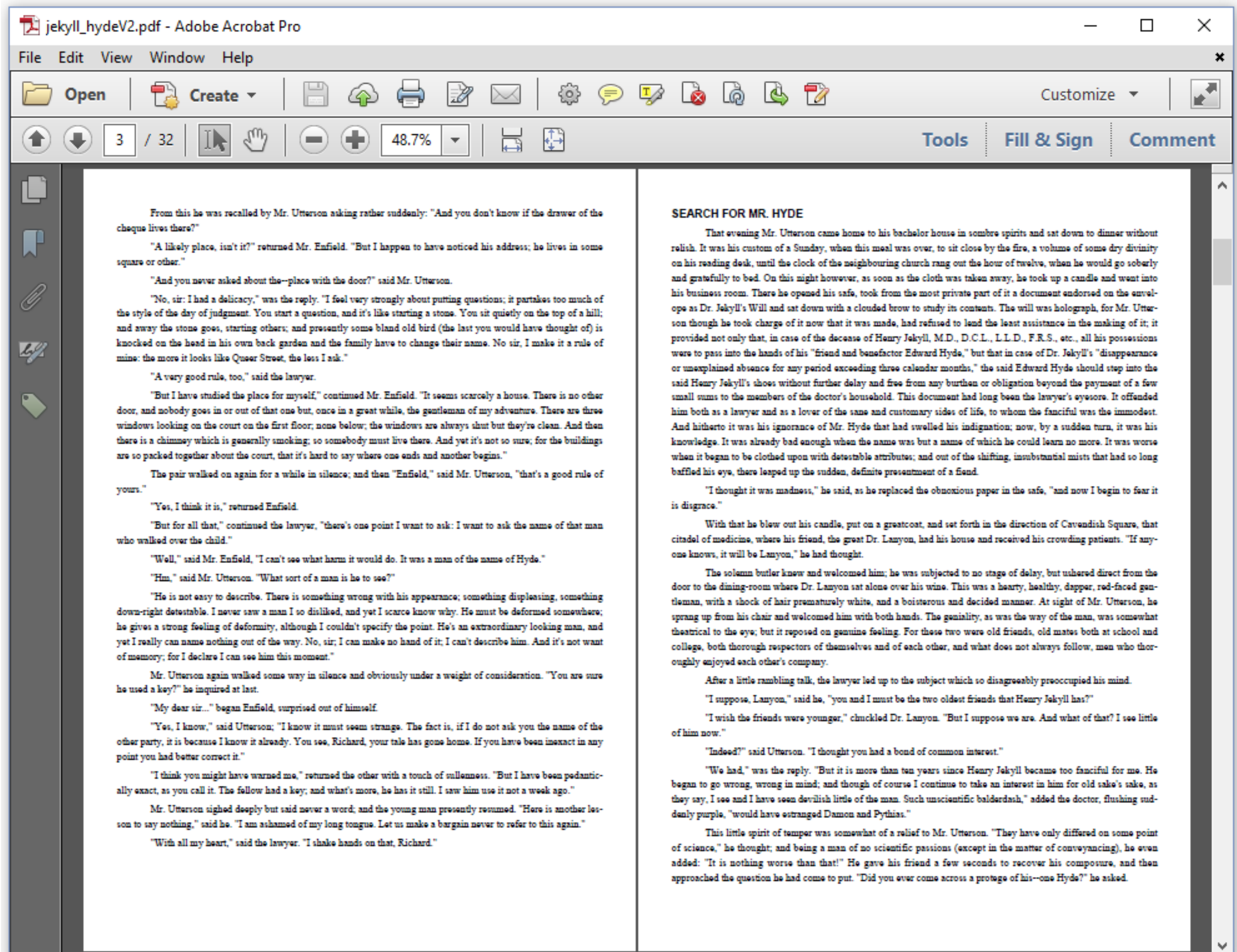
The `setKeepWithNext()` method was introduced in iText 7.0.1. You won't find it in the very first iText 7 release. We're investigating if we could support the method for nested objects. We're reluctant to do this because this could have a significant negative impact on the overall performance of the library.

The ParagraphAndDiv2[67] example shows how it's used.

[67]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1966-c04e05_paragraphanddiv2.java

```
BufferedReader br = new BufferedReader(new FileReader(SRC));
String line;
Div div = new Div();
while ((line = br.readLine()) != null) {
    document.add(new Paragraph(line)
        .setFont(bold).setFontSize(12)
        .setMarginBottom(0)
        .setKeepWithNext(true));
    div = new Div()
        .setFont(font).setFontSize(11)
        .setMarginBottom(18);
    while ((line = br.readLine()) != null) {
        div.add(
            new Paragraph(line)
                .setMarginBottom(0)
                .setFirstLineIndent(36)
        );
        if (line.isEmpty()) {
            document.add(div);
            break;
        }
    }
}
document.add(div);
```

We use a `Paragraph` added straight to the `Document` for the title (line 5); we create a `Div` to combine the rest of the content in the chapter (line 9). We indicate that the `Paragraph` needs to be kept on the same page as (the first part of) the `Div` by adding `setKeepWithNext(true)`. The result is shown in figure 4.5. The title “SEARCH FOR MR. HYDE” is now forwarded to the next page when compared to figure 4.4.




From this he was recalled by Mr. Utterson asking rather suddenly: "And you don't know if the drawer of the cheque lives there?"

"A likely place, isn't it?" returned Mr. Enfield. "But I happen to have noticed his address; he lives in some square or other."

"And you never asked about the--place with the door?" said Mr. Utterson.

"No, sir: I had a delicacy," was the reply. "I feel very strongly about putting questions; it partakes too much of the style of the day of judgment. You start a question, and it's like starting a stone. You sit quietly on the top of a hill; and away the stone goes, starting others; and presently some bland old bird (the last you would have thought of) is knocked on the head in his own back garden and the family have to change their name. No sir, I make it a rule of mine: the more it looks like Queer Street, the less I ask."

"A very good rule, too," said the lawyer.

"But I have studied the place for myself," continued Mr. Enfield. "It seems scarcely a house. There is no other door, and nobody goes in or out of that one but, once in a great while, the gentleman of my adventure. There are three windows looking on the court on the first floor; none below; the windows are always shut but they're clean. And then there is a chimney which is generally smoking; so somebody must live there. And yet it's not so sure; for the buildings are so packed together about the court, that it's hard to say where one ends and another begins."

The pair walked on again for a while in silence; and then "Enfield," said Mr. Utterson, "that's a good rule of yours."

"Yes, I think it is," returned Enfield.

"But for all that," continued the lawyer, "there's one point I want to ask: I want to ask the name of that man who walked over the child."

"Well," said Mr. Enfield, "I can't see what harm it would do. It was a man of the name of Hyde."

"Hm," said Mr. Utterson. "What sort of a man is he to see?"

"He is not easy to describe. There is something wrong with his appearance; something displeasing, something down-right detestable. I never saw a man I so disliked, and yet I scarce know why. He must be deformed somewhere; he gives a strong feeling of deformity, although I couldn't specify the point. He's an extraordinary looking man, and yet I really can name nothing out of the way. No, sir; I can make no hand of it; I can't describe him. And it's not want of memory; for I declare I can see him this moment."

Mr. Utterson again walked some way in silence and obviously under a weight of consideration. "You are sure he used a key?" he inquired at last.

"My dear sir..." began Enfield, surprised out of himself.

"Yes, I know," said Utterson; "I know it must seem strange. The fact is, if I do not ask you the name of the other party, it is because I know it already. You see, Richard, your tale has gone home. If you have been inexact in any point you had better correct it."

"I think you might have warned me," returned the other with a touch of sullenness. "But I have been pedantic-ally exact, as you call it. The fellow had a key; and what's more, he has it still. I saw him use it not a week ago."

Mr. Utterson sighed deeply but said never a word; and the young man presently resumed. "Here is another les-son to say nothing," said he. "I am ashamed of my long tongue. Let us make a bargain never to refer to this again."

"With all my heart," said the lawyer. "I shake hands on that, Richard."

SEARCH FOR MR. HYDE

That evening Mr. Utterson came home to his bachelor house in sombre spirits and sat down to dinner without relish. It was his custom of a Sunday, when this meal was over, to sit close by the fire, a volume of some dry divinity on his reading desk, until the clock of the neighbouring church rang out the hour of twelve, when he would go soberly and gratefully to bed. On this night however, as soon as the cloth was taken away, he took up a candle and went into his business room. There he opened his safe, took from the most private part of it a document endorsed on the envel-ope as Dr. Jekyll's Will and sat down with a clouded brow to study its contents. The will was holograph, for Mr. Utter-son though he took charge of it now that it was made, had refused to lend the least assistance in the making of it; it provided not only that, in case of the decease of Henry Jekyll, M.D., D.C.L., L.L.D., F.R.S., etc., all his possessions were to pass into the hands of his "friend and benefactor Edward Hyde," but that in case of Dr. Jekyll's "disappearance or unexplained absence for any period exceeding three calendar months," the said Edward Hyde should step into the said Henry Jekyll's shoes without further delay and free from any burthen or obligation beyond the payment of a few small sums to the members of the doctor's household. This document had long been the lawyer's eyesore. It offended him both as a lawyer and as a lover of the sane and customary sides of life, to whom the fanciful was the immodest. And hitherto it was his ignorance of Mr. Hyde that had swelled his indignation; now, by a sudden turn, it was his knowledge. It was already bad enough when the name was but a name of which he could learn no more. It was worse when it began to be clothed upon with detestable attributes; and out of the shifting, insubstantial mists that had so long baffled his eye, there leaped up the sudden, definite presentment of a fiend.

"I thought it was madness," he said, as he replaced the obnoxious paper in the safe, "and now I begin to fear it is disgrace."

With that he blew out his candle, put on a greatcoat, and set forth in the direction of Cavendish Square, that citadel of medicine, where his friend, the great Dr. Lanyon, had his house and received his crowding patients. "If any-one knows, it will be Lanyon," he had thought.

The solemn butler knew and welcomed him; he was subjected to no stage of delay, but ushered direct from the door to the dining-room where Dr. Lanyon sat alone over his wine. This was a hearty, healthy, dapper, red-faced gen-tleman, with a shock of hair prematurely white, and a boisterous and decided manner. At sight of Mr. Utterson, he sprang up from his chair and welcomed him with both hands. The geniality, as was the way of the man, was somewhat theatrical to the eye; but it reposed on genuine feeling. For these two were old friends, old mates both at school and college, both thorough respectors of themselves and of each other, and what does not always follow, men who thor-oughly enjoyed each other's company.

After a little rambling talk, the lawyer led up to the subject which so disagreeably preoccupied his mind.

"I suppose, Lanyon," said he, "you and I must be the two oldest friends that Henry Jekyll has?"

"I wish the friends were younger," chuckled Dr. Lanyon. "But I suppose we are. And what of that? I see little of him now."

"Indeed?" said Utterson. "I thought you had a bond of common interest."

"We had," was the reply. "But it is more than ten years since Henry Jekyll became too fanciful for me. He began to go wrong, wrong in mind; and though of course I continue to take an interest in him for old sake's sake, as they say, I see and I have seen devilish little of the man. Such unscientific balderdash," added the doctor, flushing sud-denly purple, "would have estranged Damon and Pythias."

This little spirit of temper was somewhat of a relief to Mr. Utterson. "They have only differed on some point of science," he thought; and being a man of no scientific passions (except in the matter of conveyancing), he even added: "It is nothing worse than that!" He gave his friend a few seconds to recover his composure, and then approached the question he had come to put. "Did you ever come across a protege of his--one Hyde?" he asked.

**Figure 4.5: keeping the title together with the text**

The `setKeepWithNext()` method can be used with all other `AbstractElement` implementations, except `Cell`. The method only works for elements added straight to the `Document` instance. It doesn't work for nested objects such as a `Cell` that is always added to a `Table` and never straight to a `Document`. In the case of our example, it wouldn't work if the title `Paragraph` was added to the `Div` instead of to the `Document`.

# Changing the leading of a Paragraph

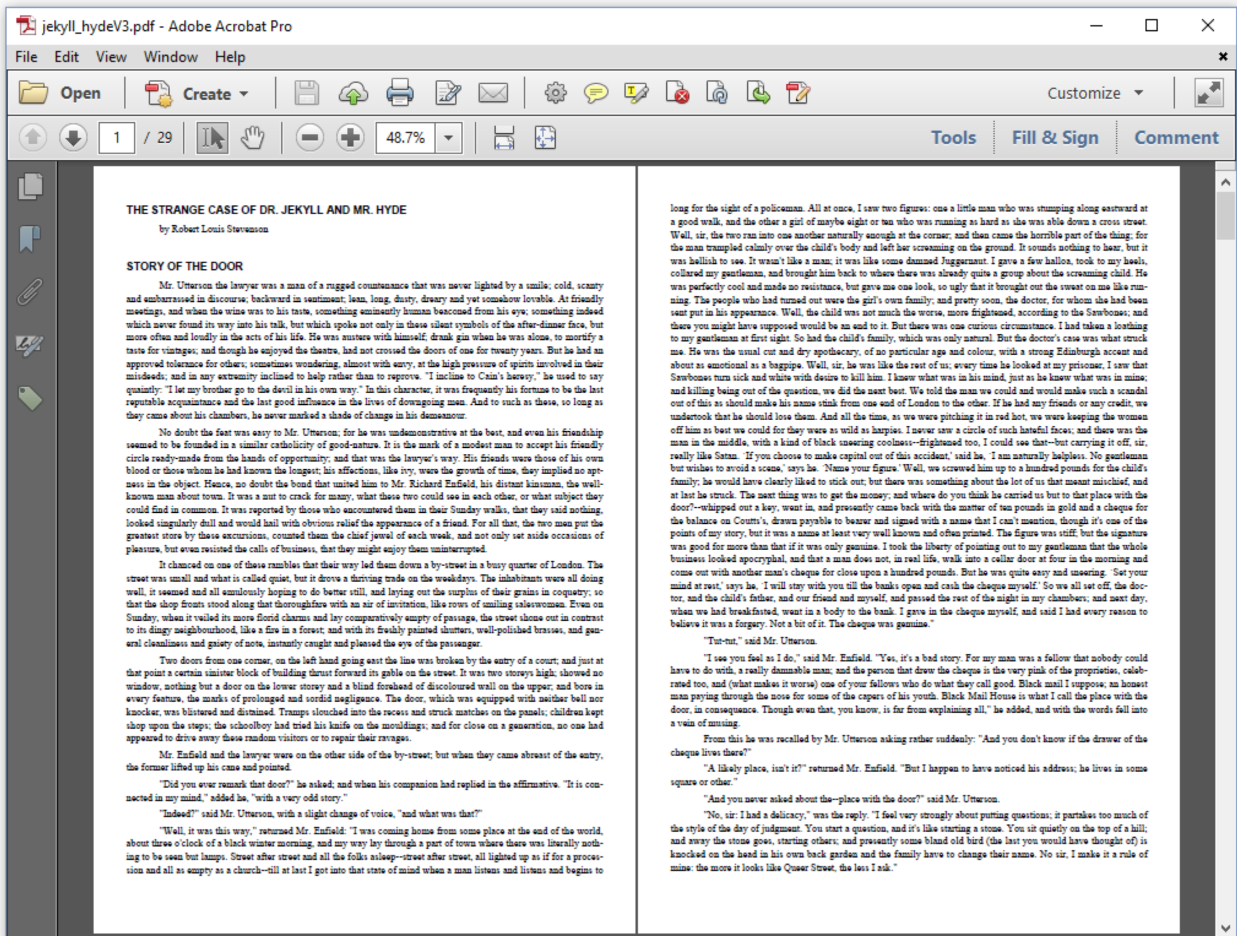
The `Paragraph` class has some extra methods on top of the methods defined at the `AbstractElement` level. We've already used the methods involving `TabStops` in the previous chapter. We also introduced the `setFirstLineIndent()` method on the sly. Now we are going to look at a method to change the leading.

The word *leading* is pronounced as *ledding*, and it's derived from the word *lead* (the metal). When type was set by hand for printing presses, strips of lead were placed between lines of type to add space. The word originally referred to the thickness of these strips of lead that were placed between the lines. The PDF standard redefines the leading as "the vertical distanced between the baselines of adjacent lines of text" (ISO-32000-1, section 9.3.5).

There are two ways to change the leading of a `Paragraph`:

- `setFixedLeading()`— changes the leading to an absolute value. For instance: if you define a fixed leading of 18, the distance between the baseline of two lines of text will be 18 user units.
- `setMultipliedLeading`—changes the leading to a value relative to the font size. For instance, if you define a multiplied leading of 1.5f and the font is 12 pt, then the leading will be 18 user units (which is 1.5 times 12).

These methods are mutually exclusive. If you use both methods on the same `Paragraph`, the last method that was invoked will prevail. Figure 4.6 shows yet another conversion of the story to PDF. The total number of pages is lower because we changed the distance between the lines by adding `.setMultipliedLeading(1.2f)`.




THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and laying out the surplus of their grains in coquetry; so that the shop fronts stood along that thoroughfare with an air of invitation, like rows of smiling saleswomen. Even on Sunday, when it veiled its more florid charms and lay comparatively empty of passage, the street shone out in contrast to its dingy neighbourhood, like a fire in a forest; and with its freshly painted shutters, well-polished brasses, and general cleanliness and gaiety of note, instantly caught and pleased the eye of the passenger.

Two doors from one corner, on the left hand going east the line was broken by the entry of a court; and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept shop upon the steps; the schoolboy had tried his knife on the mouldings; and for close on a generation, no one had appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the rest of us; every time he looked at my prisoner, I saw that Sawbones turn sick and white with desire to kill him. I knew what was in his mind, just as he knew what was in mine; and killing being out of the question, we did the next best. We told the man we could and would make such a scandal out of this as should make his name stink from one end of London to the other. If he had any friends or any credit, we undertook that he should lose them. And all the time, as we were pitching it in red hot, we were keeping the women off him as best we could for they were as wild as harpies. I never saw a circle of such hateful faces; and there was the man in the middle, with a kind of black sneering coolness--frightened too, I could see that--but carrying it off, sir, really like Satan. 'If you choose to make capital out of this accident,' said he, 'I am naturally helpless. No gentleman but wishes to avoid a scene,' says he. 'Name your figure.' Well, we screwed him up to a hundred pounds for the child's family; he would have clearly liked to stick out; but there was something about the lot of us that meant mischief, and at last he struck. The next thing was to get the money; and where do you think he carried us but to that place with the door?--whipped out a key, went in, and presently came back with the matter of ten pounds in gold and a cheque for the balance on Coutts's, drawn payable to bearer and signed with a name that I can't mention, though it's one of the points of my story, but it was a name at least very well known and often printed. The figure was stiff; but the signature was good for more than that if it was only genuine. I took the liberty of pointing out to my gentleman that the whole business looked apocryphal, and that a man does not, in real life, walk into a cellar door at four in the morning and come out with another man's cheque for close upon a hundred pounds. But he was quite easy and sneering. 'Set your mind at rest,' says he, 'I will stay with you till the banks open and cash the cheque myself.' So we all set off, the doctor, and the child's father, and our friend and myself, and passed the rest of the night in my chambers; and next day, when we had breakfasted, went in a body to the bank. I gave in the cheque myself, and said I had every reason to believe it was a forgery. Not a bit of it. The cheque was genuine."

"Tut-tut," said Mr. Utterson.

"I see you feel as I do," said Mr. Enfield. "Yes, it's a bad story. For my man was a fellow that nobody could have to do with, a really damnable man; and the person that drew the cheque is the very pink of the proprieties, celebrated too, and (what makes it worse) one of your fellows who do what they call good. Black mail I suppose; an honest man paying through the nose for some of the capers of his youth. Black Mail House is what I call the place with the door, in consequence. Though even that, you know, is far from explaining all," he added, and with the words fell into a vein of musing.

From this he was recalled by Mr. Utterson asking rather suddenly: "And you don't know if the drawer of the cheque lives there?"

"A likely place, isn't it?" returned Mr. Enfield. "But I happen to have noticed his address; he lives in some square or other."

"And you never asked about the--place with the door?" said Mr. Utterson.

"No, sir: I had a delicacy," was the reply. "I feel very strongly about putting questions; it partakes too much of the style of the day of judgment. You start a question, and it's like starting a stone. You sit quietly on the top of a hill; and away the stone goes, starting others; and presently some bland old bird (the last you would have thought of) is knocked on the head in his own back garden and the family have to change their name. No sir, I make it a rule of mine: the more it looks like Queer Street, the less I ask."

**Figure 4.6: changing indentation and leading**

The code of the ParagraphAndDiv3[68] example is identical to what we had in the previous example, except for the following snippet.

```
1   div.add(
2       new Paragraph(line)
3           .setMarginBottom(0)
4           .setFirstLineIndent(36)
5           .setMultipliedLeading(1.2f)
6   );
```

When we add an object to a `Document` either directly or indirectly (e.g. through a `Div`), iText uses the appropriate `IRenderer` to render this object to PDF. In the "Before we start" section of this book, figure 0.4 shows an overview of the different renderers. Normal use of iText hardly ever

[68] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1967-c04e05_paragraphanddiv3.java

requires creating a custom renderer, but we'll take a look at one example in which we create a `MyParagraphRenderer` extending the default `ParagraphRenderer`.

# Creating a custom renderer

When we look at figure 4.7, we see two `Paragraphs` with a different background. For the first `Paragraph`, we used the `.setBackgroundColor()` method. This method draws a rectangle based on the position of the `Paragraph`. For the second `Paragraph`, we wanted a rectangle with rounded corners. As iText 7 doesn't have a method to achieve this, we wrote a custom `ParagraphRenderer` class.

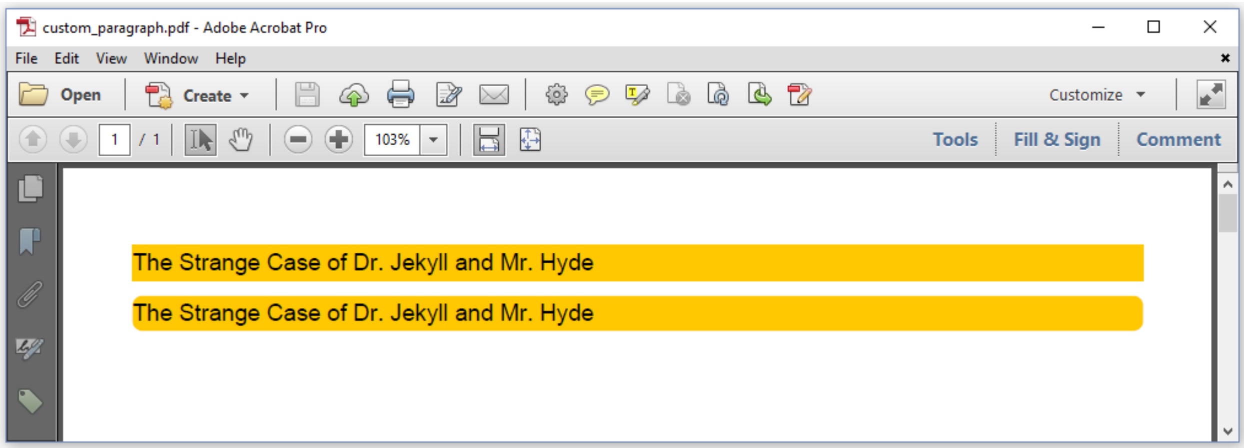



**Figure 4.7: default and custom background for a Paragraph**

Let's take a look at the CustomParagraph[69] example to see the difference between the two approaches. The first `Paragraph` was added like this:

```
1 Paragraph p1 = new Paragraph(
2     "The Strange Case of Dr. Jekyll and Mr. Hyde");
3 p1.setBackgroundColor(Color.ORANGE);
4 document.add(p1);
```

The second `Paragraph` was added like this:

[69] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1968-c04e06_custompargraph.java

```java
Paragraph p2 = new Paragraph(
    "The Strange Case of Dr. Jekyll and Mr. Hyde");
p2.setBackgroundColor(Color.ORANGE);
p2.setNextRenderer(new MyParagraphRenderer(p2));
document.add(p2);
```

This second approach requires an extra class:

```java
class MyParagraphRenderer extends ParagraphRenderer {
    public MyParagraphRenderer(Paragraph modelElement) {
        super(modelElement);
    }
    @Override
    public void drawBackground(DrawContext drawContext) {
        Background background =
            this.<Background>getProperty(Property.BACKGROUND);
        if (background != null) {
            Rectangle bBox = getOccupiedAreaBBox();
            boolean isTagged =
                drawContext.isTaggingEnabled()
                && getModelElement() instanceof IAccessibleElement;
            if (isTagged) {
                drawContext.getCanvas().openTag(new CanvasArtifact());
            }
            Rectangle bgArea = applyMargins(bBox, false);
            if (bgArea.getWidth() <= 0 || bgArea.getHeight() <= 0) {
                return;
            }
            drawContext.getCanvas().saveState()
                .setFillColor(background.getColor())
                .roundRectangle(
                (double)bgArea.getX() - background.getExtraLeft(),
                (double)bgArea.getY() - background.getExtraBottom(),
                (double)bgArea.getWidth()
                    + background.getExtraLeft() + background.getExtraRight(),
                (double)bgArea.getHeight()
                    + background.getExtraTop() + background.getExtraBottom(),
                5)
                .fill().restoreState();
            if (isTagged) {
                drawContext.getCanvas().closeTag();
            }
```

```
35            }
36        }
37    }
```

We extend the existing `ParagraphRenderer` class and we override one single method. We take the original `drawBackground()` method from the `AbstractRenderer` class, and we replace the `rectangle()` method with the `roundRectangle()` method (line 23). As you can see in line 24-29. the dimension of the rectangle can be fine-tuned with extra space to the left, right, top, and bottom. These values can be passed to the internal `Background` object by using a different flavor of the `setBackgroundColor()` method that takes 4 extra `float` values (`extraLeft`, `extraTop`, `extraRight`, and `extraBottom`).

We'll conclude this chapter with some examples involving the `List` and `ListItem` class.

# Lists and list symbols

Figure 4.8 shows the different types of lists that are available by default. We recognized numbered lists (roman and arabic numbers), lists with letters of the alphabet (lowercase, uppercase, Latin, Greek), and so on.

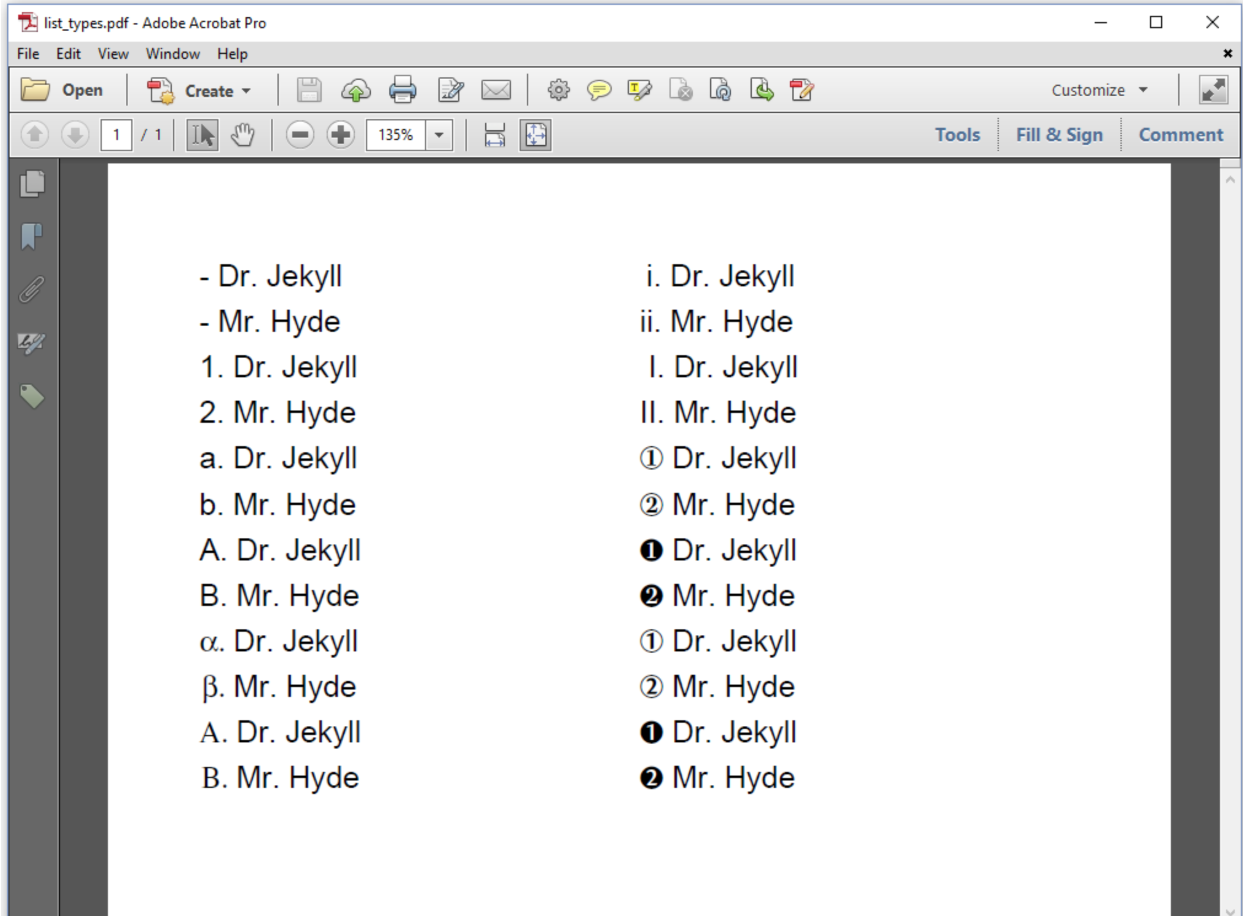



**Figure 4.8: different types of lists**

The ListTypes[70] example shows how the first three lists are added.

```
1  List list = new List();
2  list.add("Dr. Jekyll");
3  list.add("Mr. Hyde");
4  document.add(list);
5  list = new List(ListNumberingType.DECIMAL);
6  list.add("Dr. Jekyll");
7  list.add("Mr. Hyde");
8  document.add(list);
9  list = new List(ListNumberingType.ENGLISH_LOWER);
10 list.add("Dr. Jekyll");
11 list.add("Mr. Hyde");
12 document.add(list);
```

[70] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1969-c04e07_listtypes.java

In line 1, we create a list without specifying a type. By default, this will result in a list with hyphens as list symbols. We add two list items the *quick and dirty* way in line 2-3; then we add the `list` to the `Document` in line 4. We repeat these four lines many times, first we create a decimal list (line 5), then we define an alphabetic list with lowercase letters (line 9).

The parameters we use to create different types of lists are stored in an `enum`. This `ListNumberingType` enumeration consists of the following values:

- `DECIMAL`– the list symbols are arabic numbers: 1, 2, 3, 4, 5,...
- `ROMAN_LOWER`– the list symbols are lowercase roman numbers: i, ii, iii, iv, v,...
- `ROMAN_UPPER`– the list symbols are uppercase roman numbers: I, II, III, IV, V,...
- `ENGLISH_LOWER`– the list symbols are lowercase alphabetic letters (using the English alphabet): a, b, c, d, e,...
- `ENGLISH_UPPER`– the list symbols are uppercase alphabetic letters (using the English alphabet): A, B, C, D, E,...
- `GREEK_LOWER`– the list symbols are lowercase Greek letters: α, β, γ, δ, ε,...
- `GREEK_UPPER`– the list symbols are uppercase Greek letters: Α, Β, Γ, Δ, Ε,...
- `ZAPF_DINGBATS_1`– the list symbols are bullets from the Zapfdingbats font, more specifically characters in the range [172; 181].
- `ZAPF_DINGBATS_2`– the list symbols are bullets from the Zapfdingbats font, more specifically characters in the range [182; 191].
- `ZAPF_DINGBATS_3`– the list symbols are bullets from the Zapfdingbats font, more specifically characters in the range [192; 201].
- `ZAPF_DINGBATS_4`– the list symbols are bullets from the Zapfdingbats font, more specifically characters in the range [202; 221].

Obviously, we can also define our own custom list symbols, or we can use a combination of the default list symbols (e.g. numbers) and combine them with a prefix or a suffix. That's demonstrated in figure 4.9.

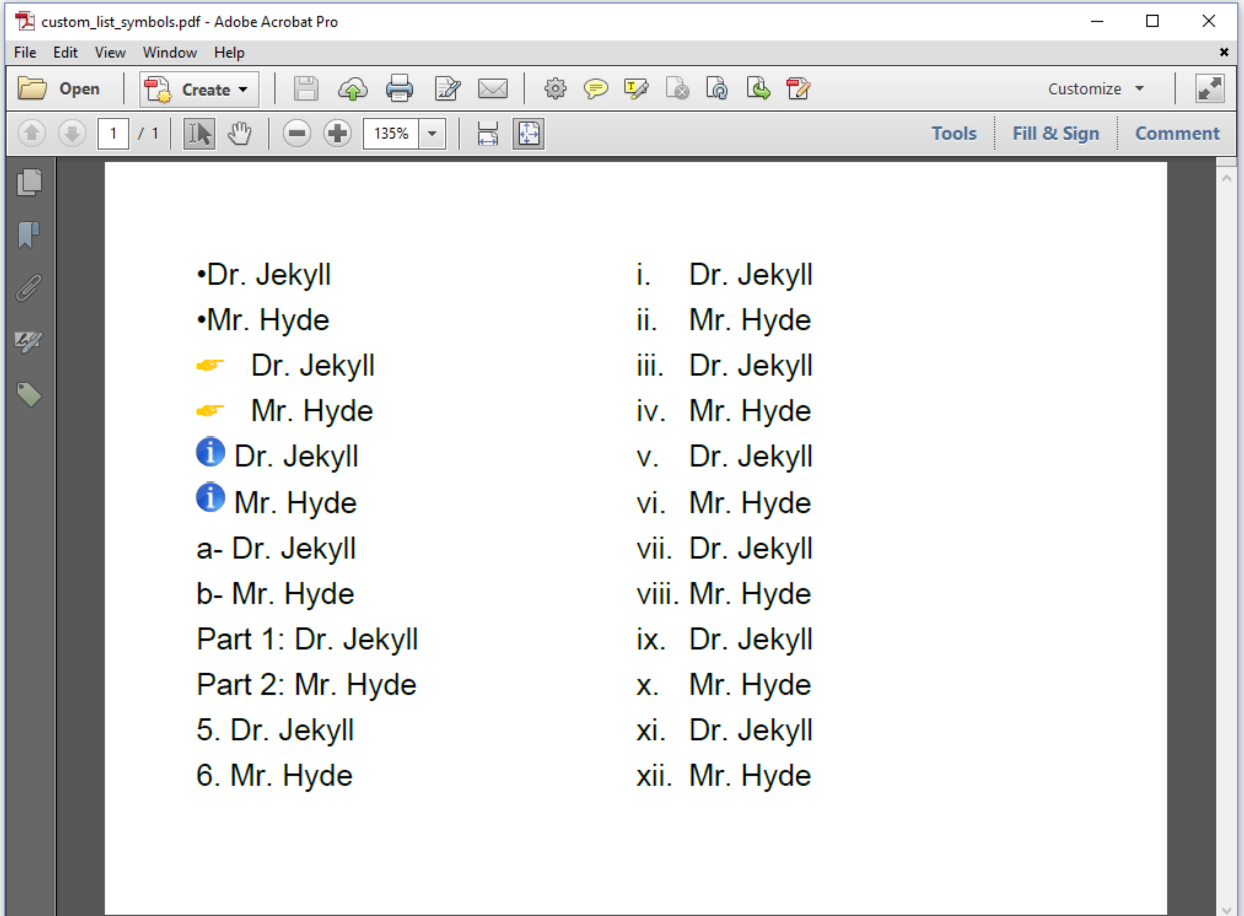



**Figure 4.9: custom list symbols**

The PDF in the screen shot of figure 4.9 was the result of the CustomListSymbols[71] example. We'll examine this example snippet by snippet.

First we take a look at how we can introduce a simple bullet as list symbol, instead of the default hyphen.

```
1 List list = new List();
2 list.setListSymbol("\u2022");
3 list.add("Dr. Jekyll");
4 list.add("Mr. Hyde");
5 document.add(list);
```

We create a `List` and we use the `setListSymbol()` method to change the list symbol. We can use any `String` as list symbol. In our case, we want a single bullet. The Unicode value of the bullet character

[71] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1970-c04e08_customlistsymbols.java

is /u2022. If you examine the screen shot, you notice that the bullet is rather close to the content of the list items. We can change this by defining an indentation using the `setSymbolIndent()` method as is done in the next code snippet.

```
1 list = new List();
2 PdfFont font = PdfFontFactory.createFont(FontConstants.ZAPFDINGBATS);
3 list.setListSymbol(new Text("*").setFont(font).setFontColor(Color.ORANGE));
4 list.setSymbolIndent(10);
5 list.add("Dr. Jekyll");
6 list.add("Mr. Hyde");
7 document.add(list);
```

Here we set the list symbol to *, but we use a `Text` object instead of a `String`. and we set the font to ZapfDingbats. We also change the font color to orange. This results in a list symbol that looks as an orange pointing finger. In the next snippet, we use an `Image` object as a list symbol.

```
1 Image info = new Image(ImageDataFactory.create(INFO));
2 info.scaleAbsolute(12, 12);
3 list = new List().setSymbolIndent(3);
4 list.setListSymbol(info);
5 list.add("Dr. Jekyll");
6 list.add("Mr. Hyde");
7 document.add(list);
```

In line 1. we create an `Image` object; `INFO` contains the path to a blue info bullet. We scale the image so that it measures 12 by 12 user units, and we pass the `Image` as a parameter of the `setListSymbol()` method.

In the default list types, iText always added a dot after the list symbol of numbered lists: a., b., c., and so on. Maybe we don't want this dot. Maybe we want the list symbols to look like this: a-, b-, c-, and so on. The following code snippet shows how to achieve this.

```
1 list = new List();
2 list.setListSymbol(ListNumberingType.ENGLISH_LOWER);
3 list.setPostSymbolText("- ");
4 list.add("Dr. Jekyll");
5 list.add("Mr. Hyde");
6 document.add(list);
```

Line 1 and 2 are the equivalent of `list = new List(ListNumberingType.ENGLISH_LOWER);` It results in a numbered list using the English alphabet. We use the `setPostSymbolText()` method to replace the dot that is automatically added after each letter with `"- "`.

There's also a `setPreSymbolText()` method to add text in front of the default list symbol. The following code snippet creates a decimal list (`1.`, `2.`, `3.`,...), but by adding a pre- and a post-symbol, the list symbols have become list labels that look like this: `Part 1: `, `Part 2: `, `Part 3: `, and so on.

```
list = new List(ListNumberingType.DECIMAL);
list.setPreSymbolText("Part ");
list.setPostSymbolText(": ");
list.add("Dr. Jekyll");
list.add("Mr. Hyde");
document.add(list);
```

Not every numbered list needs to start with 1, i, a, and so on. You can also choose to start with a higher number (or letter) using the `setItemStartIndex()` method. In the following code sample, we start counting at 5.

```
list = new List(ListNumberingType.DECIMAL);
list.setItemStartIndex(5);
list.add("Dr. Jekyll");
list.add("Mr. Hyde");
document.add(list);
```

Finally, we'll use the `setListSymbolAlignment()` to change the alignment of the labels. If you compare the lowercase Roman numbers list in figure 4.8 with the one in figure 4.9, you'll see a difference in the way the list labels are aligned.

```
list = new List(ListNumberingType.ROMAN_LOWER);
list.setListSymbolAlignment(ListSymbolAlignment.LEFT);
for (int i = 0; i < 6; i++) {
    list.add("Dr. Jekyll");
    list.add("Mr. Hyde");
}
document.add(list);
```

So far, we've always added list items to a list using `String`s. These `String` values are changed into `ListItem`s internally.

# Adding ListItem objects to a List

Looking at the class diagram in the "Before we start" section of this book, we notice that `ListItem` is a subclass of the `Div` class. We can add different objects to a `ListItem` just like we did with the `Div` object, but now we do so in the context of a list.

Let's do the test and adapt one of the first examples of this chapter to use `ListItem`s instead of `Div`s. Figure 4.10 shows the result.

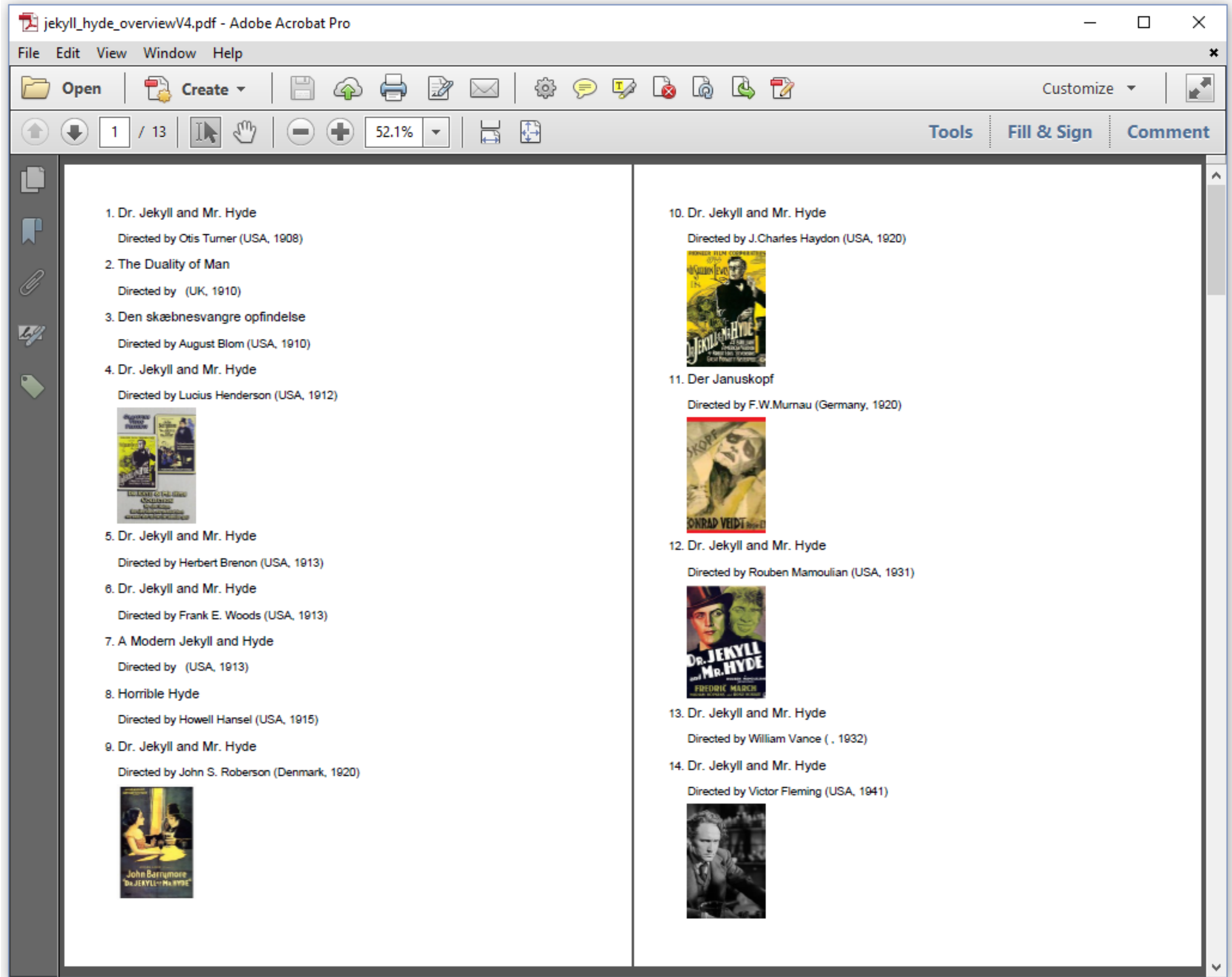



**Figure 4.10: List items**

The code of the ListItemExample[72] example is very similar to the code of the `Div` examples.

```
1  public void createPdf(String dest) throws IOException {
2      PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
3      Document document = new Document(pdf);
4      List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
5      resultSet.remove(0);
6      com.itextpdf.layout.element.List list =
7          new com.itextpdf.layout.element.List(ListNumberingType.DECIMAL);
8      for (List<String> record : resultSet) {
9          ListItem li = new ListItem();
10         li.setKeepTogether(true);
```

[72]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1971-c04e09_listitemexample.java

```
11          String url = String.format(
12              "http://www.imdb.com/title/tt%s", record.get(0));
13          Link movie = new Link(record.get(2), PdfAction.createURI(url));
14          li.add(new Paragraph(movie.setFontSize(14)))
15              .add(new Paragraph(String.format(
16                  "Directed by %s (%s, %s)",
17                  record.get(3), record.get(4), record.get(1))));
18          File file = new File(String.format(
19              "src/main/resources/img/%s.jpg", record.get(0)));
20          if (file.exists()) {
21              Image img = new Image(ImageDataFactory.create(file.getPath()));
22              img.scaleToFit(10000, 120);
23              li.add(img);
24          }
25          list.add(li);
26      }
27      document.add(list);
28      document.close();
29  }
```

As we already use a `java.util.List` (line 4), we need to fully qualify `com.itextpdf.layout.element.List` (line 6) to avoid ambiguity for our compiler. We use iText's `List` class to create a numbered list (line 7). We create a `ListItem` for every item in the `java.util.List` (line 9). We add `Paragraphs` and an `Image` (if present) to each `ListItem` (line 11-24). We add each `ListItem` to the `List` (line 25), and eventually we add the `List` to the `Document` (line 27).

# Nested lists

In the final example of this chapter, we'll create nested lists as shown in figure 4.11.

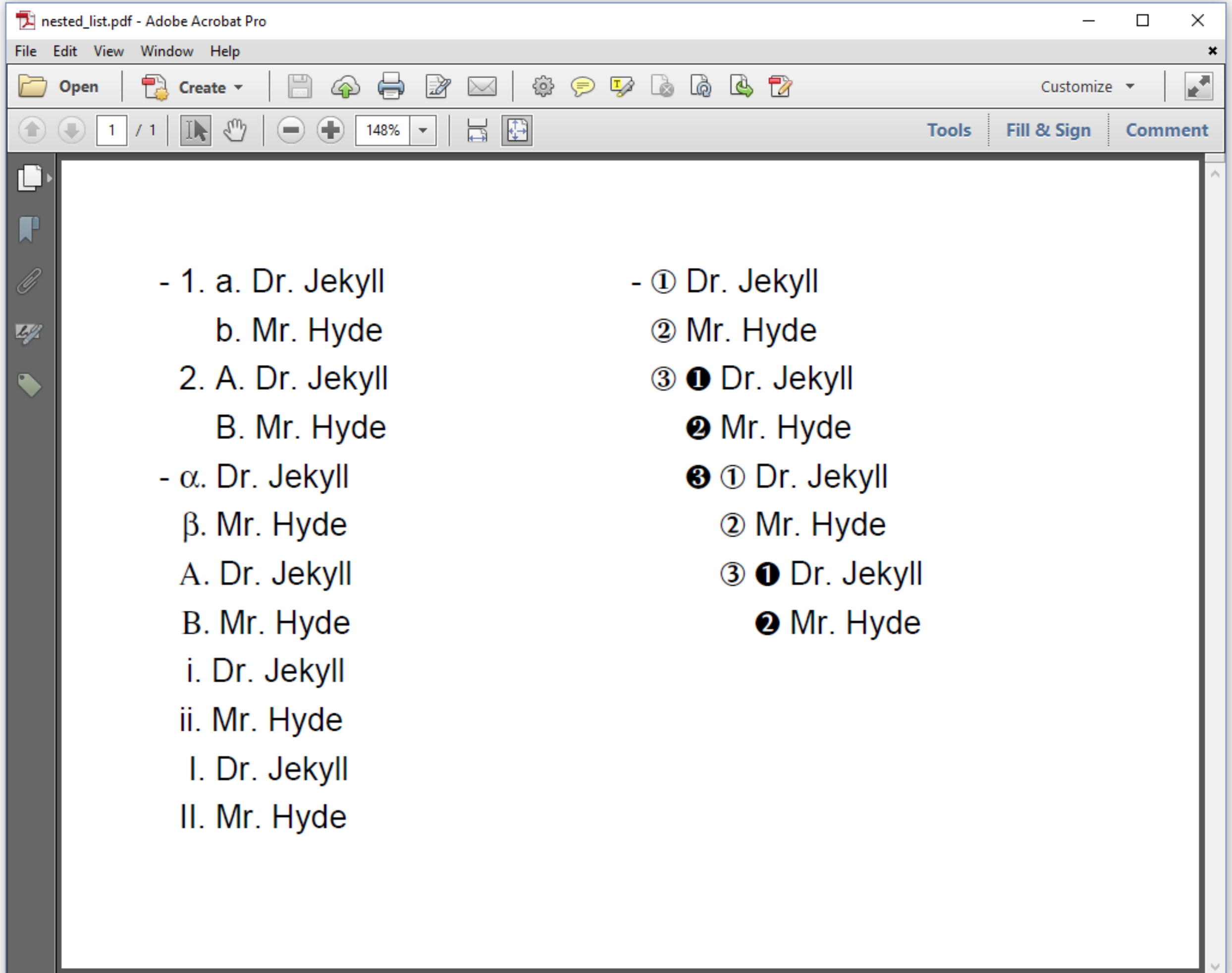



**Figure 4.11: nested lists**

The NestedLists[73] example is rather artificial, so please bear with me. We start with an ordinary list, named `list`. That's the list with the hyphens as list symbol.

```
1 List list = new List();
```

We create a numbered list `list1` (line 1). This list will have two `ListItems`, `liEL` (line 5) and `liEU` (line 11). We create a new `List` to be added to each of these list items respectively: `listEL` (line 2; lowercase English letters) and `listEU` (line 8, uppercase English letters). We add list items `"Dr. Jekyll"` and `"Mr. Hyde"` to each of these lists (line 3-4; line 9-10).

[73]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-4-examples-abstractelement-part-1#1972-c04e10_nestedlists.java

```
List list1 = new List(ListNumberingType.DECIMAL);
List listEL = new List(ListNumberingType.ENGLISH_LOWER);
listEL.add("Dr. Jekyll");
listEL.add("Mr. Hyde");
ListItem liEL = new ListItem();
liEL.add(listEL);
list1.add(liEL);
List listEU = new List(ListNumberingType.ENGLISH_UPPER);
listEU.add("Dr. Jekyll");
listEU.add("Mr. Hyde");
ListItem liEU = new ListItem();
liUL.add(listEU);
list1.add(liEU);
ListItem li1 = new ListItem();
li1.add(list1);
list.add(li1);
```

When we look at figure 4.11, we see the hyphen, we see a numbered list with list symbols `1.` and `2.`. Nested inside these lists are two lists using the English alphabet (lower- and uppercase).

In the next snippet, we create an extra `ListItem` for `list`, more specifically `li` (line 1). We add four lists to this `ListItem`: `listGL` (line 2), `listGU` (line 6), `listRL` (line 10), and `listRU` (line 14). These lists are added one after the other (Greek lowercase, Greek uppercase, Roman numbers lowercase, Roman number uppercase) to the list item with the default list symbol.

```
ListItem li = new ListItem();
List listGL = new List(ListNumberingType.GREEK_LOWER);
listGL.add("Dr. Jekyll");
listGL.add("Mr. Hyde");
li.add(listGL);
List listGU = new List(ListNumberingType.GREEK_UPPER);
listGU.add("Dr. Jekyll");
listGU.add("Mr. Hyde");
li.add(listGU);
List listRL = new List(ListNumberingType.ROMAN_LOWER);
listRL.add("Dr. Jekyll");
listRL.add("Mr. Hyde");
li.add(listRL);
List listRU = new List(ListNumberingType.ROMAN_UPPER);
listRU.add("Dr. Jekyll");
listRU.add("Mr. Hyde");
li.add(listRU);
list.add(li);
```

Furthermore, we create a list `listZ1` with numbered ZapfDingbats bullets. We add this list to a list item named `listZ1`.

```
1 List listZ1 = new List(ListNumberingType.ZAPF_DINGBATS_1);
2 listZ1.add("Dr. Jekyll");
3 listZ1.add("Mr. Hyde");
4 ListItem liZ1 = new ListItem();
5 liZ1.add(listZ1);
```

We create a second list `listZ2` with a different set of ZapfDingbats bullets. We add this list to a list item named `listZ2`.

```
1 List listZ2 = new List(ListNumberingType.ZAPF_DINGBATS_2);
2 listZ2.add("Dr. Jekyll");
3 listZ2.add("Mr. Hyde");
4 ListItem liZ2 = new ListItem();
5 liZ2.add(listZ2);
```

We create a second list `listZ3` with another set of ZapfDingbats bullets. We add this list to a list item named `listZ3`.

```
1 List listZ3 = new List(ListNumberingType.ZAPF_DINGBATS_3);
2 listZ3.add("Dr. Jekyll");
3 listZ3.add("Mr. Hyde");
4 ListItem liZ3 = new ListItem();
5 liZ3.add(listZ3);
```

We create a final list `listZ4` with yet another set of ZapfDingbats bullets. We add this list to a list item named `listZ4`.

```
1 List listZ4 = new List(ListNumberingType.ZAPF_DINGBATS_4);
2 listZ4.add("Dr. Jekyll");
3 listZ4.add("Mr. Hyde");
4 ListItem liZ4 = new ListItem();
5 liZ4.add(listZ4);
```

Now we nest these lists as follows:

- we add `liZ4` to `listZ3`, which was already added to `liZ3`,
- we add `liZ3` to `listZ2`, which was already added to `liZ2`,
- we add `liZ2` to `listZ1`, which was already added to `liZ1`.
- we add `liZ1` to `list`, which is the original list we created (the one with the hyphen as list symbol).

Finally, we add `list` to the `Document`.

```
1 listZ3.add(liZ4);
2 listZ2.add(liZ3);
3 listZ1.add(liZ2);
4 list.add(liZ1);
5 document.add(list);
```

The nested ZapfDingbats list is shown to the right in figure 4.11. As you can see, the different list items are indented exactly the way one would expect. This concludes the first series of `AbstractElement` examples.

# Summary

In this chapter, we discussed the building blocks `Div`, `LineSeparator`, `Paragraph`, `List`, and `ListItem`. We used `Div` to group other building blocks and `LineSeparator` to draw horizontal lines. We fixed a problem with the chapter 2 examples we weren't aware of: we learned how to keep specific elements together on one page. We didn't go into detail regarding the `IRenderer` implementations, but we looked at an example in which we changed the way a background is drawn for a `Paragraph`. We created a custom `ParagraphRenderer` to achieve this. Finally, we created a handful of `List` examples demonstrating different types of lists (numbered, unnumbered, straight-forward, nested, and so on).

The next chapter will be dedicated entirely to tables, more specifically to the `Table` and `Cell` class.

# Chapter 5: Adding AbstractElement objects (part 2)

Once we've finished this chapter, we'll have covered all of the basic building blocks available in iText 7. We've saved two of the most used building blocks for last: `Table` and `Cell`. These objects were designed to render content in a tabular form. Many developers use iText to convert the result set of a database query into a report in PDF. They create a `Table` of which every row corresponds with a database record, wrapping every field value in a `Cell` object.

We could easily create a similar table using our Jekyll and Hyde database to a PDF, but let's start with a handful of simple examples first.

## My first table

Figure 5.1 shows a simple table that was created with iText 7.

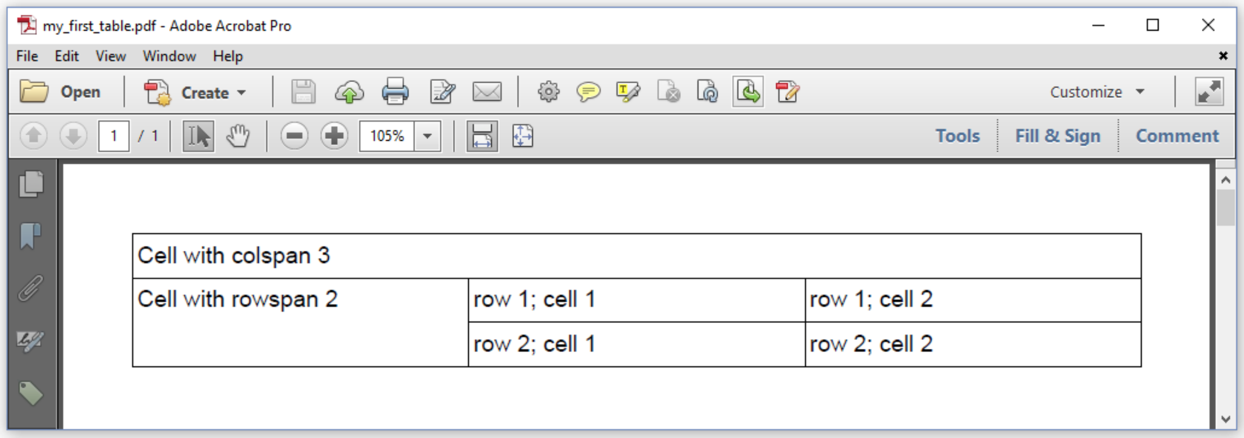



**Figure 5.1: my first table**

The code to create this table is really simple; see the MyFirstTable[74] example.

[74] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2031-c05e01_myfirsttable.java

```
1  public void createPdf(String dest) throws IOException {
2      PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
3      Document document = new Document(pdf);
4      Table table = new Table(3);
5      table.addCell(new Cell(1, 3).add("Cell with colspan 3"));
6      table.addCell(new Cell(2, 1).add("Cell with rowspan 2"));
7      table.addCell("row 1; cell 1");
8      table.addCell("row 1; cell 2");
9      table.addCell("row 2; cell 1");
10     table.addCell("row 2; cell 2");
11     document.add(table);
12     document.close();
13 }
```

We create a table with 3 columns in line 4. We add 6 cells in line 5-10:

- The first cell has a rowspan of 1 and a colspan of 3.
- The second cell has a rowspan of 2 and a colspan of 1.
- The following four cells have a rowspan and colspan of 1.

For the first two cells we explicitly created a `Cell` object because we wanted to define a specific rowspan or colspan. For the next four cells, we just added a `String` to the `Table`. A `Cell` object was created internally by iText. Line 7 is shorthand for `table.addCell(new Cell().add("row 1; cell 1"))`.

The `PdfPTable` and `PdfPCell` classes that we all know from iText 5 are no longer present. They were replaced by `Table` and `Cell`, and we simplified the way tables are created. The iText 5 concept of *text mode* versus *composite mode* caused a lot of confusion among first-time iText users. Adding content to a `Cell` is now done using the `add()` method.

Figure 5.2 shows a variation of our first table. We changed the width of the table, its alignment, and the width of the columns.

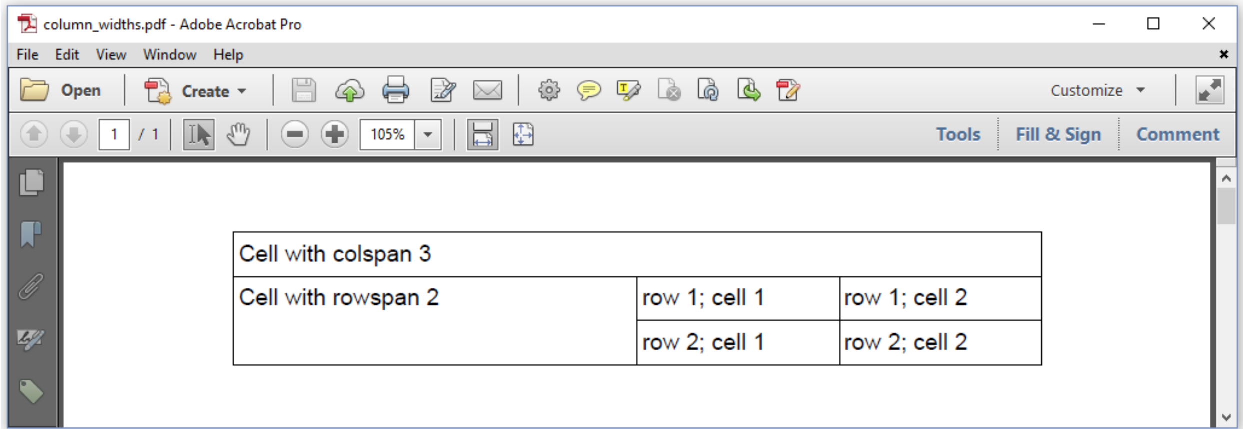



**Figure 5.2: defining column widths**

This was achieved by changing the constructor and by adding two extra lines; see the ColumnWidths[75] example.

```
1 Table table = new Table(new float[]{2, 1, 1});
2 table.setWidthPercent(80);
3 table.setHorizontalAlignment(HorizontalAlignment.CENTER);
```

Instead of passing the number of columns to the `Table` constructor, we now pass an array with as many elements as there are columns. Each element is a float value indicating the relative width of the corresponding column. In this case, the first column will be twice as wide as the second and third column.

We use the `setWidthPercent()` method so that the table takes 80% of the available width –that's the width of the page minus the width reserved for the left and right margin.

The default width percentage is 100%. There's also a `setWidth()` method that allows you to set the absolute width. Use this method if you prefer a value in user units over a width that is relative to the available width.

We use the `setHorizontalAlignment()` method to center the table.

# Table and cell Alignment

In figure 5.3, we also change the alignment of the content inside the cells.

[75] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2032-c05e02_columnwidths.java

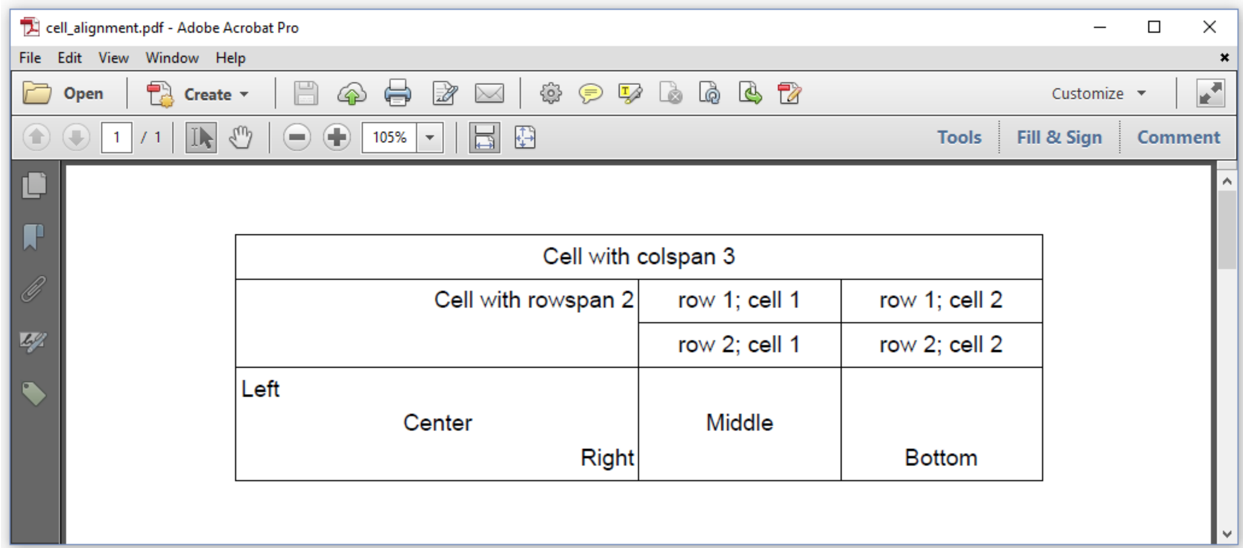



**Figure 5.3: alignment of cell content**

We can change the alignment of the content of a `Cell` in different ways. The CellAlignment[76] example demonstrates the different options.

```
1 Table table = new Table(new float[]{2, 1, 1});
2 table.setWidthPercent(80);
3 table.setHorizontalAlignment(HorizontalAlignment.CENTER);
4 table.setTextAlignment(TextAlignment.CENTER);
5 table.addCell(new Cell(1, 3).add("Cell with colspan 3"));
6 table.addCell(new Cell(2, 1).add("Cell with rowspan 2")
7     .setTextAlignment(TextAlignment.RIGHT));
8 table.addCell("row 1; cell 1");
9 table.addCell("row 1; cell 2");
10 table.addCell("row 2; cell 1");
11 table.addCell("row 2; cell 2");
12 Cell cell = new Cell()
13     .add(new Paragraph("Left").setTextAlignment(TextAlignment.LEFT))
14     .add(new Paragraph("Center"))
15     .add(new Paragraph("Right").setTextAlignment(TextAlignment.RIGHT));
16 table.addCell(cell);
17 cell = new Cell().add("Middle")
18     .setVerticalAlignment(VerticalAlignment.MIDDLE);
19 table.addCell(cell);
20 cell = new Cell().add("Bottom")
21     .setVerticalAlignment(VerticalAlignment.BOTTOM);
```

[76] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2033-c05e03_cellalignment.java

```
22 table.addCell(cell);
23 document.add(table);
```

Once more we use the `setHorizontalAlignment()` method to define the horizontal alignment of the table itself (line 3). Additionally, we use the `setTextAlignment()` method to change the default alignment of the content of the `Cell` added to this table. By default, this content is aligned to the left (`TextAlignment.LEFT`); we change the alignment to `TextAlignment.CENTER` (line 4). As a result, `"Cell with colspan 3"` will be centered in the first cell we add (line 5).

We change the alignment of `"Cell with rowspan 2"` to `TextAlignment.RIGHT` for the second cell. This time, we use the `setTextAlignment()` method at the level of the `Cell` (line 6-7). We complete the two rows in this rowspan by adding four more cells without specifying the alignment. The alignment is inherited from the table; their content is centered.

In line 12, we define a `Cell` for which we define the alignment at the level of the content.

- In line 13, we add a `Paragraph` that is aligned to the left.
- In line 14, we don't define an alignment for the `Paragraph`. The alignment is inherited from the `Cell`. No alignment was defined at the level of the `Cell` either, so the alignment is inherited from the `Table`. As a result, the content is centered.
- In line 15, we add a `Paragraph` that is aligned to the right.

The next two cell demonstrate the vertical alignment and the `setVerticalAlignment()` method. Content is aligned to the top by default (`VerticalAlignment.TOP`). In line 17-18, we create a `Cell` of which the alignment is set to the middle (vertically: `VerticalAlignment.MIDDLE`). In line 20-21, the content is bottom-aligned (`VerticalAlignment.BOTTOM`).

# Row and cell height

The height of a row will automatically adapt to the height of the cells in that row. The height of a cell will depend on its content, but we can always increase its height. Let's take a look at figure 5.4.

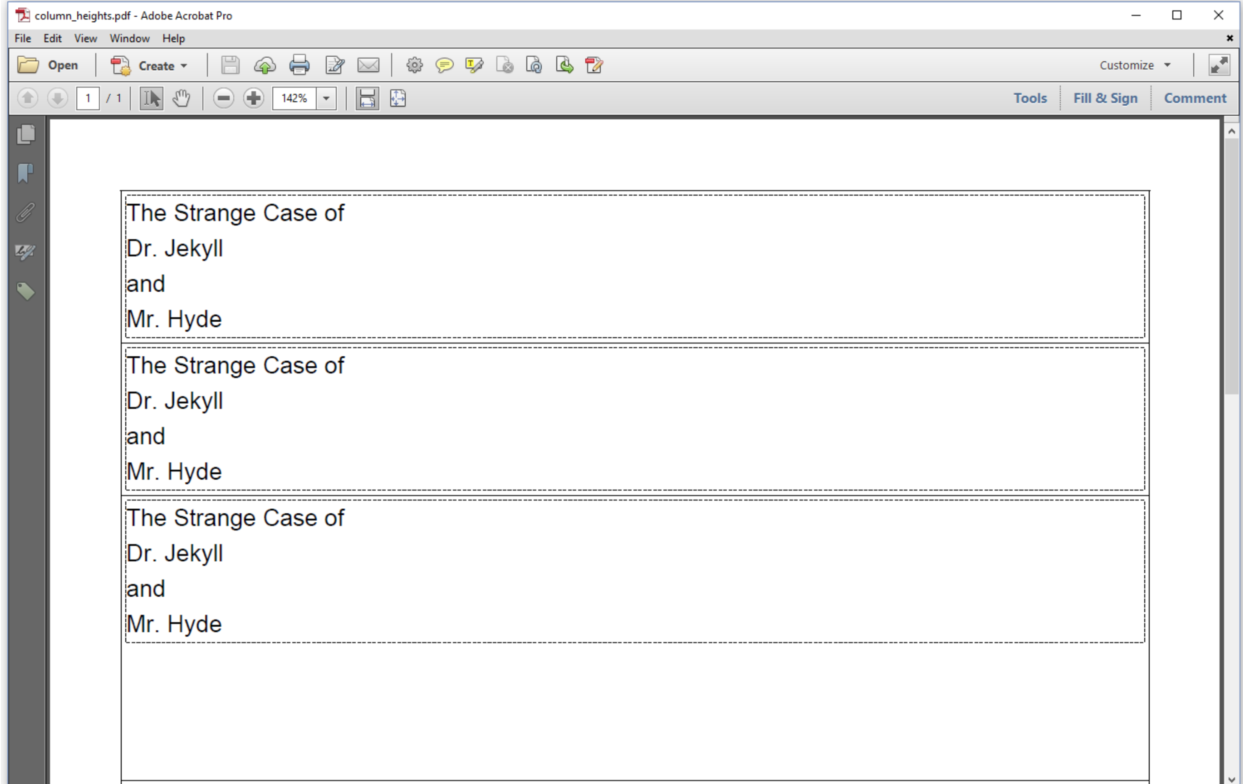



**Figure 5.4: changing the cell height**

In this table, we are adding the same `Paragraph` to a table with 1 column; see the ColumnHeights[77] example.

```
1 Paragraph p =
2     new Paragraph("The Strange Case of\nDr. Jekyll\nand\nMr. Hyde")
3         .setBorder(new DashedBorder(0.3f));
```

We define a border of 0.3 user units for the `Paragraph`, so that we can clearly make the distinction between the boundaries of the `Paragraph` and the borders of the `Cell`.

The first time, we add the `Paragraph` directly to the `Table`.

```
1 Table table = new Table(1);
2 table.addCell(p);
```

In this case, iText will determine the height in such a way that the content of the `Paragraph` fits the `Cell`.

In the second row, we change the height of the `Cell` in such a way that the content wouldn't fit.

[77] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2034-c05e04_columnheights.java

```
Cell cell = new Cell().setHeight(16).add(p);
table.addCell(cell);
```

If iText would reduce the cell height to 16 user units, content would be lost. Usually this isn't acceptable, so iText ignores the `setHeight()` method. Just like before, the height of the `Cell` is determined by its content.

For the third row, we define a height that is much higher than needed.

```
cell = new Cell().setHeight(144).add(p);
table.addCell(cell);
```

The dashed line shows the space needed for the `Paragraph`. The full line is the border of the `Cell`. When we look at the third row in figure 5.4, we see that there's quite some extra space between the bottom boundary of the `Paragraph` and the bottom border of the `Cell`.

We can also set a rotation angle for the `Cell`. This is done in figure 6.5. The full block of the `Paragraph` is rotated, and the height of the `Cell` adapts to the height that is necessary to render that rotated block completely.

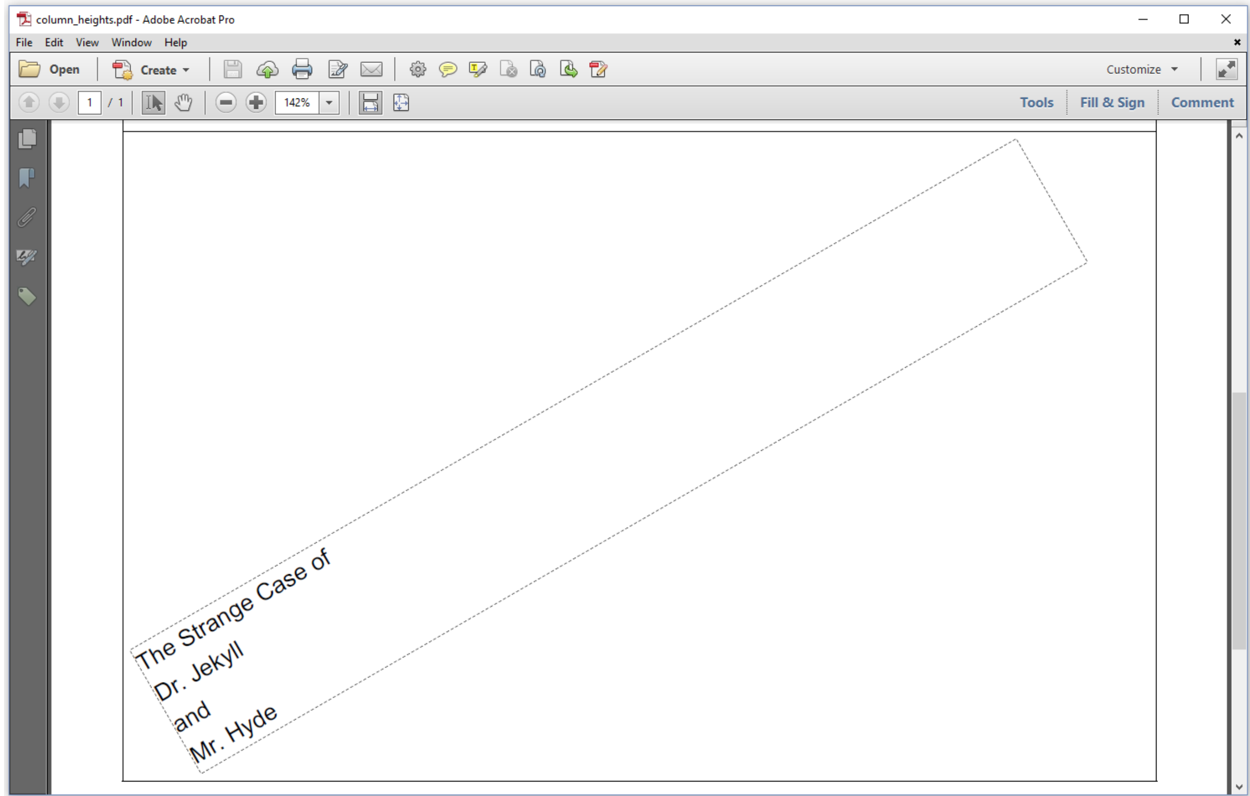



**Figure 5.5: rotating the content of a cell**

Rotating the content of a `Cell` is done using the `setRotationAngle()` method. The angle needs to be expressed in Radians.

```
cell = new Cell().add(p).setRotationAngle(Math.PI / 6);
table.addCell(cell);
```

The space between the dashed border of the `Paragraph` and the border of the `Cell` is called the padding. In the next example, we'll examine the difference between the margin and the padding.

# Cell margins and padding

In figure 5.6, we have set the background of the table to orange. We've also defined a background color for the different cells. This way, we can distinguish the difference between the margin of a cell and its padding.

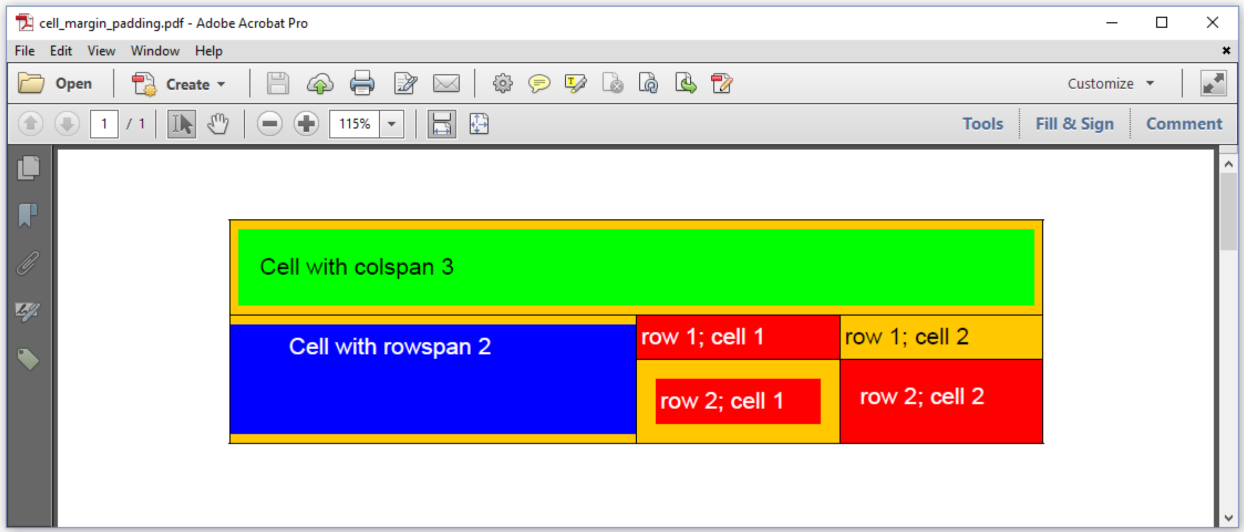



**Figure 5.6: the difference between the margin and the padding of a cell**

Let's take a look at the CellMarginPadding[78] example to see how this PDF was created.

```
Table table = new Table(new float[]{2, 1, 1});
table.setBackgroundColor(Color.ORANGE);
table.setWidthPercent(80);
table.setHorizontalAlignment(HorizontalAlignment.CENTER);
table.addCell(
    new Cell(1, 3).add("Cell with colspan 3")
        .setPadding(10).setMargin(5).setBackgroundColor(Color.GREEN));
table.addCell(new Cell(2, 1).add("Cell with rowspan 2")
    .setMarginTop(5).setMarginBottom(5).setPaddingLeft(30)
    .setFontColor(Color.WHITE).setBackgroundColor(Color.BLUE));
```

[78]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2035-c05e05_cellmarginpadding.java

```
11 table.addCell(new Cell().add("row 1; cell 1")
12     .setFontColor(Color.WHITE).setBackgroundColor(Color.RED));
13 table.addCell(new Cell().add("row 1; cell 2"));
14 table.addCell(new Cell().add("row 2; cell 1").setMargin(10)
15     .setFontColor(Color.WHITE).setBackgroundColor(Color.RED));
16 table.addCell(new Cell().add("row 2; cell 2").setPadding(10)
17     .setFontColor(Color.WHITE).setBackgroundColor(Color.RED));
18 document.add(table);
```

We set the background for the full table to orange in line 2. We add six cells to this table:

1. line 5-7: a cell with a green background, a margin of 5 user units and a padding of 10 user units. Looking at the screen shot, we see that the margin is the space between the border of the cell and the green rectangle –the background. The padding is the space between the border of that green rectangle and the content of the cell.
2. line 8-10: a cell with white text, a blue background, a top and bottom margin of 5 user units, and a left padding of 30 user units. We don't see any orange ribbons to the left and the right. We only see 5 user units of orange at the top and the bottom. The default margin of a `Cell` is 0 user units. The text doesn't start immediately at the left. There's 30 user units of space between the left border and the text.
3. line 11-12: a cell with white text, a red background, and default values for the margin and the padding. The text doesn't stick to the border because iText uses a default padding of 2 user units.
4. line 13: a cell with default properties. This cell has no background color. It's orange because of the background color of the table.
5. line 14-15: a cell with white text, a red background and a margin of 10 user units.
6. line 16-17: a cell with white text, a red background and a padding of 10 user units.

So far, we haven't defined the border of any of the cells. The default border is a `Border` instance define like this: `new SolidBorder(0.5f)`. There is something special about cell borders that requires more explanation.

# Table and cell borders

Figure 5.7 shows three tables with different borders. We'll discuss each of these tables one by one by examining the CellBorders[79] example..

[79] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2036-c05e06_cellborders.java

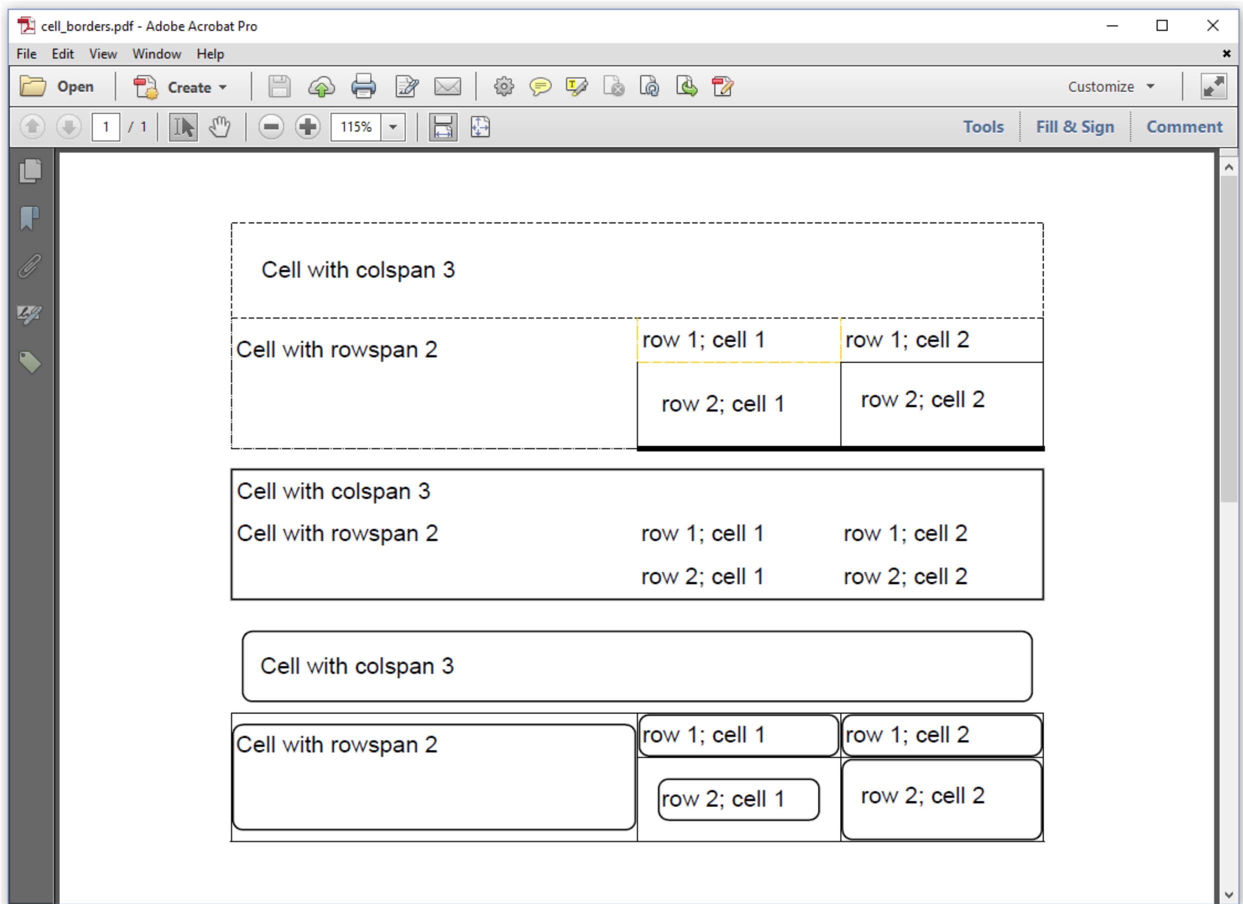



**Figure 5.7: changing table and cell borders**

The first table was created like this:

```
Table table1 = new Table(new float[]{2, 1, 1});
table1.setWidthPercent(80);
table1.setHorizontalAlignment(HorizontalAlignment.CENTER);
table1.addCell(
    new Cell(1, 3).add("Cell with colspan 3")
        .setPadding(10).setMargin(5).setBorder(new DashedBorder(0.5f)));
table1.addCell(new Cell(2, 1).add("Cell with rowspan 2")
    .setMarginTop(5).setMarginBottom(5)
    .setBorderBottom(new DottedBorder(0.5f))
    .setBorderLeft(new DottedBorder(0.5f)));
table1.addCell(new Cell().add("row 1; cell 1")
    .setBorder(new DottedBorder(Color.ORANGE, 0.5f)));
table1.addCell(new Cell().add("row 1; cell 2"));
table1.addCell(new Cell().add("row 2; cell 1").setMargin(10)
    .setBorderBottom(new SolidBorder(2)));
```

```
16 table1.addCell(new Cell().add("row 2; cell 2").setPadding(10)
17     .setBorderBottom(new SolidBorder(2)));
18 document.add(table1);
```

Let's compare the code and the resulting table, shown in figure 5.8.

- line 4-6: The first cell has a dashed border that is 0.5 user units wide. The border consists of a complete rectangle.
- line 7-10: For the second cell, we only defined a bottom border and a left border. A dotted line is drawn to the left and at the bottom of the cell. The top and the right border are actually the borders of other cells.
- line 11-12: We introduce an orange dotted border that is 0.5 user units wide. Although we set the border for the full cell, the top border isn't drawn as an orange dotted line. The top border is part of the dashed border of our first cell; iText won't draw an extra border on top of that already existing border.
- line 13: We don't define a border. By default, a solid border of 0.5 user units is drawn. Two borders were already defined previously, in the context of other, previously added cells. The borders of those cells prevail.
- line 14-15 and line 16-17: We define a solid bottom border that is 2 user units wide. The top borders of both cells are already defined: they are also the bottom borders of the corresponding cells in the previous row. The left and right borders aren't defined anywhere; iText will use the default border: a solid line of 0.5 user units.

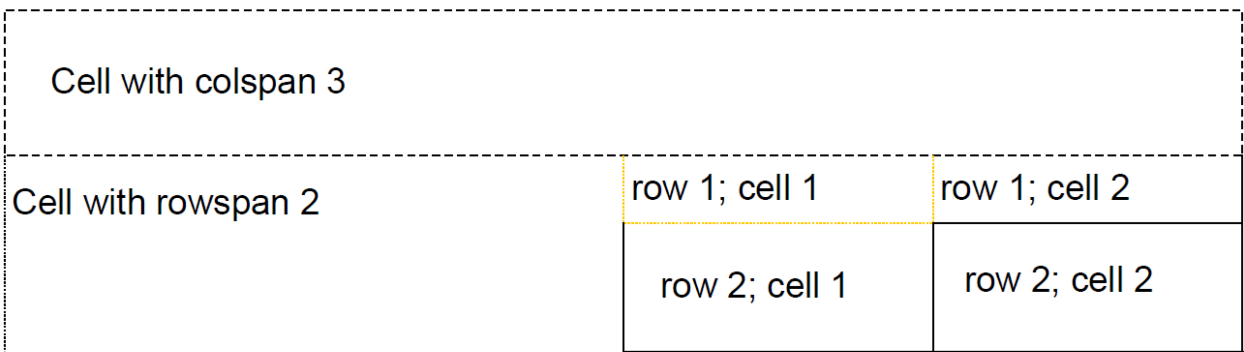



**Figure 5.8: different borders for different cells**

This behavior is the result of a design decision.

One way to deal with borders would be to let every `Cell`, or more specifically every `CellRenderer`, draw its own borders. In that case, the borders of adjacent cells would overlap. For instance: the dashed border at the bottom of the cell in the first row would overlap with the orange dotted top border of a cell in the second row. This is what happened in previous versions of iText. The border of two adjacent cells often consisted of two identical lines that overlapped each other. The extra line wasn't only redundant, it also caused a visual side-effect in some viewers. Many viewers render identical content that overlaps in a special way. In the case of overlapping text, a regular font looks as if it is bold. In the case of overlapping lines, the line width looks thicker than defined. The line width of two lines that are 0.5 user units wide and that are added at the exact same coordinates is rendered with a width slightly higher than 0.5 user units. Although this difference isn't always visible to the naked eye, we made the design decision to avoid this. All the borders are drawn at the level of the `Table`. That is: at the level of the `TableRenderer`.

In the next example, we define a border for the table, while setting the borders of every cell to `Border.NO_BORDER`.

```
1  Table table2 = new Table(new float[]{2, 1, 1});
2  table2.setMarginTop(10);
3  table2.setBorder(new SolidBorder(1));
4  table2.setWidthPercent(80);
5  table2.setHorizontalAlignment(HorizontalAlignment.CENTER);
6  table2.addCell(new Cell(1, 3)
7      .add("Cell with colspan 3").setBorder(Border.NO_BORDER));
8  table2.addCell(new Cell(2, 1)
9      .add("Cell with rowspan 2").setBorder(Border.NO_BORDER));
10 table2.addCell(new Cell()
11     .add("row 1; cell 1").setBorder(Border.NO_BORDER));
12 table2.addCell(new Cell()
13     .add("row 1; cell 2").setBorder(Border.NO_BORDER));
14 table2.addCell(new Cell()
15     .add("row 2; cell 1").setBorder(Border.NO_BORDER));
16 table2.addCell(new Cell()
17     .add("row 2; cell 2").setBorder(Border.NO_BORDER));
18 document.add(table2);
```

The result is shown in figure 5.9. The table has a border, but the cells don't have any "inside borders".

| Cell with colspan 3 | | |
|---|---|---|
| Cell with rowspan 2 | row 1; cell 1 | row 1; cell 2 |
| | row 2; cell 1 | row 2; cell 2 |

**Figure 5.9: table border but no cell borders**

Our design decision also has an impact on how we deal with custom renderers for cells. Suppose that we'd want to create cells with rounded borders. In that case, we could extend the `CellRenderer` class and create a `RoundedCornersCellRenderer` like this:

```
private class RoundedCornersCellRenderer extends CellRenderer {
    public RoundedCornersCellRenderer(Cell modelElement) {
        super(modelElement);
    }
    @Override
    public void drawBorder(DrawContext drawContext) {
        Rectangle occupiedAreaBBox = getOccupiedAreaBBox();
        float[] margins = getMargins();
        Rectangle rectangle = applyMargins(occupiedAreaBBox, margins, false);
        PdfCanvas canvas = drawContext.getCanvas();
        canvas.roundRectangle(rectangle.getX() + 1, rectangle.getY() + 1,
            rectangle.getWidth() - 2, rectangle.getHeight() -2, 5).stroke();
        super.drawBorder(drawContext);
    }
}
```

In the previous chapter, we've used the `setNextRenderer()` method to replace the default `ParagraphRenderer` of a `Paragraph` by our custom renderer. We could do the same with every `Cell` we create. In that case, we'd have something like:

```
Cell cell = new Cell();
cell.setNextRenderer(new RoundedCornersCellRenderer(cell));
```

However, we don't like having to do this for every `Cell` we create. It's much easier to extend the `Cell` class, overriding the `makeNewRenderer()` method.

```
private class RoundedCornersCell extends Cell {
    public RoundedCornersCell() {
        super();
    }
    public RoundedCornersCell(int rowspan, int colspan) {
        super(rowspan, colspan);
    }
    @Override
    protected IRenderer makeNewRenderer() {
        return new RoundedCornersCellRenderer(this);
    }
}
```

We can now use the `RoundedCornersCell` object instead of the `Cell` object.

```
Table table3 = new Table(new float[]{2, 1, 1});
table3.setMarginTop(10);
table3.setWidthPercent(80);
table3.setHorizontalAlignment(HorizontalAlignment.CENTER);
Cell cell = new RoundedCornersCell(1, 3).add("Cell with colspan 3")
        .setPadding(10).setMargin(5).setBorder(Border.NO_BORDER);
table3.addCell(cell);
cell = new RoundedCornersCell(2, 1).add("Cell with rowspan 2")
    .setMarginTop(5).setMarginBottom(5);
table3.addCell(cell);
cell = new RoundedCornersCell().add("row 1; cell 1");
table3.addCell(cell);
cell = new RoundedCornersCell().add("row 1; cell 2");
table3.addCell(cell);
cell = new RoundedCornersCell().add("row 2; cell 1").setMargin(10);
table3.addCell(cell);
cell = new RoundedCornersCell().add("row 2; cell 2").setPadding(10);
table3.addCell(cell);
document.add(table3);
```

We removed the border of the first cell in line 6. We didn't remove the borders of the other cells. Looking at figure 5.10, we see that those cells have two borders.

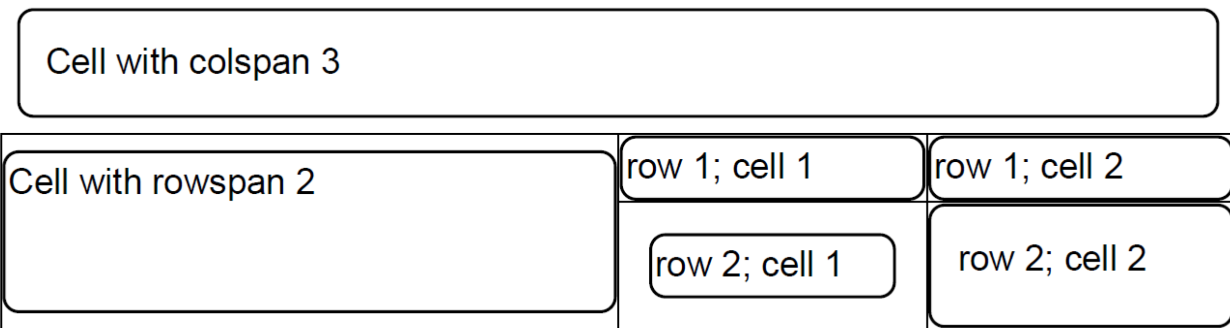



**Figure 5.10: custom borders**

This may be surprising: now that we've overridden the `drawBorder()` method of the `CellRenderer`, why is iText still drawing that extra border? We've already answered that question. We have made the design decision to draw the borders at the level of the `Table`. The original `drawBorder()` method in the `CellRenderer` class is empty. It doesn't draw any borders. If we want to use a custom border, we can either do what we've done in line 6 for every cell we create. The better solution would be to add `setBorder(Border.NO_BORDER);` to every `RoundedCornersCell` constructor.

In the next example, we'll add tables inside tables.

# Nesting tables

Figure 5.11 shows two or four tables, depending on how you look at the screen shot. There are two *outer* tables. Each of these tables has an *inner* table nested inside.

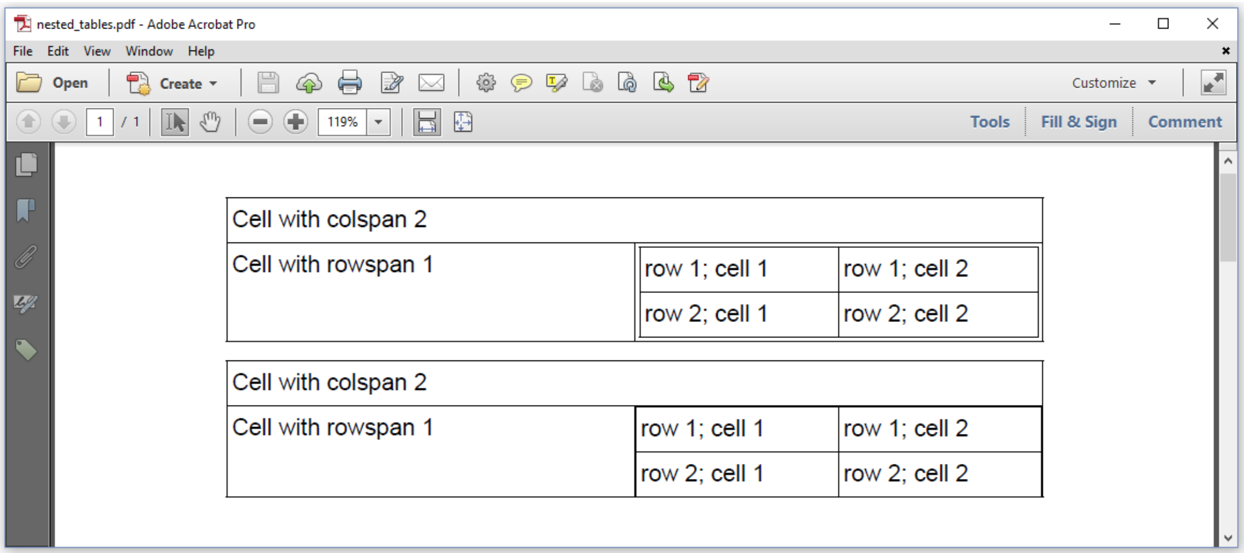



**Figure 5.11: nested tables**

Let's examine the NestedTable[80] example. This is how the first table was created:

[80] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2037-c05e07_nestedtable.java

```
Table table = new Table(2);
table.setWidthPercent(80);
table.setHorizontalAlignment(HorizontalAlignment.CENTER);
table.addCell(new Cell(1, 2).add("Cell with colspan 2"));
table.addCell(new Cell().add("Cell with rowspan 1"));
Table inner = new Table(2);
inner.addCell("row 1; cell 1");
inner.addCell("row 1; cell 2");
inner.addCell("row 2; cell 1");
inner.addCell("row 2; cell 2");
table.addCell(inner);
```

We create a `Table` object named `table`. We add four `Cell` objects to this table, but one `Cell` object is special. We created another `Table` object named `inner` and we added this table to the outer table `table` using the `addCell()` method. If we look at figure 5.11, we see that there's a padding between the border of the fourth cell and the border of the inner table. That's the default padding of `2` user units.

The second table was created in almost the exact same way as the first table. The main difference can be found in the last line.

```
table.addCell(new Cell().add(inner).setPadding(0));
```

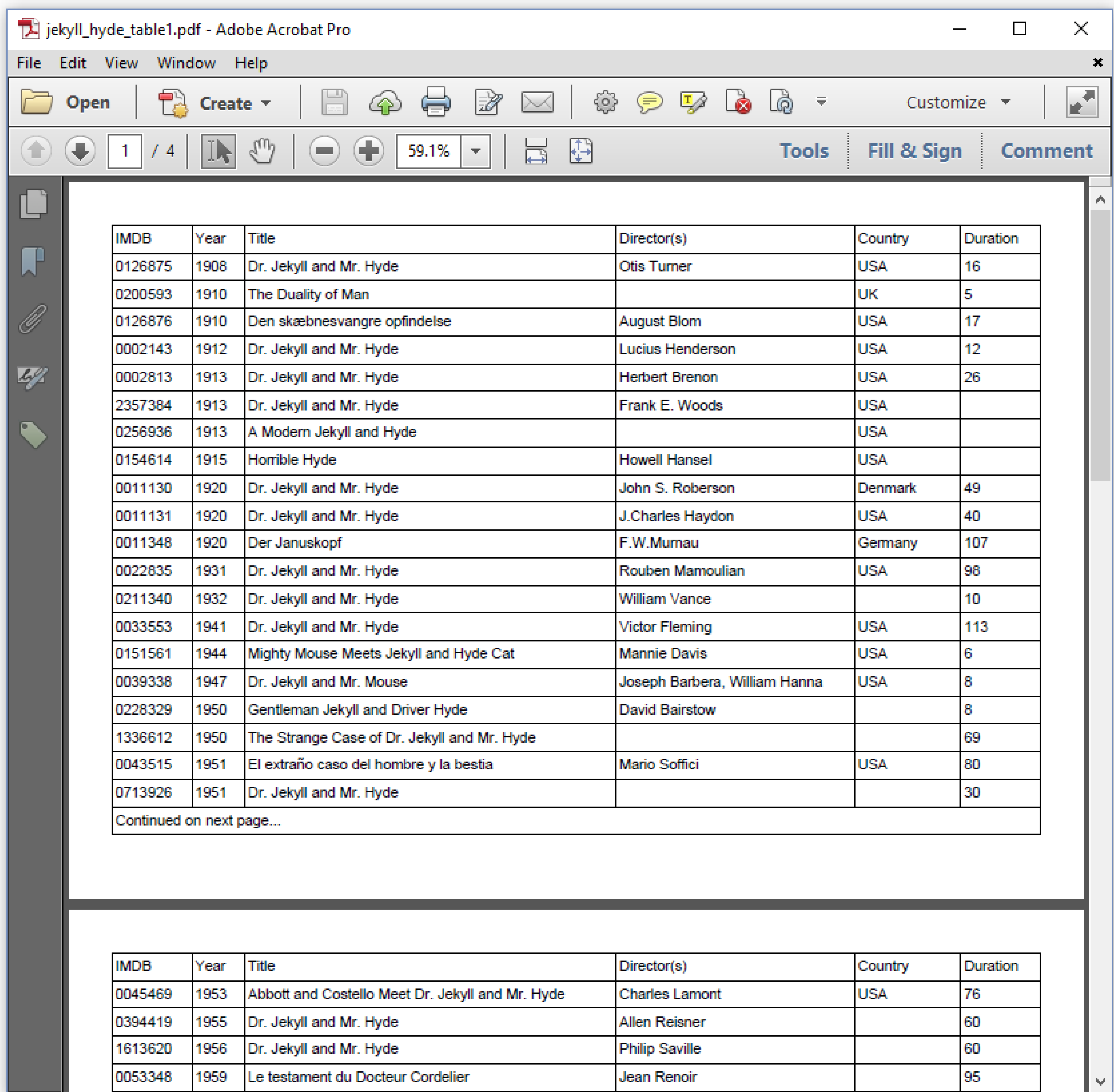
Instead of adding the nested table straight to the `table` object, we now create a `Cell` object to which we add the `inner` table. We set the padding of this cell to `0`. Now it looks as if the cell with content `"Cell with rowspan 1"` has a rowspan of `2`. This isn't the case. We have mimicked a rowspan of `2` by using a nested table.

If you look closely at the screen shot, you may see why you should avoid using nested tables. Common sense tells us that nesting tables has a negative impact on the performance of an application, but there's another reason why you might want to avoid using them in the context of iText. As mentioned before, all cell borders are drawn at the `Table` level. In this case, the border of the cell containing the nested table is drawn by the `TableRenderer` of the outer table `table`. The border of the cells of the nested table are drawn by the `TableRenderer` of the inner table `inner`. This results in overlapping lines, which may cause an undesired effect. In some PDF viewers, the width of the overlapping lines may seem to be wider than the width of each separate line.

Now let's switch to some examples that are less artificial. Let's convert our CSV file to a `Table` and render it to PDF.

## Repeating headers and footers

In chapter 3, we used `Tab` elements to render a database containing movies and videos based on Stevenson's story about Dr. Jekyll and Mr. Hyde in a tabular structure. Although this worked well, we experienced some disadvantages, for instance when the content didn't fit the space we had allocated. It's a much better idea to use a `Table` for this kind of work. Figure 5.12 shows how we introduced a repeating header with the column names and a repeating footer that reads "Continued on next page…" when the table doesn't fit the current page.



| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |
| 0151561 | 1944 | Mighty Mouse Meets Jekyll and Hyde Cat | Mannie Davis | USA | 6 |
| 0039338 | 1947 | Dr. Jekyll and Mr. Mouse | Joseph Barbera, William Hanna | USA | 8 |
| 0228329 | 1950 | Gentleman Jekyll and Driver Hyde | David Bairstow | | 8 |
| 1336612 | 1950 | The Strange Case of Dr. Jekyll and Mr. Hyde | | | 69 |
| 0043515 | 1951 | El extraño caso del hombre y la bestia | Mario Soffici | USA | 80 |
| 0713926 | 1951 | Dr. Jekyll and Mr. Hyde | | | 30 |
| Continued on next page... | | | | | |

| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0045469 | 1953 | Abbott and Costello Meet Dr. Jekyll and Mr. Hyde | Charles Lamont | USA | 76 |
| 0394419 | 1955 | Dr. Jekyll and Mr. Hyde | Allen Reisner | | 60 |
| 1613620 | 1956 | Dr. Jekyll and Mr. Hyde | Philip Saville | | 60 |
| 0053348 | 1959 | Le testament du Docteur Cordelier | Jean Renoir | | 95 |

**Figure 5.12: repeating headers and footers**

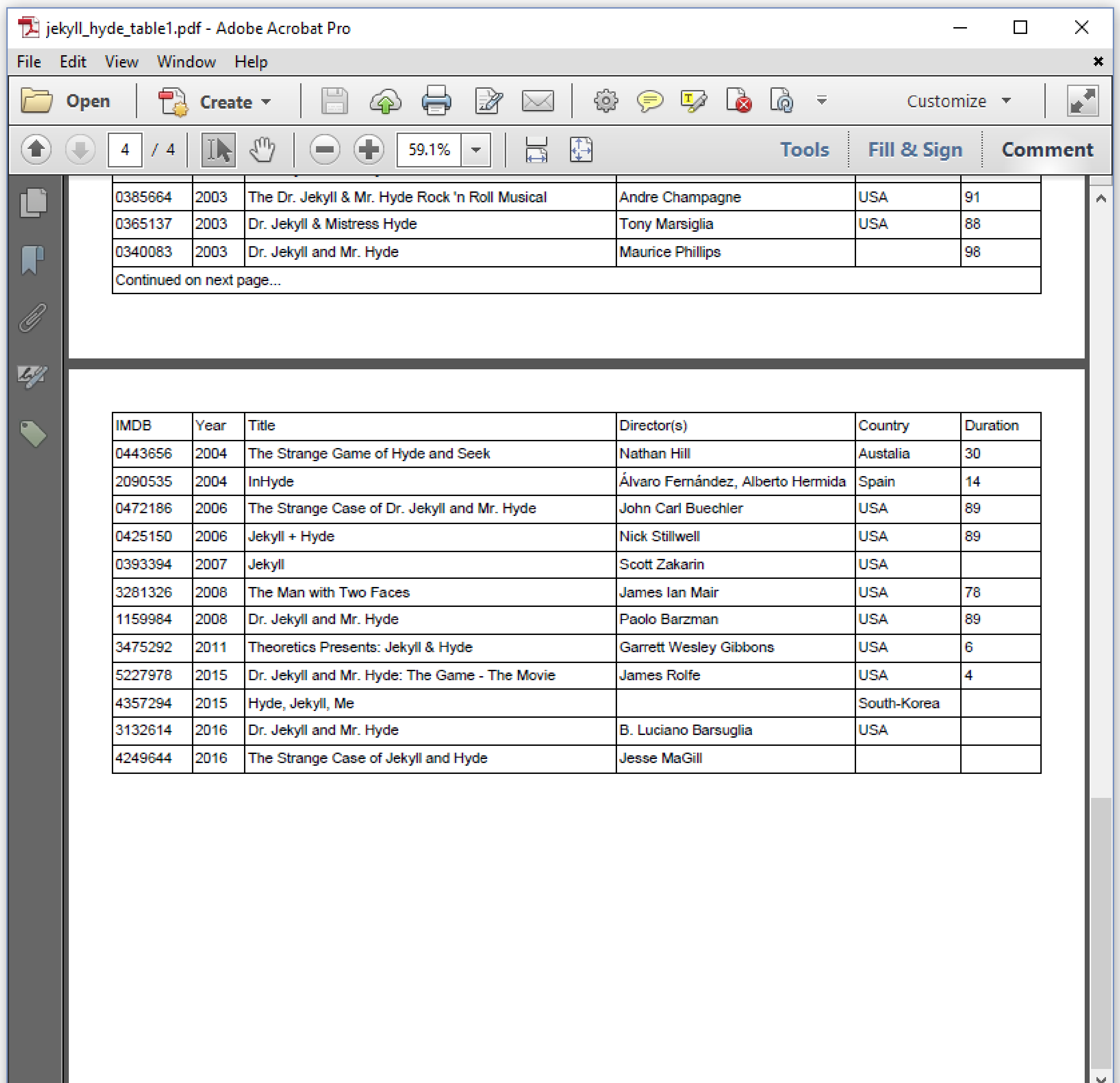
The JekyllHydeTableV1[81] example shows how it's done.

[81] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2038-c05e08_jekyllhydetablev1.java

```
1  Table table = new Table(new float[]{3, 2, 14, 9, 4, 3});
2  table.setWidthPercent(100);
3  List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
4  List<String> header = resultSet.remove(0);
5  for (String field : header) {
6      table.addHeaderCell(field);
7  }
8  Cell cell = new Cell(1, 6).add("Continued on next page...");
9  table.addFooterCell(cell)
10     .setSkipLastFooter(true);
11 for (List<String> record : resultSet) {
12     for (String field : record) {
13         table.addCell(field);
14     }
15 }
16 document.add(table);
```

We get our data from a CSV file (line 3) and we get the line containing the header information (line 4). Instead of using `addCell()`, we add each field in that line using the `addHeaderCell()` method. This marks these cell as header cells: they will be repeated at the top of the page every time a new page is started.

We also create footer cell that spans the six columns (line 8). We make this cell a footer cell by using the `addFooterCell()` method (line 9). We also instruct the table to skip the last footer (line 10). This way, the cell won't appear as a footer after the last row of the table. This is shown in figure 5.13.

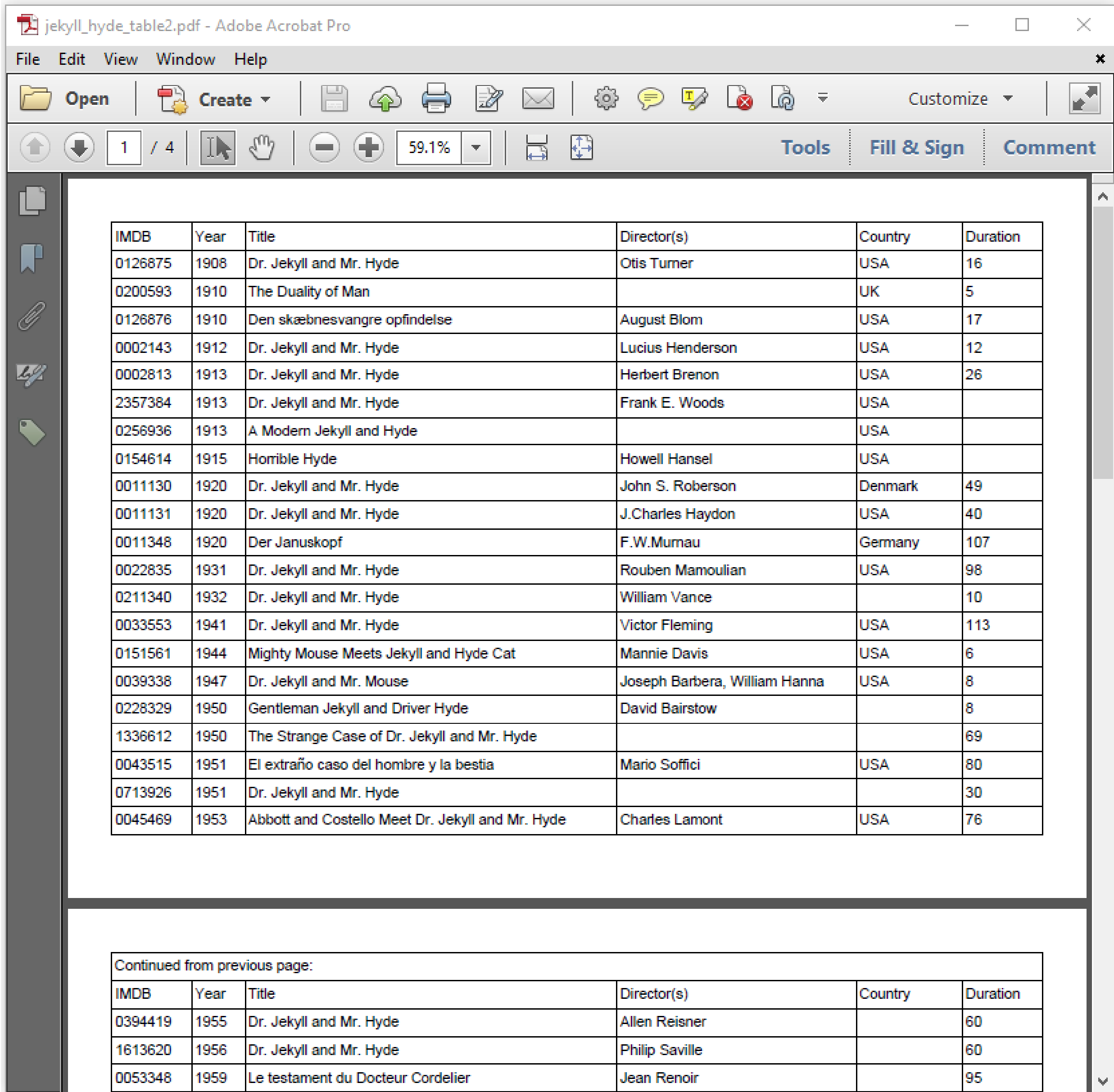



| | | | | | |
|---|---|---|---|---|---|
| 0385664 | 2003 | The Dr. Jekyll & Mr. Hyde Rock 'n Roll Musical | Andre Champagne | USA | 91 |
| 0365137 | 2003 | Dr. Jekyll & Mistress Hyde | Tony Marsiglia | USA | 88 |
| 0340083 | 2003 | Dr. Jekyll and Mr. Hyde | Maurice Phillips | | 98 |
| Continued on next page... | | | | | |

| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0443656 | 2004 | The Strange Game of Hyde and Seek | Nathan Hill | Austalia | 30 |
| 2090535 | 2004 | InHyde | Álvaro Fernández, Alberto Hermida | Spain | 14 |
| 0472186 | 2006 | The Strange Case of Dr. Jekyll and Mr. Hyde | John Carl Buechler | USA | 89 |
| 0425150 | 2006 | Jekyll + Hyde | Nick Stillwell | USA | 89 |
| 0393394 | 2007 | Jekyll | Scott Zakarin | USA | |
| 3281326 | 2008 | The Man with Two Faces | James Ian Mair | USA | 78 |
| 1159984 | 2008 | Dr. Jekyll and Mr. Hyde | Paolo Barzman | USA | 89 |
| 3475292 | 2011 | Theoretics Presents: Jekyll & Hyde | Garrett Wesley Gibbons | USA | 6 |
| 5227978 | 2015 | Dr. Jekyll and Mr. Hyde: The Game - The Movie | James Rolfe | USA | 4 |
| 4357294 | 2015 | Hyde, Jekyll, Me | | South-Korea | |
| 3132614 | 2016 | Dr. Jekyll and Mr. Hyde | B. Luciano Barsuglia | USA | |
| 4249644 | 2016 | The Strange Case of Jekyll and Hyde | Jesse MaGill | | |

**Figure 5.13: repeating headers and footers**

There is also a way to skip the first header. See figure 5.14.



| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |
| 0151561 | 1944 | Mighty Mouse Meets Jekyll and Hyde Cat | Mannie Davis | USA | 6 |
| 0039338 | 1947 | Dr. Jekyll and Mr. Mouse | Joseph Barbera, William Hanna | USA | 8 |
| 0228329 | 1950 | Gentleman Jekyll and Driver Hyde | David Bairstow | | 8 |
| 1336612 | 1950 | The Strange Case of Dr. Jekyll and Mr. Hyde | | | 69 |
| 0043515 | 1951 | El extraño caso del hombre y la bestia | Mario Soffici | USA | 80 |
| 0713926 | 1951 | Dr. Jekyll and Mr. Hyde | | | 30 |
| 0045469 | 1953 | Abbott and Costello Meet Dr. Jekyll and Mr. Hyde | Charles Lamont | USA | 76 |

Continued from previous page:

| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0394419 | 1955 | Dr. Jekyll and Mr. Hyde | Allen Reisner | | 60 |
| 1613620 | 1956 | Dr. Jekyll and Mr. Hyde | Philip Saville | | 60 |
| 0053348 | 1959 | Le testament du Docteur Cordelier | Jean Renoir | | 95 |

**Figure 5.14: repeating headers**

In this case, we had to use nested tables, because we have two types of headers. We have a header that needs to be skipped on the first page. We also have a header that needs to appear on every page. The JekyllHydeTableV2[82] example shows how it's done.

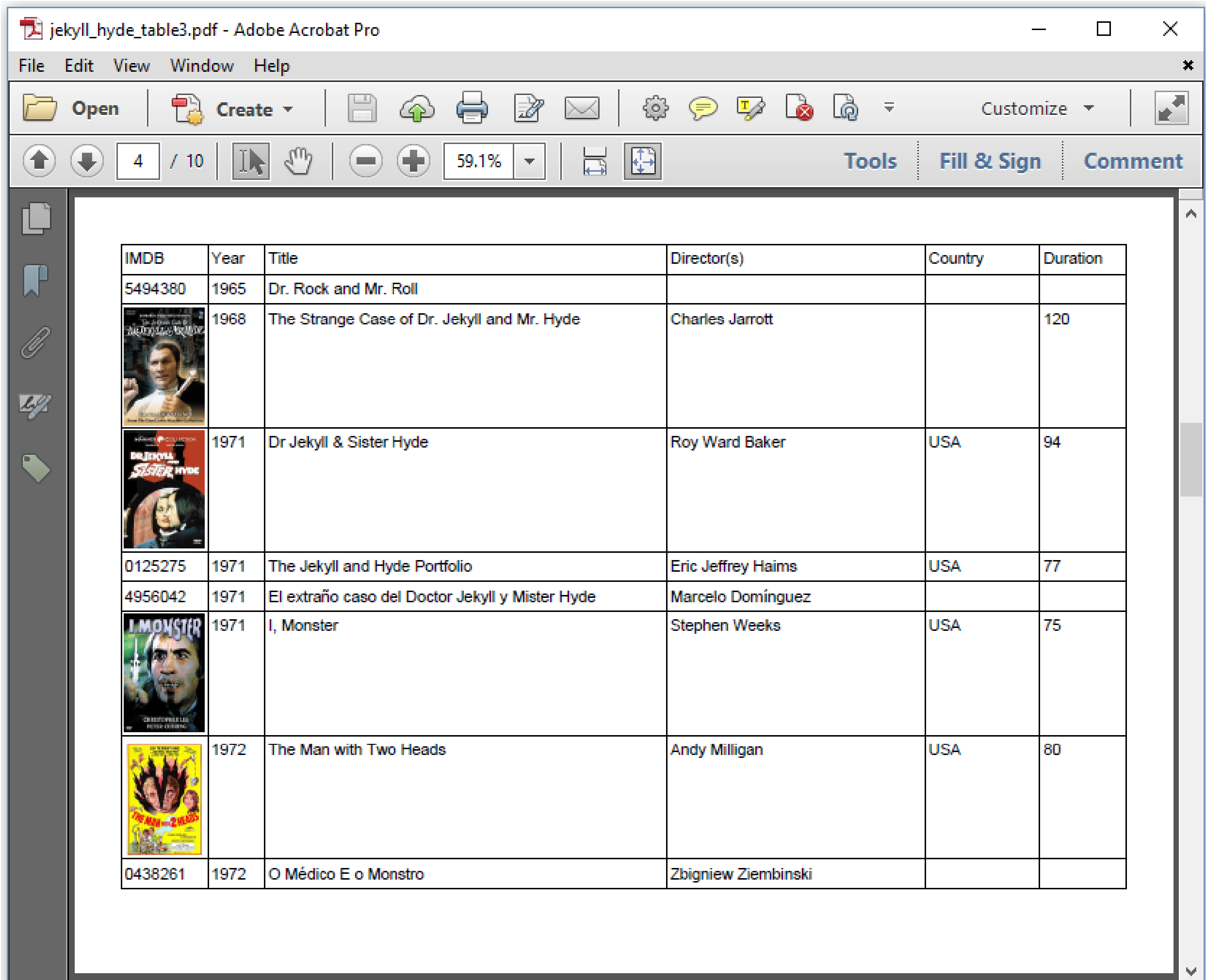
[82] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2039-c05e09_jekyllhydetablev2.java

```
Table table = new Table(new float[]{3, 2, 14, 9, 4, 3});
table.setWidthPercent(100);
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
List<String> header = resultSet.remove(0);
for (String field : header) {
    table.addHeaderCell(field);
}
for (List<String> record : resultSet) {
    for (String field : record) {
        table.addCell(field);
    }
}
Table outerTable = new Table(1)
    .addHeaderCell("Continued from previous page:")
    .setSkipFirstHeader(true)
    .addCell(new Cell().add(table).setPadding(0));
document.add(outerTable);
```

Lines 1-12 should have no secrets to us. In lines 13-16, we use what we've learned when we discussed nested tables to create an outer table with a second header. We use the `setSkipFirstHeader()` method to make sure that header doesn't appear on the first page, only on subsequent pages.

## Images in tables

Figure 5.15 demonstrates that we can also add images to a table. We can even make them scale so that they fit the width of the cell.




| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 5494380 | 1965 | Dr. Rock and Mr. Roll | | | |
| | 1968 | The Strange Case of Dr. Jekyll and Mr. Hyde | Charles Jarrott | | 120 |
| | 1971 | Dr Jekyll & Sister Hyde | Roy Ward Baker | USA | 94 |
| 0125275 | 1971 | The Jekyll and Hyde Portfolio | Eric Jeffrey Haims | USA | 77 |
| 4956042 | 1971 | El extraño caso del Doctor Jekyll y Mister Hyde | Marcelo Domínguez | | |
| | 1971 | I, Monster | Stephen Weeks | USA | 75 |
| | 1972 | The Man with Two Heads | Andy Milligan | USA | 80 |
| 0438261 | 1972 | O Médico E o Monstro | Zbigniew Ziembinski | | |

**Figure 5.15: images in tables**

That's done in the JekyllHydeTableV3[83] example.

```
Table table = new Table(new float[]{3, 2, 14, 9, 4, 3});
table.setWidthPercent(100);
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
List<String> header = resultSet.remove(0);
for (String field : header) {
    table.addHeaderCell(field);
}
Cell cell;
for (List<String> record : resultSet) {
    cell = new Cell();
```

[83] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2040-c05e10_jekyllhydetablev3.java

```
11        File file = new File(String.format(
12            "src/main/resources/img/%s.jpg", record.get(0)));
13        if (file.exists()) {
14            Image img = new Image(ImageDataFactory.create(file.getPath()));
15            img.setAutoScaleWidth(true);
16            cell.add(img);
17        }
18        else {
19            cell.add(record.get(0));
20        }
21        table.addCell(cell);
22        table.addCell(record.get(1));
23        table.addCell(record.get(2));
24        table.addCell(record.get(3));
25        table.addCell(record.get(4));
26        table.addCell(record.get(5));
27    }
28    document.add(table);
```

We can add the image to a `Cell` using the `add()` method –the same way we've added content to a `Cell` before. We use the `setAutoScaleWidth()` method to tell the image that it should try to scale itself to fit the width of its container, in this case the `Cell` to which it is added.

There's also a `setAutoScaleHeight()` method if you want the images to scale automatically depending on the available height, and a `setAutoScale()` method to scale the image based on the width and the height.

Not scaling images can result in ugly tables; when the images are too large for the cell, they will take up space from the adjacent cells.

# Splitting cells versus keeping content together

We're not using any images in figure 5.16. The second column just contains information that consists of different `Paragraph` objects added to a `Cell`.

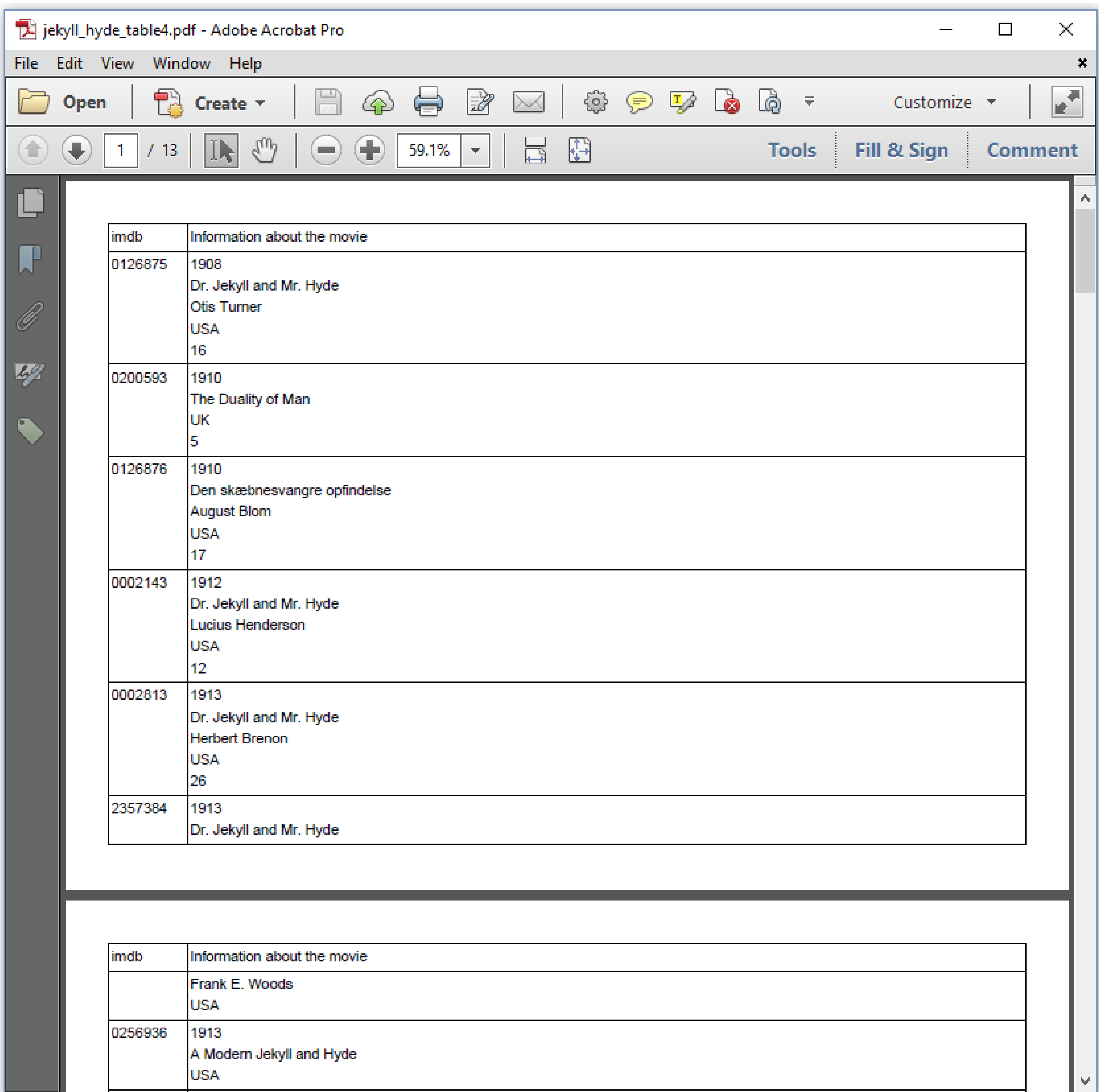


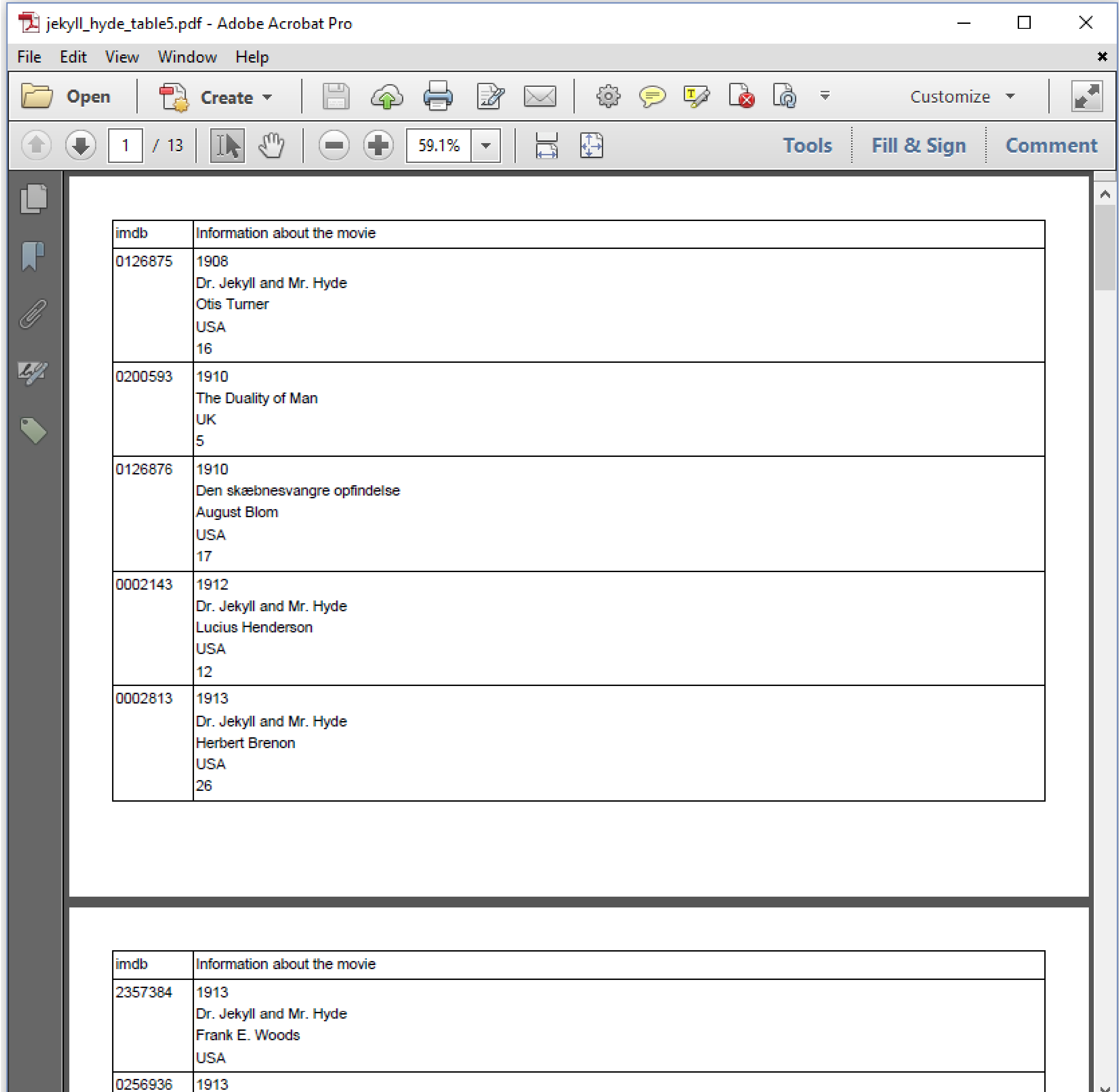

**Figure 5.16: splitting cell that don't fit the page**

When the content doesn't fit the page, the cell is split. The production year and title are on one page, the director and the country the movie was produced in on the other page. This is the default behavior when you write your code as done in the JekyllHydeTabV4[84] example.

[84]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2041-c05e11_jekyllhydetablev4.java

```
Table table = new Table(new float[]{3, 32});
table.setWidthPercent(100);
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
resultSet.remove(0);
table.addHeaderCell("imdb")
    .addHeaderCell("Information about the movie");
Cell cell;
for (List<String> record : resultSet) {
    table.addCell(record.get(0));
    cell = new Cell()
        .add(new Paragraph(record.get(1)))
        .add(new Paragraph(record.get(2)))
        .add(new Paragraph(record.get(3)))
        .add(new Paragraph(record.get(4)))
        .add(new Paragraph(record.get(5)));
    table.addCell(cell);
}
document.add(table);
```

You may want iText to do an effort to keep the content of a cell together on one page (if possible).



| imdb | Information about the movie |
|---|---|
| 0126875 | 1908<br>Dr. Jekyll and Mr. Hyde<br>Otis Turner<br>USA<br>16 |
| 0200593 | 1910<br>The Duality of Man<br>UK<br>5 |
| 0126876 | 1910<br>Den skæbnesvangre opfindelse<br>August Blom<br>USA<br>17 |
| 0002143 | 1912<br>Dr. Jekyll and Mr. Hyde<br>Lucius Henderson<br>USA<br>12 |
| 0002813 | 1913<br>Dr. Jekyll and Mr. Hyde<br>Herbert Brenon<br>USA<br>26 |

| imdb | Information about the movie |
|---|---|
| 2357384 | 1913<br>Dr. Jekyll and Mr. Hyde<br>Frank E. Woods<br>USA |
| 0256936 | 1913 |

**Figure 5.17: keeping cell content together**

The PDF in the screen shot of figure 5.17 was created using the JekyllHydeTableV5[85] example. There's only one difference with the previous example. We've added the following line of code after line 15:

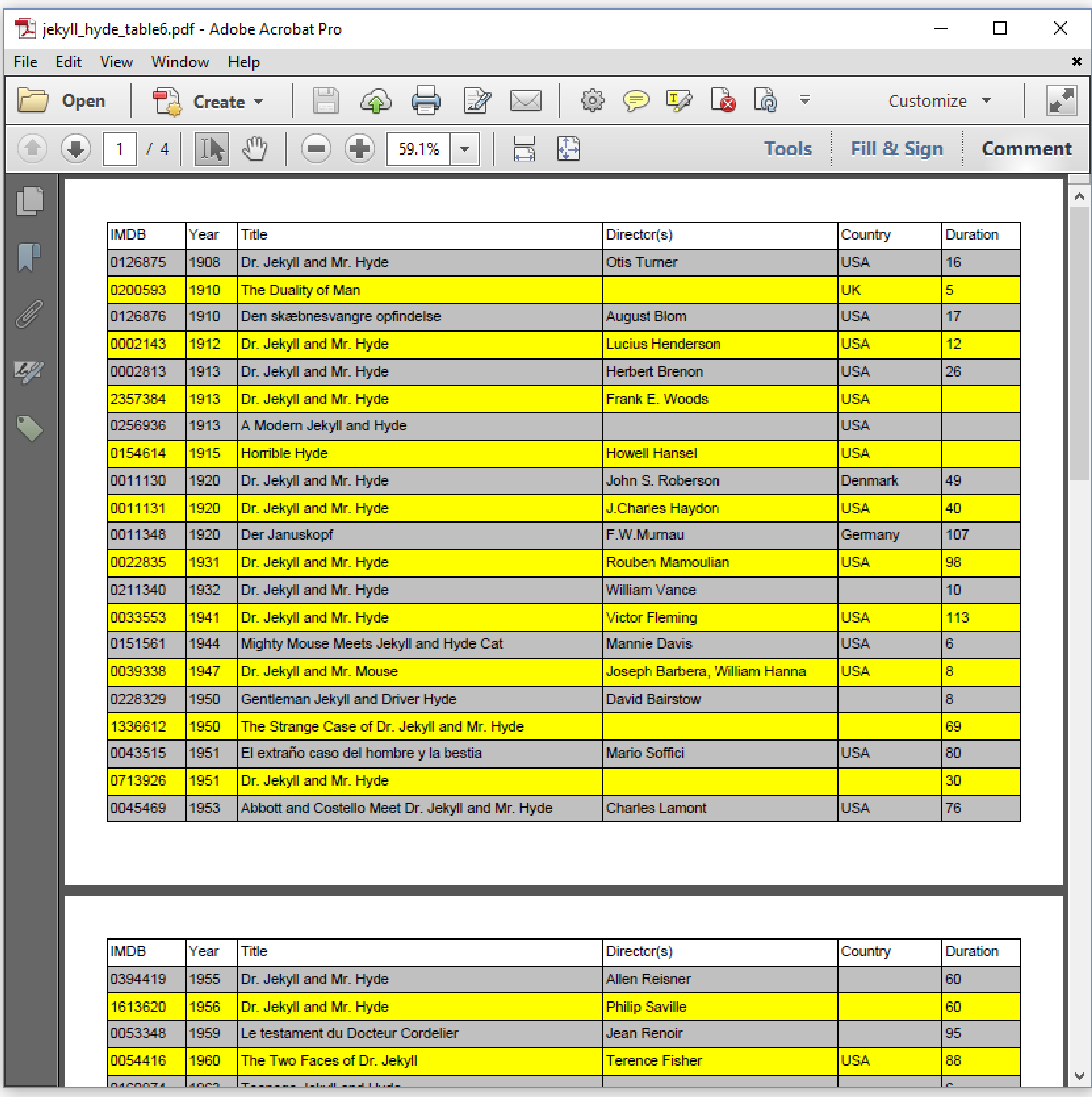
```
1 cell.setKeepTogether(true);
```

The `setKeepTogether()` method is defined at the `BlockElement` level. We've used that method before in the previous chapter. Note that the `setKeepWithNext()` can't be used in this context, because we're not adding the `Cell` object directly to the `Document`.

[85] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2042-c05e12_jekyllhydetablev5.java

# Table and cell renderers

Let's make some more renderer methods. We've already created a custom `CellRenderer` to add rounded corners. In figure 5.18, we're introducing a `TableRenderer` to display alternate backgrounds for the rows.




| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0126875 | 1908 | Dr. Jekyll and Mr. Hyde | Otis Turner | USA | 16 |
| 0200593 | 1910 | The Duality of Man | | UK | 5 |
| 0126876 | 1910 | Den skæbnesvangre opfindelse | August Blom | USA | 17 |
| 0002143 | 1912 | Dr. Jekyll and Mr. Hyde | Lucius Henderson | USA | 12 |
| 0002813 | 1913 | Dr. Jekyll and Mr. Hyde | Herbert Brenon | USA | 26 |
| 2357384 | 1913 | Dr. Jekyll and Mr. Hyde | Frank E. Woods | USA | |
| 0256936 | 1913 | A Modern Jekyll and Hyde | | USA | |
| 0154614 | 1915 | Horrible Hyde | Howell Hansel | USA | |
| 0011130 | 1920 | Dr. Jekyll and Mr. Hyde | John S. Roberson | Denmark | 49 |
| 0011131 | 1920 | Dr. Jekyll and Mr. Hyde | J.Charles Haydon | USA | 40 |
| 0011348 | 1920 | Der Januskopf | F.W.Murnau | Germany | 107 |
| 0022835 | 1931 | Dr. Jekyll and Mr. Hyde | Rouben Mamoulian | USA | 98 |
| 0211340 | 1932 | Dr. Jekyll and Mr. Hyde | William Vance | | 10 |
| 0033553 | 1941 | Dr. Jekyll and Mr. Hyde | Victor Fleming | USA | 113 |
| 0151561 | 1944 | Mighty Mouse Meets Jekyll and Hyde Cat | Mannie Davis | USA | 6 |
| 0039338 | 1947 | Dr. Jekyll and Mr. Mouse | Joseph Barbera, William Hanna | USA | 8 |
| 0228329 | 1950 | Gentleman Jekyll and Driver Hyde | David Bairstow | | 8 |
| 1336612 | 1950 | The Strange Case of Dr. Jekyll and Mr. Hyde | | | 69 |
| 0043515 | 1951 | El extraño caso del hombre y la bestia | Mario Soffici | USA | 80 |
| 0713926 | 1951 | Dr. Jekyll and Mr. Hyde | | | 30 |
| 0045469 | 1953 | Abbott and Costello Meet Dr. Jekyll and Mr. Hyde | Charles Lamont | USA | 76 |

| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0394419 | 1955 | Dr. Jekyll and Mr. Hyde | Allen Reisner | | 60 |
| 1613620 | 1956 | Dr. Jekyll and Mr. Hyde | Philip Saville | | 60 |
| 0053348 | 1959 | Le testament du Docteur Cordelier | Jean Renoir | | 95 |
| 0054416 | 1960 | The Two Faces of Dr. Jekyll | Terence Fisher | USA | 88 |

**Figure 5.18: creating alternate backgrounds using a TableRenderer**

Let's take a look at the JekyllHydeTableV6[86] example to see what this custom `TableRenderer` looks

[86] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2043-c05e13_jekyllhydetablev6.java

like.

```
class AlternatingBackgroundTableRenderer extends TableRenderer {
    private boolean isOdd = true;
    public AlternatingBackgroundTableRenderer(
        Table modelElement, Table.RowRange rowRange) {
        super(modelElement, rowRange);
    }
    public AlternatingBackgroundTableRenderer(Table modelElement) {
        super(modelElement);
    }
    @Override
    public AlternatingBackgroundTableRenderer getNextRenderer() {
        return new AlternatingBackgroundTableRenderer(
            (Table) modelElement);
    }
    @Override
    public void draw(DrawContext drawContext) {
        for (int i = 0;
            i < rows.size() && null != rows.get(i) && null != rows.get(i)[0];
            i++) {
            CellRenderer[] renderers = rows.get(i);
            Rectangle leftCell =
                renderers[0].getOccupiedAreaBBox();
            Rectangle rightCell =
                renderers[renderers.length - 1].getOccupiedAreaBBox();
            Rectangle rect = new Rectangle(
                leftCell.getLeft(), leftCell.getBottom(),
                rightCell.getRight() - leftCell.getLeft(),
                leftCell.getHeight());
            PdfCanvas canvas = drawContext.getCanvas();
            canvas.saveState();
            if (isOdd) {
                canvas.setFillColor(Color.LIGHT_GRAY);
                isOdd = false;
            } else {
                canvas.setFillColor(Color.YELLOW);
                isOdd = true;
            }
            canvas.rectangle(rect);
            canvas.fill();
            canvas.restoreState();
        }
```

```
42          super.draw(drawContext);
43      }
44  }
```

We create constructors that are similar to the `TableRenderer` constructors (line 3-9), and we override the `getNextRenderer()` method so that it returns an `AlternatingBackgroundTableRenderer` (line 10-14). We introduce a `boolean` variable named `isOdd` to keep track of the rows (line 2).

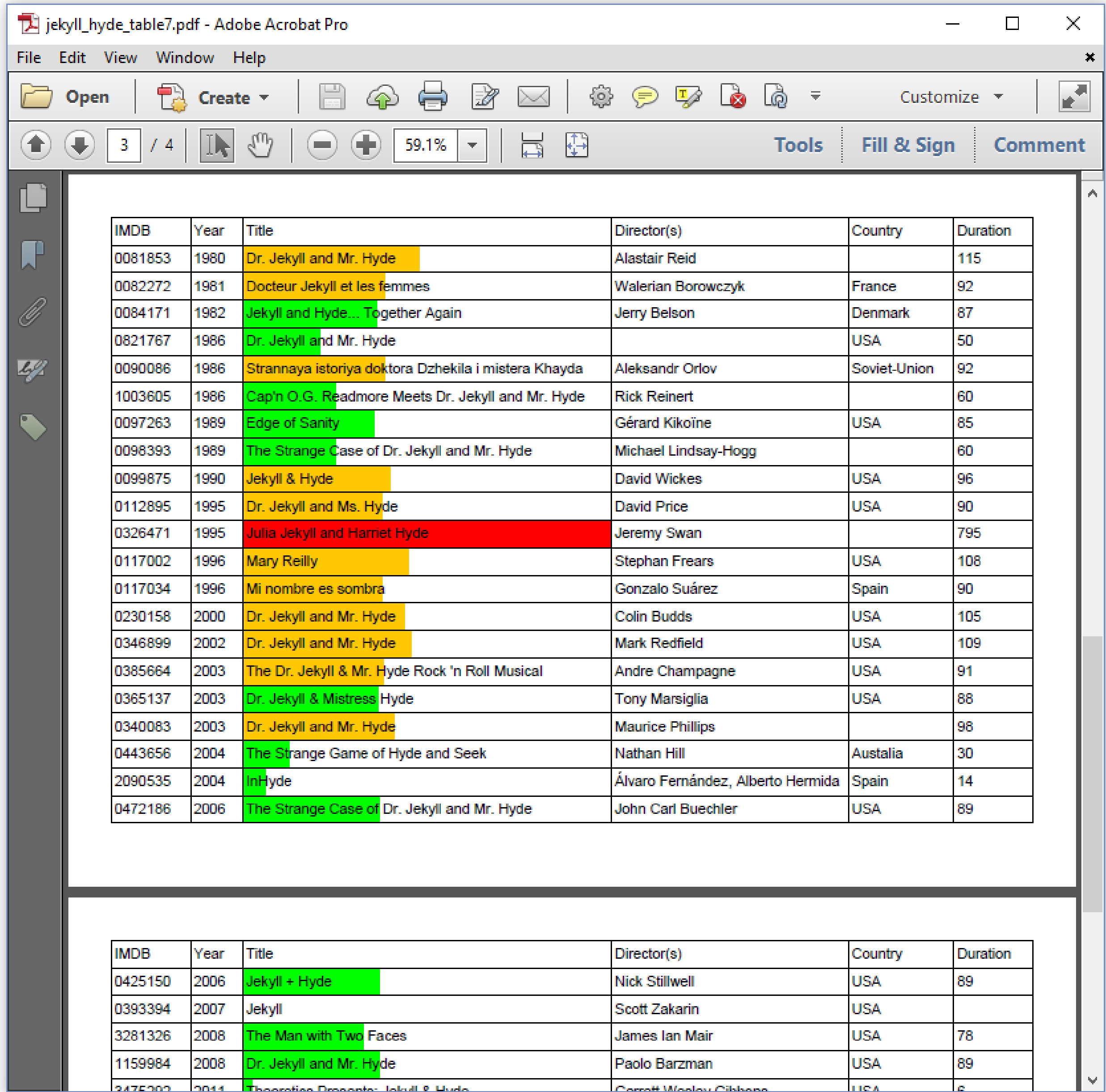
The `draw()` method is where we do our magic (line 15-43). We loop over the rows (line 17-19), and we get the `CellRenderer` instances of all the cells in each row (line 20). We get the renderer of the left cell and the right cell in each row (line 21-24), and we use those renderers to determine the coordinates of the row (line 25-28). We draw the `Rectangle` based on those coordinates in a color that depends on the alternating value of the `isOdd` parameter (line 29-40).

In the next code snippet, we'll create a table, and we'll declare the `AlternatingBackgroundTableRenderer` as the new renderer for that table.

```
1  Table table = new Table(new float[]{3, 2, 14, 9, 4, 3});
2  int nRows = resultSet.size();
3  table.setNextRenderer(new AlternatingBackgroundTableRenderer(
4      table, new Table.RowRange(0, nRows - 1)));
```

Note that we have to define the `RowRange`. We take the number of elements in our `resultSet` after having removed the header row. That gives us the number of actual rows that we are going to add to the table, and to which we want to apply an alternating background.

Figure 5.19 shows another type of background. The width of the "Title" column represents four hours; the colored bar in the "Title" cells represents the run length of the video. For instance: if the colored bar takes half of the width of the cell, the run length of the movie is half of four hours; that is: two hours.



| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0081853 | 1980 | Dr. Jekyll and Mr. Hyde | Alastair Reid | | 115 |
| 0082272 | 1981 | Docteur Jekyll et les femmes | Walerian Borowczyk | France | 92 |
| 0084171 | 1982 | Jekyll and Hyde... Together Again | Jerry Belson | Denmark | 87 |
| 0821767 | 1986 | Dr. Jekyll and Mr. Hyde | | USA | 50 |
| 0090086 | 1986 | Strannaya istoriya doktora Dzhekila i mistera Khayda | Aleksandr Orlov | Soviet-Union | 92 |
| 1003605 | 1986 | Cap'n O.G. Readmore Meets Dr. Jekyll and Mr. Hyde | Rick Reinert | | 60 |
| 0097263 | 1989 | Edge of Sanity | Gérard Kikoïne | USA | 85 |
| 0098393 | 1989 | The Strange Case of Dr. Jekyll and Mr. Hyde | Michael Lindsay-Hogg | | 60 |
| 0099875 | 1990 | Jekyll & Hyde | David Wickes | USA | 96 |
| 0112895 | 1995 | Dr. Jekyll and Ms. Hyde | David Price | USA | 90 |
| 0326471 | 1995 | Julia Jekyll and Harriet Hyde | Jeremy Swan | | 795 |
| 0117002 | 1996 | Mary Reilly | Stephan Frears | USA | 108 |
| 0117034 | 1996 | Mi nombre es sombra | Gonzalo Suárez | Spain | 90 |
| 0230158 | 2000 | Dr. Jekyll and Mr. Hyde | Colin Budds | USA | 105 |
| 0346899 | 2002 | Dr. Jekyll and Mr. Hyde | Mark Redfield | USA | 109 |
| 0385664 | 2003 | The Dr. Jekyll & Mr. Hyde Rock 'n Roll Musical | Andre Champagne | USA | 91 |
| 0365137 | 2003 | Dr. Jekyll & Mistress Hyde | Tony Marsiglia | USA | 88 |
| 0340083 | 2003 | Dr. Jekyll and Mr. Hyde | Maurice Phillips | | 98 |
| 0443656 | 2004 | The Strange Game of Hyde and Seek | Nathan Hill | Austalia | 30 |
| 2090535 | 2004 | InHyde | Álvaro Fernández, Alberto Hermida | Spain | 14 |
| 0472186 | 2006 | The Strange Case of Dr. Jekyll and Mr. Hyde | John Carl Buechler | USA | 89 |

| IMDB | Year | Title | Director(s) | Country | Duration |
|---|---|---|---|---|---|
| 0425150 | 2006 | Jekyll + Hyde | Nick Stillwell | USA | 89 |
| 0393394 | 2007 | Jekyll | Scott Zakarin | USA | |
| 3281326 | 2008 | The Man with Two Faces | James Ian Mair | USA | 78 |
| 1159984 | 2008 | Dr. Jekyll and Mr. Hyde | Paolo Barzman | USA | 89 |

**Figure 5.19: introducing visual information using a CellRenderer**

These are the color codes we used:

- *No background*– we don't know the run length of the movie,
- *Green background*– the movie is shorter than 90 minutes,
- *Orange background*– the movie is longer than 90 minutes, but shorter than 4 hours,
- *Red background*– the move is longer than 4 hours (e.g. it's a series with many episodes). In this case, we clip the length to 240 minutes.

The code for the custom `CellRenderer` to achieve this can be found in the JekyllHydeTable7[87] example.

```
 1 private class RunlengthRenderer extends CellRenderer {
 2     private int runlength;
 3     public RunlengthRenderer(Cell modelElement, String duration) {
 4         super(modelElement);
 5         if (duration.trim().isEmpty()) runlength = 0;
 6         else runlength = Integer.parseInt(duration);
 7     }
 8     @Override
 9     public CellRenderer getNextRenderer() {
10         return new RunlengthRenderer(
11             getModelElement(), String.valueOf(runlength));
12     }
13     @Override
14     public void drawBackground(DrawContext drawContext) {
15         if (runlength == 0) return;
16         PdfCanvas canvas = drawContext.getCanvas();
17         canvas.saveState();
18         if (runlength < 90) {
19             canvas.setFillColor(Color.GREEN);
20         } else if (runlength > 240) {
21             runlength = 240;
22             canvas.setFillColor(Color.RED);
23         } else {
24             canvas.setFillColor(Color.ORANGE);
25         }
26         Rectangle rect = getOccupiedAreaBBox();
27         canvas.rectangle(rect.getLeft(), rect.getBottom(),
28                 rect.getWidth() * runlength / 240, rect.getHeight());
29         canvas.fill();
30         canvas.restoreState();
31         super.drawBackground(drawContext);
32     }
33 }
```

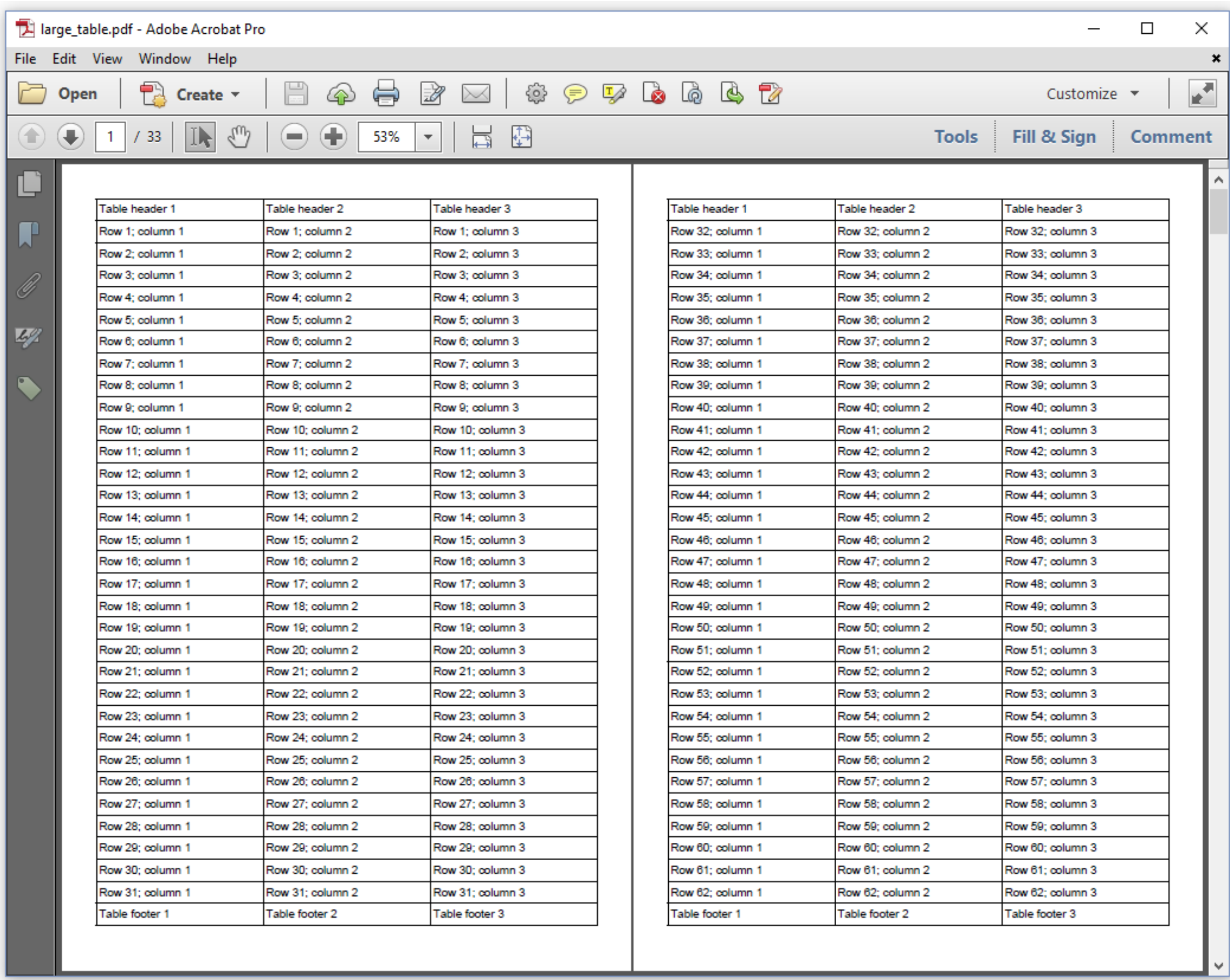
Once more, we create a constructor (line 3-7) and we override the `getNextRenderer()` method (line 8-12). We store the run length of the video in a `runlength` variable (line 2). We override the `drawBackground()` method and we draw the background using the appropriate size and color depending on the value of the `runlength` variable (line 13-32).

[87] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2044-c05e14_jekyllhydetablev7.java

We'll conclude this example with a trick to keep the memory use low when creating and adding tables to a document.

# Tables and memory use

Figure 5.20 shows a table that spans 33 pages. It has three columns and a thousand rows.




| Table header 1 | Table header 2 | Table header 3 |
|---|---|---|
| Row 1; column 1 | Row 1; column 2 | Row 1; column 3 |
| Row 2; column 1 | Row 2; column 2 | Row 2; column 3 |
| Row 3; column 1 | Row 3; column 2 | Row 3; column 3 |
| Row 4; column 1 | Row 4; column 2 | Row 4; column 3 |
| Row 5; column 1 | Row 5; column 2 | Row 5; column 3 |
| Row 6; column 1 | Row 6; column 2 | Row 6; column 3 |
| Row 7; column 1 | Row 7; column 2 | Row 7; column 3 |
| Row 8; column 1 | Row 8; column 2 | Row 8; column 3 |
| Row 9; column 1 | Row 9; column 2 | Row 9; column 3 |
| Row 10; column 1 | Row 10; column 2 | Row 10; column 3 |
| Row 11; column 1 | Row 11; column 2 | Row 11; column 3 |
| Row 12; column 1 | Row 12; column 2 | Row 12; column 3 |
| Row 13; column 1 | Row 13; column 2 | Row 13; column 3 |
| Row 14; column 1 | Row 14; column 2 | Row 14; column 3 |
| Row 15; column 1 | Row 15; column 2 | Row 15; column 3 |
| Row 16; column 1 | Row 16; column 2 | Row 16; column 3 |
| Row 17; column 1 | Row 17; column 2 | Row 17; column 3 |
| Row 18; column 1 | Row 18; column 2 | Row 18; column 3 |
| Row 19; column 1 | Row 19; column 2 | Row 19; column 3 |
| Row 20; column 1 | Row 20; column 2 | Row 20; column 3 |
| Row 21; column 1 | Row 21; column 2 | Row 21; column 3 |
| Row 22; column 1 | Row 22; column 2 | Row 22; column 3 |
| Row 23; column 1 | Row 23; column 2 | Row 23; column 3 |
| Row 24; column 1 | Row 24; column 2 | Row 24; column 3 |
| Row 25; column 1 | Row 25; column 2 | Row 25; column 3 |
| Row 26; column 1 | Row 26; column 2 | Row 26; column 3 |
| Row 27; column 1 | Row 27; column 2 | Row 27; column 3 |
| Row 28; column 1 | Row 28; column 2 | Row 28; column 3 |
| Row 29; column 1 | Row 29; column 2 | Row 29; column 3 |
| Row 30; column 1 | Row 30; column 2 | Row 30; column 3 |
| Row 31; column 1 | Row 31; column 2 | Row 31; column 3 |
| Table footer 1 | Table footer 2 | Table footer 3 |

| Table header 1 | Table header 2 | Table header 3 |
|---|---|---|
| Row 32; column 1 | Row 32; column 2 | Row 32; column 3 |
| Row 33; column 1 | Row 33; column 2 | Row 33; column 3 |
| Row 34; column 1 | Row 34; column 2 | Row 34; column 3 |
| Row 35; column 1 | Row 35; column 2 | Row 35; column 3 |
| Row 36; column 1 | Row 36; column 2 | Row 36; column 3 |
| Row 37; column 1 | Row 37; column 2 | Row 37; column 3 |
| Row 38; column 1 | Row 38; column 2 | Row 38; column 3 |
| Row 39; column 1 | Row 39; column 2 | Row 39; column 3 |
| Row 40; column 1 | Row 40; column 2 | Row 40; column 3 |
| Row 41; column 1 | Row 41; column 2 | Row 41; column 3 |
| Row 42; column 1 | Row 42; column 2 | Row 42; column 3 |
| Row 43; column 1 | Row 43; column 2 | Row 43; column 3 |
| Row 44; column 1 | Row 44; column 2 | Row 44; column 3 |
| Row 45; column 1 | Row 45; column 2 | Row 45; column 3 |
| Row 46; column 1 | Row 46; column 2 | Row 46; column 3 |
| Row 47; column 1 | Row 47; column 2 | Row 47; column 3 |
| Row 48; column 1 | Row 48; column 2 | Row 48; column 3 |
| Row 49; column 1 | Row 49; column 2 | Row 49; column 3 |
| Row 50; column 1 | Row 50; column 2 | Row 50; column 3 |
| Row 51; column 1 | Row 51; column 2 | Row 51; column 3 |
| Row 52; column 1 | Row 52; column 2 | Row 52; column 3 |
| Row 53; column 1 | Row 53; column 2 | Row 53; column 3 |
| Row 54; column 1 | Row 54; column 2 | Row 54; column 3 |
| Row 55; column 1 | Row 55; column 2 | Row 55; column 3 |
| Row 56; column 1 | Row 56; column 2 | Row 56; column 3 |
| Row 57; column 1 | Row 57; column 2 | Row 57; column 3 |
| Row 58; column 1 | Row 58; column 2 | Row 58; column 3 |
| Row 59; column 1 | Row 59; column 2 | Row 59; column 3 |
| Row 60; column 1 | Row 60; column 2 | Row 60; column 3 |
| Row 61; column 1 | Row 61; column 2 | Row 61; column 3 |
| Row 62; column 1 | Row 62; column 2 | Row 62; column 3 |
| Table footer 1 | Table footer 2 | Table footer 3 |

**Figure 5.20: working with large tables**

Suppose that we would create a `Table` object consisting of 3 header cells, 3 footer cells, and 3,000 normal cells, *before* adding this `Table` to a document. That would mean that at some point, we'd have 3,006 `Cell` objects in memory. That can easily lead to an `OutOfMemoryException` or an `OutOfMemoryError`. We can avoid this by adding the the table to the document while we are still adding content to the table. See the LargeTable[88] example.

[88] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-5#2825-c05e15_largetable.java

```
Table table = new Table(3, true);
table.addHeaderCell("Table header 1");
table.addHeaderCell("Table header 2");
table.addHeaderCell("Table header 3");
table.addFooterCell("Table footer 1");
table.addFooterCell("Table footer 2");
table.addFooterCell("Table footer 3");
document.add(table);
for (int i = 0; i < 1000; i++) {
    table.addCell(String.format("Row %s; column 1", i + 1));
    table.addCell(String.format("Row %s; column 2", i + 1));
    table.addCell(String.format("Row %s; column 3", i + 1));
    if (i %50 == 0) {
        table.flush();
    }
}
table.complete();
```

The `Table` class implements the `ILargeElement` interface. This interface defines methods such as `setDocument()`, `isComplete()` and `flushContent()` that are used internally by iText. When we use the `ILargeElement` interface in our code, we only need to use the `flush()` and `complete()` method.

We start by creating a `Table` for which we set the value of the `largeTable` parameter to `true` (line 1). We add the `Table` object to the `document` before we've completed adding content (line 8). As we marked the table as a large table, iText will use the `setDocument()` method internally so that the `table` and the `document` know of each other's existence. We add our 3,000 cells in a loop (line 9), but we `flush()` the content every 50 rows (line 13-15). When we flush the content, we already render part of the table. The `Cell` objects that were rendered are made available to the garbage collector so that the memory used by those objects can be released. Once we've added all the cells, we use the `complete()` method to write the remainder of the table that wasn't rendered yet, including the footer row.

This concludes the chapter about tables and cells.

# Summary

In this chapter, we've experimented with tables and cells. We talked about the dimensions and the alignment of tables, cells, and cell content. We learned about the difference between the margin and the spacing of a cell. We changed the borders of tables and cells using predefined `Border` objects and using a custom `CellRenderer` implementation. We nested tables, repeated headers and footers, changed the way tables are split when they don't fit a page. We extended the `TableRenderer` and the `CellRenderer` class to implement special features that aren't offered out-of-the-box. Finally, we learned how to reduce the memory use when creating and adding a `Table`.

We could stop here, because we've now covered every building block, but we'll add two more chapters to discuss some extra functionality that is useful when creating PDF documents using iText.

# Chapter 6: Creating actions, destinations, and bookmarks

When we discussed the `Link` building block in chapter 3, we created a URI action that opened a web page on IMDB when we clicked the text rendered by the `Link` object. We briefly mentioned that clickable areas are defined using *Link annotations*, and we referred to chapter 6 –this chapter– when we explained that `createURI()` created only one of many types of actions. In the examples that follow, we'll discover some more types, and we'll also learn about different types of destinations that can be used in a link. Finally, we'll also use those actions and destinations to create outlines, better known as bookmarks.

## URI actions

If you look at the `AbstractAction` class, you notice that it has a method named `secAction()`. When you use this method on a building block, you can define actions that will be triggered when clicking on its content. This is an alternative to using the `Link` object.

The `setAction()` method doesn't make sense for every building block. For instance: you can't click an `AreaBreak`. Please consult the appendix to find out for which objects the `setAction()` method can be used.

In figure 6.1, we see a PDF that is almost identical to the one we created in chapter 4 when we rendered the entries in our CSV file to a PDF with a numbered list.

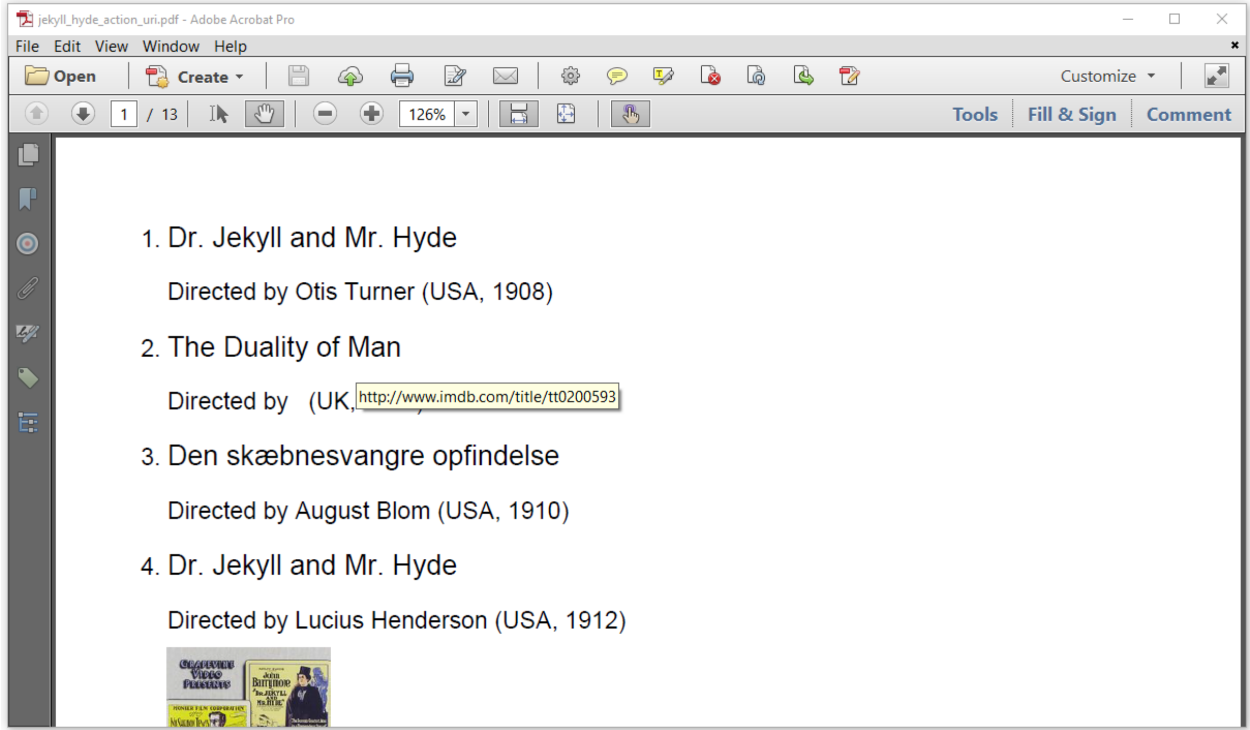



**Figure 6.1: using setAction() on a ListItem**

In the original example, we used a `Link` object so that you could jump to the corresponding IMDB page when clicking the title. In the URIAction[89] example, we make the complete `ListItem` clickable.

```
List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
resultSet.remove(0);
com.itextpdf.layout.element.List list =
    new com.itextpdf.layout.element.List(ListNumberingType.DECIMAL);
for (List<String> record : resultSet) {
    ListItem li = new ListItem();
    li.setKeepTogether(true);
    li.add(new Paragraph().setFontSize(14).add(record.get(2)))
        .add(new Paragraph(String.format(
            "Directed by %s (%s, %s)",
            record.get(3), record.get(4), record.get(1))));
    File file = new File(String.format(
        "src/main/resources/img/%s.jpg", record.get(0)));
    if (file.exists()) {
        Image img = new Image(ImageDataFactory.create(file.getPath()));
        img.scaleToFit(10000, 120);
        li.add(img);
```

[89] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2567-c06e01_uriaction.java

```
18      }
19      String url = String.format(
20          "http://www.imdb.com/title/tt%s", record.get(0));
21      li.setAction(PdfAction.createURI(url));
22      list.add(li);
23  }
24  document.add(list);
```

In line 21, we create a URI action using a link to IMDB and we set the action for the complete list item using the `setAction()` method.

# Named actions

Figure 6.2 shows links that are added to the first and the last page of a similar document. The link on the first page is marked "Go to last page"; the link on the last page is marked "Go to first page", and that's exactly what the links do when you click them.

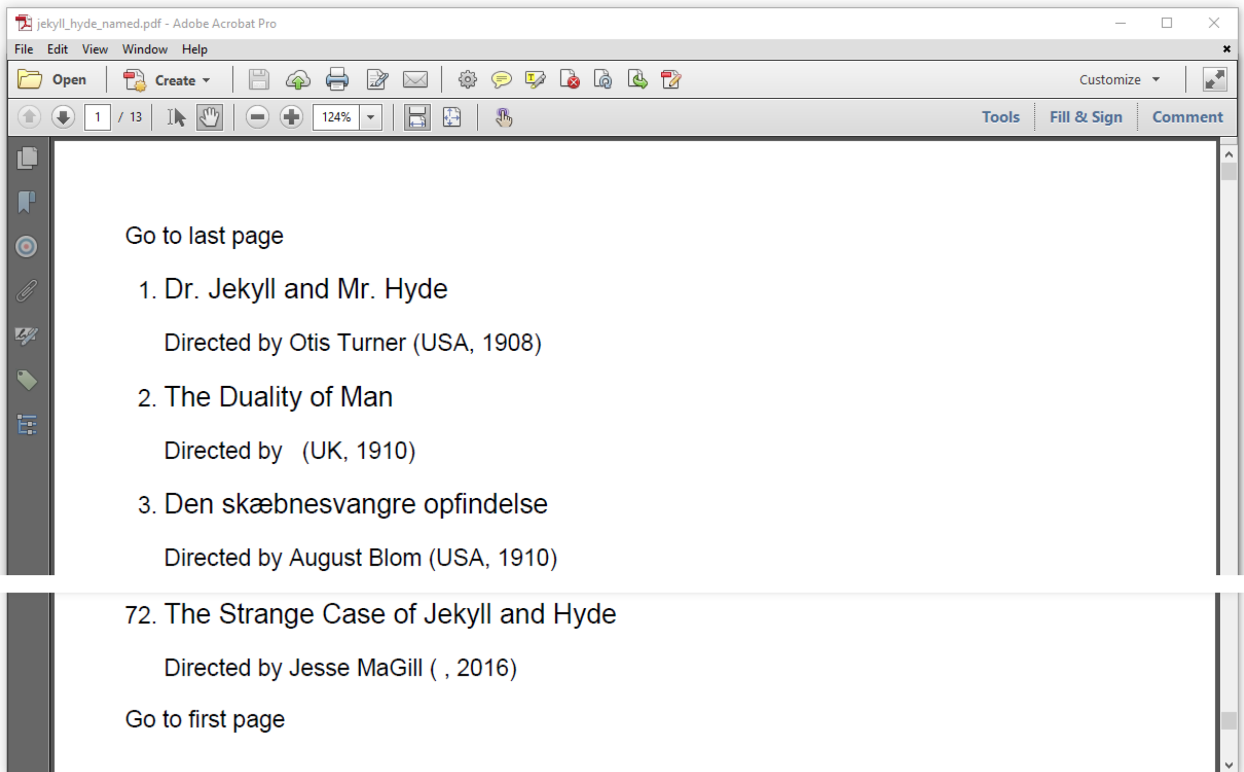



**Figure 6.2: Named actions**

We used named actions to achieve this; see the NamedAction[90] example.

[90] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2568-c06e02_namedaction.java

```
1 Paragraph p = new Paragraph()
2     .add("Go to last page")
3     .setAction(PdfAction.createNamed(PdfName.LastPage));
4 document.add(p);
5 p = new Paragraph()
6     .add("Go to first page")
7     .setAction(PdfAction.createNamed(PdfName.FirstPage));
8 document.add(p);
```

The `createNamed()` method accepts a `PdfName` as a parameter. You can use one of the following values:

- `PdfName.FirstPage`– the action allows you to jump to the first page of the document.
- `PdfName.PrevPage`– the action allows you to jump to the previous page in the document.
- `PdfName.NextPage`– the action allows you to jump to the next page in the document.
- `PdfName.LastPage`– the action allows you to jump to the last page of the document.

You could create these names yourself, for instance `new PdfName("PrevPage")`, but it's always better to use the names that are predefined in the `PdfName` class.

iText won't check if you pass a parameter that corresponds to one of these four values, because a PDF viewer may support additional, non-standard named actions. However, any document using such a non-standard action isn't portable.

These named actions allow us to navigate through a document, but they are rather limited, aren't they? If we want to create a table of contents that allows us to jump to a specific page, we need a GoTo action.

# GoTo actions

Figure 6.3 shows the table of contents of the Jekyll and Hyde story. If we'd click on a line, we'd jump to the corresponding page.

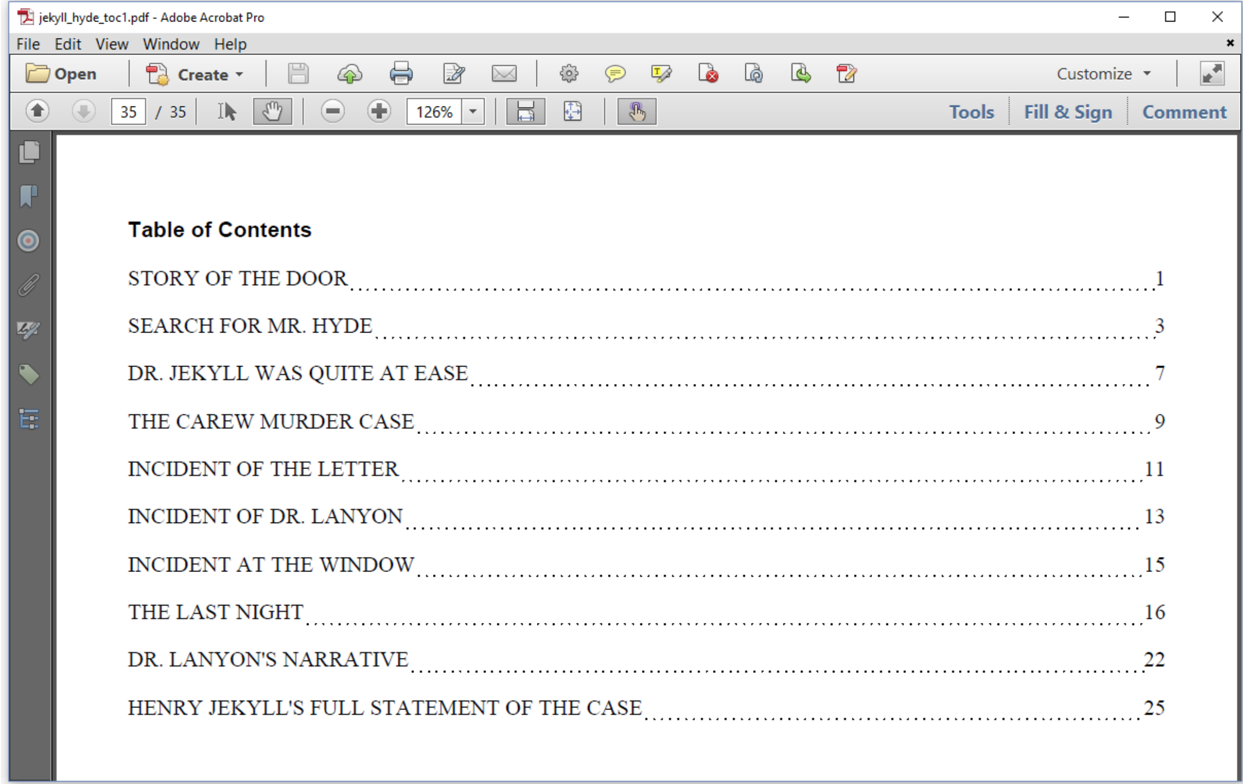



**Figure 6.3: A clickable table of contents**

To achieve this, we keep track of the titles and the page numbers on which these titles appear. The TOC_GoToPage[91] example shows how.

```
1  BufferedReader br = new BufferedReader(new FileReader(SRC));
2  String name, line;
3  Paragraph p;
4  boolean title = true;
5  int counter = 0;
6  List<SimpleEntry<String, Integer>> toc = new ArrayList<>();
7  while ((line = br.readLine()) != null) {
8      p = new Paragraph(line);
9      p.setKeepTogether(true);
10     if (title) {
11         name = String.format("title%02d", counter++);
12         p.setFont(bold).setFontSize(12)
13             .setKeepWithNext(true)
14             .setDestination(name);
15         title = false;
```

[91] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2569-c06e03_toc_gotopage.java

```
16          document.add(p);
17          toc.add(new SimpleEntry(line, pdf.getNumberOfPages()));
18      }
19      else {
20          p.setFirstLineIndent(36);
21          if (line.isEmpty()) {
22              p.setMarginBottom(12);
23              title = true;
24          }
25          else {
26              p.setMarginBottom(0);
27          }
28          document.add(p);
29      }
30  }
31  document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
32  p = new Paragraph().setFont(bold).add("Table of Contents");
33  document.add(p);
34  toc.remove(0);
35  List<TabStop> tabstops = new ArrayList();
36  tabstops.add(new TabStop(580, TabAlignment.RIGHT, new DottedLine()));
37  for (SimpleEntry<String, Integer> entry : toc) {
38      p = new Paragraph()
39          .addTabStops(tabstops)
40          .add(entry.getKey())
41          .add(new Tab())
42          .add(String.valueOf(entry.getValue()))
43          .setAction(PdfAction.createGoTo(
44                  PdfExplicitDestination.createFit(entry.getValue())));
45      document.add(p);
46  }
```

Most of the code repeats what we've done before to render the TXT file to a PDF, but these are the new lines that interest us the most:

- Line 6: we create an `ArrayList` named `toc` that will contain a series of `SimpleEntry` key-value pair entries. The key is a `String` we'll use for the title. They value is an `Integer` we'll use for the page number.
- Line 17: each time we add a title to the document (line 10-19), we add a new `SimpleEntry` to the `toc` list. We get the current page number using the `getNumberOfPage()` method.
- Line 31-33: once the full text is added, we go to a new page. We add a `Paragraph` saying `"Table of Contents"`.

- Line 34: we remove the first entry of the list, because that's the title of the book, not the title of a chapter.
- Line 35-36: we create a list of `TabStop` elements. We use a `DottedLine` as the tab leader.
- Line 37-46: we loop over all the entries in our `toc`. We use the key of each entry as well as the corresponding value to construct a `Paragraph` with the title and the page number as content. We also use the page number to create a GoTo action that jumps to that specific page.

In line 43, we use the `createGoTo()` method with a `PdfExplicitDestination` object as a parameter. The `PdfExplicitDestination` class extends the `PdfDestination` class. We'll take a closer look at these classes later on in this chapter. What's more important right now, is that there are two problems with this example, one problem is worse than the other.

1. The link jumps to another page in the document and shows this page in full. A more elegant solution would be to jump to the start of the actual title. We could use a different `PdfExplicitDestination` to achieve this (for instance `createFitH()` instead of `createFit()`).
2. The link doesn't always jump to the correct page. We store the page number of the last page in the document at the moment we add the title. That's the page number of the current page. However, we're also using the `setKeepWithNext()` method. This method forwards the title to a new page if the first paragraph of the chapter doesn't fit the current page. In that case, our TOC points at the wrong page, more specifically at the page just *before* the one we need.

We'll fix these two problems in the next example. Instead of an explicit destination, we'll use named destinations for a change.

## Named destinations

Figure 6.4 looks almost identical to figure 6.3. The fact that the page numbers are now correct is the only visible difference.

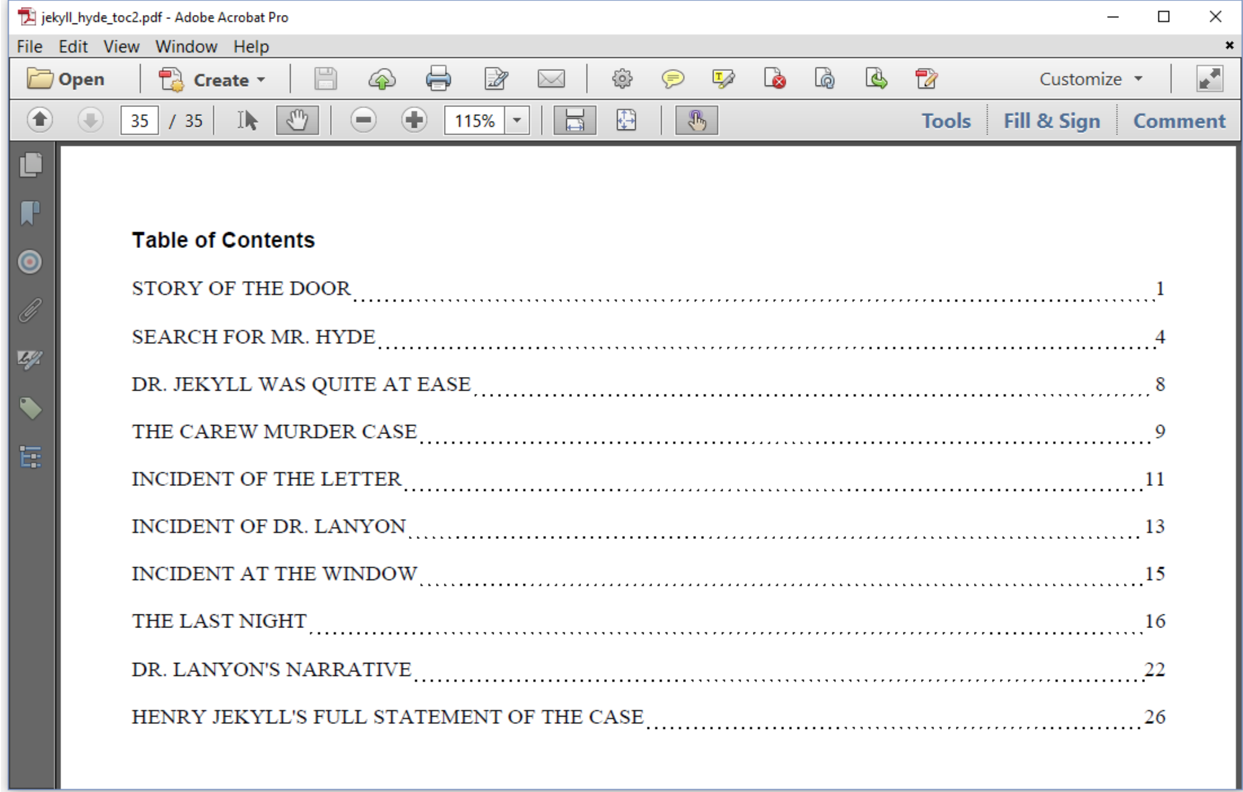



**Figure 6.4: A clickable table of contents**

The other difference is that we now used named destinations. We create those destinations by using the `setDestination()` method. This method is defined in the `ElementPropertyContainer` and can be used on many building blocks (see appendix). In the TOC_GoToNamed[92] example, we use it on a `Paragraph`.

```
1  BufferedReader br = new BufferedReader(new FileReader(SRC));
2  String name, line;
3  Paragraph p;
4  boolean title = true;
5  int counter = 0;
6  List<SimpleEntry<String,SimpleEntry<String, Integer>>> toc = new ArrayList<>();
7  while ((line = br.readLine()) != null) {
8      p = new Paragraph(line);
9      p.setKeepTogether(true);
10     if (title) {
11         name = String.format("title%02d", counter++);
12         SimpleEntry<String, Integer> titlePage
13                 = new SimpleEntry(line, pdf.getNumberOfPages());
```

[92] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2570-c06e04_toc_gotonamed.java

```
14            p.setFont(bold).setFontSize(12)
15                .setKeepWithNext(true)
16                .setDestination(name)
17                .setNextRenderer(new UpdatePageRenderer(p, titlePage));
18            title = false;
19            document.add(p);
20            toc.add(new SimpleEntry(name, titlePage));
21        }
22        else {
23            p.setFirstLineIndent(36);
24            if (line.isEmpty()) {
25                p.setMarginBottom(12);
26                title = true;
27            }
28            else {
29                p.setMarginBottom(0);
30            }
31            document.add(p);
32        }
33    }
34    document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
35    p = new Paragraph().setFont(bold)
36        .add("Table of Contents").setDestination("toc");
37    document.add(p);
38    toc.remove(0);
39    List<TabStop> tabstops = new ArrayList();
40    tabstops.add(new TabStop(580, TabAlignment.RIGHT, new DottedLine()));
41    for (SimpleEntry<String, SimpleEntry<String, Integer>> entry : toc) {
42        SimpleEntry<String, Integer> text = entry.getValue();
43        p = new Paragraph()
44            .addTabStops(tabstops)
45            .add(text.getKey())
46            .add(new Tab())
47            .add(String.valueOf(text.getValue()))
48            .setAction(PdfAction.createGoTo(entry.getKey()));
49        document.add(p);
50    }
```

Let's examine what is so different about this example when compared to the previous one.

- Line 6: we create an `ArrayList` named `toc` that will contain a series of `SimpleEntry` key-value pair entries. The key is a `String` that we'll use for a unique name. The value is no longer a

page number, but another `SimpleEntry`. The key of this second key-value pair will be the title of the chapter; the value will be the corresponding page number.

- Line 11: we create a unique name for every title: `title00`, `title01`, `title03`, and so on.
- Line 12-13: we create a `SimpleEntry` named `titlePage` using the title as a key and the current page number as the value. We know that this page number will be wrong in some cases. We will use a custom `ParagraphRenderer` to update the page number.
- Line 16: we use the unique name as a destination for the `Paragraph` using the `setDestination()` method.
- Line 17: we create an `UpdatePageRenderer` that will serve as the renderer for the title paragraph. We pass the `titlePage` entry as a parameter so that the renderer can update the page number.
- Line 20: we add a new `SimpleEntry` instance to the `toc` object. This entry contains the unique name and another entry with the title and the page number.
- Line 34-37: once the full text is added, we go to a new page. We add a `Paragraph` saying `"Table of Contents"`. Note that we define a destination named `"toc"` for that paragraph (line 36).
- Line 38: we remove the first entry of the list, because that's the title of the book, not the title of a chapter.
- Line 39-40: we create a list of `TabStop` elements. We use a `DottedLine` as the tab leader.
- Line 41-50: we loop over all the entries in our `toc`. We get the value of each entry (line 42) to construct the content of each line in the table of contents: the title (line 45) and the page number (line 47). We make the line clickable by adding a GoTo action that jumps to a location in the document based on a name.

Summarized: we mark a building block using a unique name. Internally, iText will map that name with a specific position –aka an explicit destination– in the document. Because of this, you can use the `createGoTo()` method passing that name as a parameter to create a link to that specific building block. We will even be able to use that name outside of the PDF document, but let's take a look at the `UpdatePageRenderer` before we do so.

```
1  protected class UpdatePageRenderer extends ParagraphRenderer {
2      protected SimpleEntry<String, Integer> entry;
3      public UpdatePageRenderer(
4          Paragraph modelElement, SimpleEntry<String, Integer> entry) {
5          super(modelElement);
6          this.entry = entry;
7      }
8      @Override
9      public LayoutResult layout(LayoutContext layoutContext) {
10         LayoutResult result = super.layout(layoutContext);
11         entry.setValue(layoutContext.getArea().getPageNumber());
12         return result;
```

```
13     }
14 }
```

The `entry` object contains a title and a page number. That page number could be wrong if the title is moved to the next page. We can only know if that happens when the title paragraph is rendered. Only at that moment, a layout decision will be made. The easiest way to update the page number in the `entry` object, is to override the `layout()` method as is done in line 11.

## Remote GoTo actions

Figure 6.5 is a PDF with two links marked in blue. When we click on the first link, the PDF we created in the previous example is opened on the first page in a new viewer window. When we click on the second link, the same document is opened on the table of contents page in the current window, replacing the document with the two links.

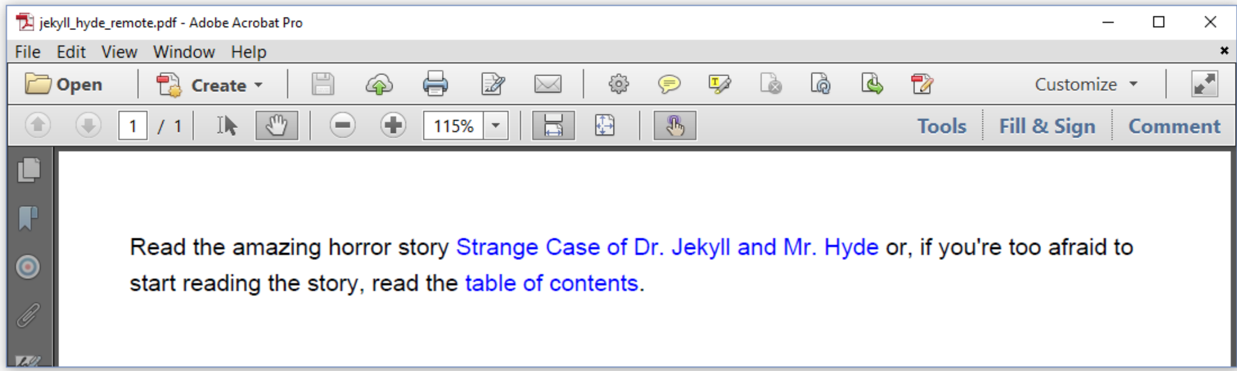



**Figure 6.5: Links to named destinations in another PDF document**

We use two `Link` objects to achieve this in the RemoteGoto[93] example.

```
1  Link link1 = new Link("Strange Case of Dr. Jekyll and Mr. Hyde",
2      PdfAction.createGoToR(
3          new File(TOC_GoToNamed.DEST).getName(), 1, true));
4  Link link2 = new Link("table of contents",
5      PdfAction.createGoToR(
6          new File(TOC_GoToNamed.DEST).getName(), "toc", false));
7  Paragraph p = new Paragraph()
8      .add("Read the amazing horror story ")
9      .add(link1.setFontColor(Color.BLUE))
10     .add(" or, if you're too afraid to start reading the story, read the ")
11     .add(link2.setFontColor(Color.BLUE))
```

[93]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2571-c06e05_remotegoto.java

```
12        .add(".");
13    document.add(p);
```

In line 2 and 3, we use the `createGoToR()` method to create a link to a remote PDF document.

- The first parameter is the name of the file we created in the previous example. We expect it to be in the same directory as the file we refer from.
- The second parameter is the page number; we want the link to jump to the first page.
- The third parameter indicates that we want to open the document in a new PDF viewer window.

In line 5 and 6, we use another `createGoToR()` method to create a link to a named destination in another document.

- The first parameter is the name of the file we created in the previous example.
- The second parameter is the name we used when we added the paragraph `"Table of Contents"`.
- The third parameter indicates that we want to open the document in the current PDF viewer window.

There are many other variations of the `createGoToR()` method, but they are all similar to one of the two methods that were just explained.

### How can I create a link that opens a PDF in a new browser window or tab?

There's a short answer to this question: you can't open a PDF in a new browser window using PDF syntax.

It is a common misconception that the boolean parameter indicating whether or not the PDF should be opened in the current window or in a new window, can also be used in the context of a browser. This isn't the case. There is a clear separation between the PDF viewer and the browser. The PDF viewer is usually a closed container that doesn't have access to the browser functionality. You shouldn't expect the PDF syntax to have the same capabilities as HTML. Those are two separate technologies.

Talking about HTML: you can use JavaScript in a PDF file that is very similar to the JavaScript you'd use in HTML. Many methods, such as methods that communicate with a server, are restricted, but you also have some extra methods that are specific to PDF. For instance: the JavaScript inside a PDF file has access to an `app` object that offers some functionality to communicate with the PDF viewer.

# JavaScript actions

We won't go into detail regarding the JavaScript functionality in PDF, but we'll create a simple PDF that shows an alert when you click a link; see figure 6.6.

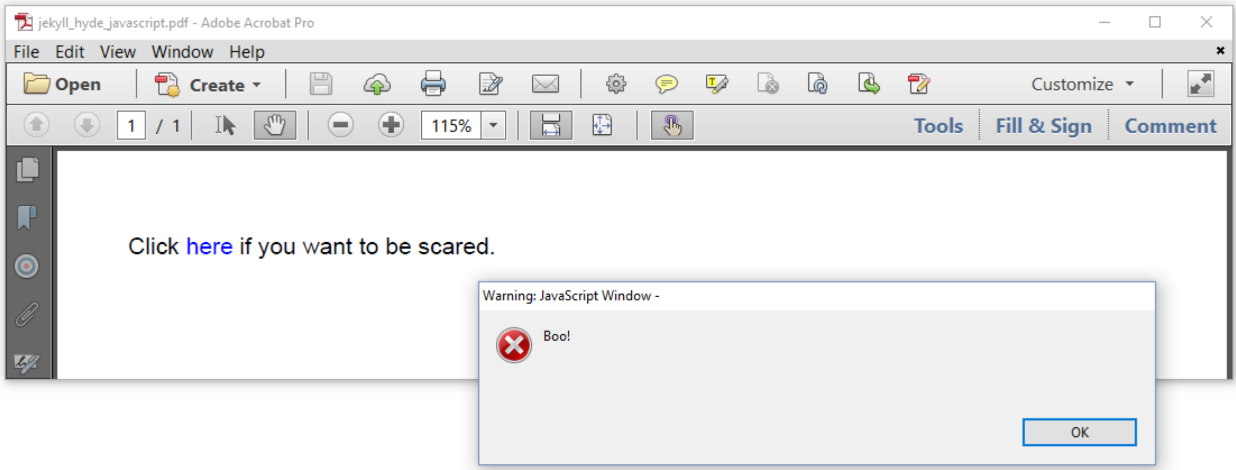



**Figure 6.6: A PDF with a JavaScript action**

We create the `Link` that allows us to trigger this alert in the JavaScript[94] example.

```
1 Link link = new Link("here",
2     PdfAction.createJavaScript("app.alert('Boo!');"));
3 Paragraph p = new Paragraph()
4     .add("Click ")
5     .add(link.setFontColor(Color.BLUE))
6     .add(" if you want to be scared.");
7 document.add(p);
```

In the next example, we'll use the same action, and we'll make it follow by another action.

# Chained actions

We've already used several `create()` convenience methods in the `PdfAction` class; we've experimented with `createURI()`, `createGoTo()`, `createGoToR()` and so on. If you consult the API documentation for the PdfAction[95] class, you'll find many more, such as `createGoToE()` to go to an embedded PDF file, `createLaunch()` to launch an application. All of these other methods are out of scope in the context of this tutorial, but we'll look at one more action example, the ChainedActions[96] example. It explains how to *chain* actions.

[94] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2572-c06e06_javascript.java
[95] http://itextsupport.com/apidocs/itext7/latest/com/itextpdf/kernel/pdf/action/PdfAction.html
[96] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2573-c06e07_chainedactions.java

```
1 PdfAction action = PdfAction.createJavaScript("app.alert('Boo');");
2 action.next(PdfAction.createGoToR(
3         new File(C06E04_TOC_GoToNamed.DEST).getName(), 1, true));
4 Link link = new Link("here", action);
5 Paragraph p = new Paragraph()
6     .add("Click ")
7     .add(link.setFontColor(Color.BLUE))
8     .add(" if you want to be scared.");
9 document.add(p);
```

In line 1, we create the same JavaScript action as in the previous example. We chain a remote GoTo action to this JavaScript action using the `next()` method in line 2. Now when we click the word `"here"`, a `Boo` alert will be triggered first; then another PDF will open in a new window.

The `createSubmitForm()` method is one of the many `PdfAction` methods we didn't discuss. We mention it here because of a common use case for the `next()` method. It is not unusual to validate fields that were filled in manually before submitting a form. This validation could be done using JavaScript. The submit action could be the last action in a validation chain.

While we were talking about actions, we mentioned the concept of destinations a couple of times. We also explained that links are actually annotations. In the next couple of examples, we'll spend some more time on these concepts.

# Destinations

The `PdfDestination` class is the abstract superclass of the `PdfExplicitDestination`, the `PdfStringDestination`, and the `PdfNamedDestination` class. The `PdfExplicitDestination` class can be used to create a destination to a specific page, using specific coordinates if needed. `PdfStringDestination` and `PdfNamedDestination` can be used to create a named destination.

### What's the difference between PdfStringDestination and PdfNamedDestination?

That's a great question, but the answer might require being read more than once. Both `PdfStringDestination` and `PdfNamedDestination` can be used to create a named destination, but:

- When we use the `PdfNamedDestination` class, the name will be stored inside the PDF document as a PDF name object. This is how named destinations were originally stored in PDF 1.1.
- When we use the `PdfStringDestination` class, the name will be stored as a PDF string object. This was introduced in PDF 1.2, because a PDF string object offers more possibilities than a name object.

Today, the name of a named destination should be stored as a PDF string, not as a PDF name. The `PdfNamedDestination` class is offered should you need it, but it is recommended that you use the `PdfStringDestination` class.

Using a PDF string as name is also the default way used by iText when you use the `setDestination()` method. We'll discover another way to create named destinations once we discuss bookmarks, but first, we'll create a couple of explicit destinations in the ExplicitDestinations[97] example.

```
1  PdfDestination jekyll =
2      PdfExplicitDestination.createFitH(1, 416);
3  PdfDestination hyde =
4      PdfExplicitDestination.createXYZ(1, 150, 516, 2);
5  PdfDestination jekyll2 =
6      PdfExplicitDestination.createFitR(2, 50, 380, 130, 440);
7  document.add(new Paragraph()
8          .add(new Link("Link to Dr. Jekyll", jekyll)));
9  document.add(new Paragraph()
10         .add(new Link("Link to Mr. Hyde", hyde)));
11 document.add(new Paragraph()
12         .add(new Link("Link to Dr. Jekyll on page 2", jekyll2)));
13 document.add(new Paragraph()
14     .setFixedPosition(50, 400, 80)
15     .add("Dr. Jekyll"));
16 document.add(new Paragraph()
17     .setFixedPosition(150, 500, 80)
18     .add("Mr. Hyde"));
19 document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
```

[97] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2574-c06e08_explicitdestinations.java

```
20 document.add(new Paragraph()
21     .setFixedPosition(50, 400, 80)
22     .add("Dr. Jekyll on page 2"));
```

We create three different types of explicit destinations:

- Line 1-2: an explicit destination that will go to page 1, and fit that page horizontally at coordinate `y = 416`.
- Line 3-4: an explicit destination that will go to page 1, so that the top-left corner has the coordinate `x = 150; y = 516` and the zoom factor is set to 200%.
- Line 5-6: an explicit destination that will go to page 2, so that at least a rectangle is visible with `x = 50; y = 380` as the coordinate of the lower-left corner and `x = 130; y = 440` as the coordinate of the upper-right corner.

These links are added in lines 7-8, 9-10, and 11-12 respectively. We also add some text that marks the destinations:

- Line 13-15: some text at coordinate `x = 50; y = 400` which is right below the first explicit destination.
- Line 16-18: some text at coordinate `x = 150; y = 500`, which puts it in the top-left corner of the visible area when we go to the second explicit destination.
- Line 20-22: some text on the second page at coordinate `x = 50; y = 400`, which makes it fit inside the rectangle defined in the third explicit destination.

We've used three different methods to create an explicit destination. The following table lists all the methods that are available to create an explicit destination. The first parameter is always an `int` referring to a page number, or a `PdfPage` instance. The other parameters, if any, are all of type `float`.

| Method | Parameters | Description |
|---|---|---|
| `createFit()` | - | The page is displayed with its contents magnified just enough to fit the document window, both horizontally and vertically. |
| `createFitB()` | - | The page is displayed magnified just enough to fit the bounding box of the contents (the smallest rectangle enclosing all of its contents). |

| Method | Parameters | Description |
|---|---|---|
| `createFitH()` | `top` | The page is displayed so that the page fits within the document window horizontally (the entire width of the page is visible). The extra parameter specifies the vertical coordinate of the top edge of the page. |
| `createFitBH()` | `top` | This option is almost identical to `createFitH()`, but the with of the bounding box of the page is visible. This isn't necessarily the entire width of the page. |
| `createFitV()` | `left` | The page is displayed so that the page fits within the document window vertically (the entire height of the page is visible). The extra parameter specifies the horizontal coordinate of the left edge of the page. |
| `createFitBV()` | `left` | This option is almost identical to `createFitV()`, but the height of the bounding box of the page is visible. This isn't necessarily the entire height of the page. |
| `createXYZ()` | `left, top, zoom` | The `left` parameter defines an `x` coordinate; `top` defines a `y` coordinate; and `zoom` defines a zoom factor. If you want to keep the current `x` coordinate, the current `y` coordinate, or zoom factor, you can pass negative values or `0` for the corresponding parameter. |
| `createFitR()` | `left, bottom, right, top` | The parameters define a rectangle. The page is displayed with its contents magnified just enough to fit this rectangle. If the required zoom factors for the horizontal and the vertical magnification are different, the smaller of the two is used. |

So far, we've created `Link` objects either by passing a `PdfAction` object as a parameter, or a `PdfDestination`. Both these methods create a `PdfLinkAnnotation`. We could have created that `PdfLinkAnnotation` ourselves and we could have passed that annotation as a parameter. This allows us to add some extra flavor to the link.

# Link annotations

There are two links in the document shown in figure 6.7. One is underlined; the other is marked by a rectangle.

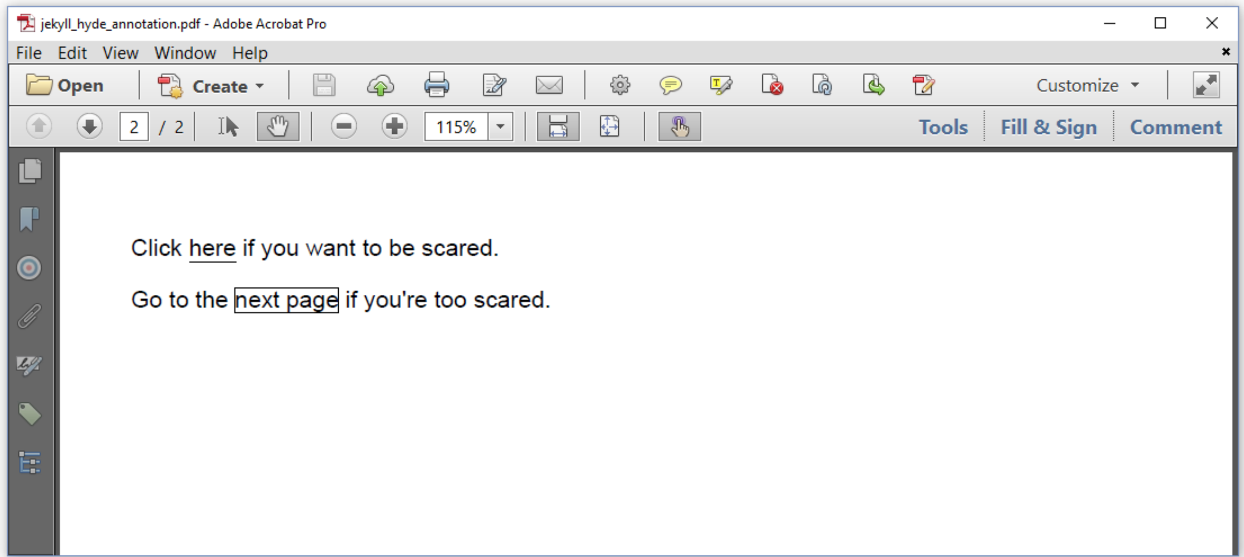



**Figure 6.7: Link annotations**

This line and rectangle shown in this screen shot are not part of the actual content of the PDF document. They weren't drawn using a sequence of `moveTo()`, `lineTo()`, and `stroke()` methods. They are part of the link annotation, and they are drawn by the PDF viewer that renders annotations on top of the existing content.

Also, when you would click the annotation, you would see a specific behavior. When clicking the first link, the colors would be inverted. When clicking the second link, you'd have a push-down effect. See the Annotation[98] example.

[98]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2575-c06e09_annotation.java

```
PdfAction js = PdfAction.createJavaScript("app.alert('Boo!');");
PdfAnnotation la1 = new PdfLinkAnnotation(new Rectangle(0, 0, 0, 0))
    .setHighlightMode(PdfAnnotation.HIGHLIGHT_INVERT)
    .setAction(js).setBorderStyle(PdfAnnotation.STYLE_UNDERLINE);
Link link1 = new Link("here", (PdfLinkAnnotation)la1);
document.add(new Paragraph()
    .add("Click ")
    .add(link1)
    .add(" if you want to be scared."));
PdfAnnotation la2 = new PdfLinkAnnotation(new Rectangle(0, 0, 0, 0))
    .setDestination(PdfExplicitDestination.createFit(2))
    .setHighlightMode(PdfAnnotation.HIGHLIGHT_PUSH)
    .setBorderStyle(PdfAnnotation.STYLE_INSET);
Link link2 = new Link("next page", (PdfLinkAnnotation)la2);
document.add(new Paragraph()
    .add("Go to the ")
    .add(link2)
    .add(" if you're too scared."));
```

We recognize the two links:

- We create a JavaScript action in line 1. We use this object as an action for the `PdfLinkAnnotation` we create in line 2. In line 2, we set the highlight mode to `HIGHLIGHT_INVERT`. This will invert the colors when we click the link. In line 4, we set the border style to `STYLE_UNDERLINE`. We use the `PdfLinkAnnotation` to create a `Link` object in line 5. We add a `Paragraph` with this link in lines 6 to 9.
- We create another `PdfLinkAnnotation` in line 10. This time we set a destination; see line 11. In line 12, we set the highlight mode to `HIGHLIGHT_PUSH` to get a push-down effect when we click the link. In line 13, we set the border style to `STYLE_INSET`. We create a `Link` with this `PdfLinkAnnotation` in line 14. We add another `Paragraph` in lines 15 to 18.

We could write a complete tutorial about annotations –and we will–, but whatever will be written in that tutorial is out of scope in this tutorial. We'll finish this chapter with a couple of bookmarks examples.

# Outlines aka bookmarks

We've already created a couple of documents that contained a table of contents. This table of contents was added as an extra page, listing the different chapters and the corresponding page numbers. When we clicked a line in this table of contents, we jumped to the corresponding chapter. In figure 6.8, we see a table of contents of a different nature. It's a table of contents that isn't printed when we print

the document. We only see it when we open the bookmarks panel in our PDF viewer, and we can use it to easily navigate the document by collapsing items in a tree structure.

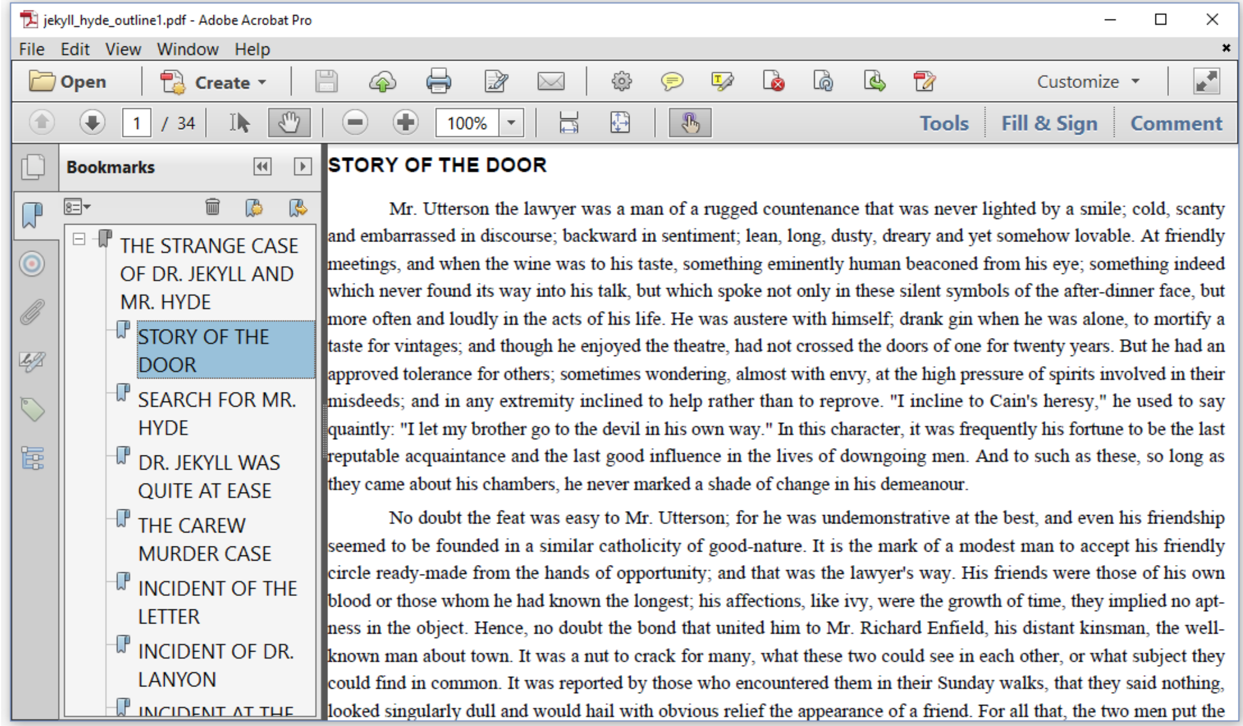



**Figure 6.8: Bookmarks using named destinations**

This tree structure is called an outline tree. Each branch and leaf of this tree is an outline object. In iText, we create these objects using the `PdfOutline` class. In the TOC_OutlinesNames[99] example, we use named destinations to jump to each chapter.

```
BufferedReader br = new BufferedReader(new FileReader(SRC));
String name, line;
Paragraph p;
boolean title = true;
int counter = 0;
PdfOutline outline = null;
while ((line = br.readLine()) != null) {
    p = new Paragraph(line);
    p.setKeepTogether(true);
    if (title) {
        name = String.format("title%02d", counter++);
        outline = createOutline(outline, pdf, line, name);
        p.setFont(bold).setFontSize(12)
```

[99] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2576-c06e10_toc_outlinesnames.java

```
14                 .setKeepWithNext(true)
15                 .setDestination(name);
16             title = false;
17             document.add(p);
18         }
19         else {
20             p.setFirstLineIndent(36);
21             if (line.isEmpty()) {
22                 p.setMarginBottom(12);
23                 title = true;
24             }
25             else {
26                 p.setMarginBottom(0);
27             }
28             document.add(p);
29         }
30     }
```

We initialize a `PdfOutline` object in line 6. We create a unique name for each chapter title in line 11. We use this name in line 15 as a destination, and pass it to the `createOutline()` method to create a `PdfOutline` that will link to the corresponding destination.

```
1  public PdfOutline createOutline(
2      PdfOutline outline, PdfDocument pdf, String title, String name) {
3      if (outline ==  null) {
4          outline = pdf.getOutlines(false);
5          outline = outline.addOutline(title);
6          outline.addDestination(
7              PdfDestination.makeDestination(new PdfString(name)));
8          return outline;
9      }
10     PdfOutline kid = outline.addOutline(title);
11     kid.addDestination(PdfDestination.makeDestination(new PdfString(name)));
12     return outline;
13 }
```

If the `outline` object passed to the `createOutline()` method is `null`, we're at the very beginning of our story. We get the root outline from the `PdfDocument` and we add an outline to this root object with the first title we encounter. This is the title of our novel "THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE". We want this `PdfOutline` to be the parent of all the other titles. We use the `makeDestination()` method using a `PdfString` object. This is equivalent to creating a `PdfStringDestination` using a `String` instance. We do more or less the same for the other titles.

When we create a destination using the `setDestination()` method, iText creates an XYZ destination using the top-left coordinate of the corresponding building block and a zoom factor of 100%. This creates the awkward effect that we no longer see the margin when we click on one of the bookmarks. We can fix this by creating explicit destinations. See figure 6.9.

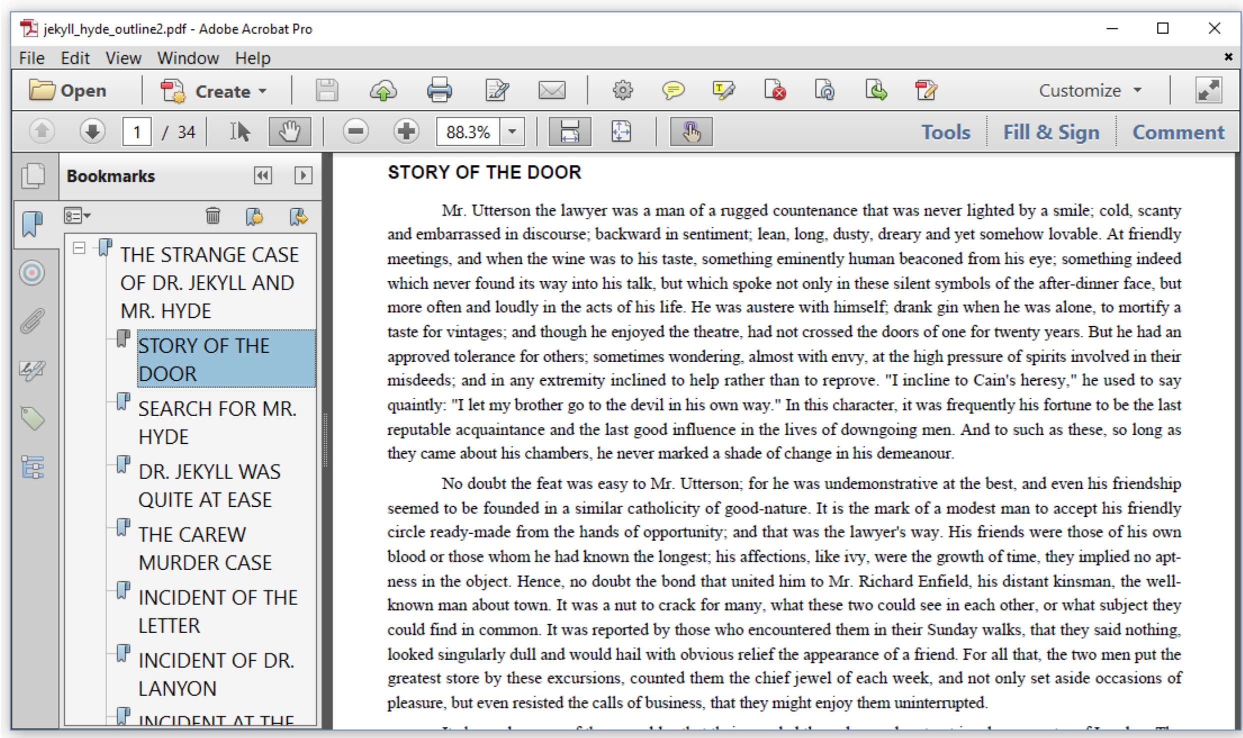



**Figure 6.9: Bookmarks using explicit destinations**

We remember from the previous table of contents example for which we used explicit destinations that it's easy to point to the wrong page. Once again, we'll use a renderer to make sure we link to the correct page. See the TOC_OutlinesDestinations[100] example.

```
1  BufferedReader br = new BufferedReader(new FileReader(SRC));
2  String line;
3  Paragraph p;
4  boolean title = true;
5  PdfOutline outline = null;
6  while ((line = br.readLine()) != null) {
7      p = new Paragraph(line);
8      p.setKeepTogether(true);
9      if (title) {
10         outline = createOutline(outline, pdf, line, p);
11         p.setFont(bold).setFontSize(12)
```

[100] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2577-c06e11_toc_outlinesdestinations.java

```
12             .setKeepWithNext(true);
13         title = false;
14         document.add(p);
15     }
16     else {
17         p.setFirstLineIndent(36);
18         if (line.isEmpty()) {
19             p.setMarginBottom(12);
20             title = true;
21         }
22         else {
23             p.setMarginBottom(0);
24         }
25         document.add(p);
26     }
27 }
```

This code snippet is shorter than the previous one because we don't have to create a name and we don't have to set that name as a destination. The main difference is in the `createOutline()` method. It now looks like this:

```
1  public PdfOutline createOutline(
2      PdfOutline outline, PdfDocument pdf, String title, Paragraph p) {
3      if (outline ==  null) {
4          outline = pdf.getOutlines(false);
5          outline = outline.addOutline(title);
6          return outline;
7      }
8      OutlineRenderer renderer = new OutlineRenderer(p, title, outline);
9      p.setNextRenderer(renderer);
10     return outline;
11 }
```

We use the first title we encounter (when `outline == null`) as the top-level outline in the outline tree. We create an `OutlineRenderer` to add the links to the kids of this top-level outline.

```
protected class OutlineRenderer extends ParagraphRenderer {
    protected PdfOutline parent;
    protected String title;
    public OutlineRenderer(
        Paragraph modelElement, String title, PdfOutline parent) {
        super(modelElement);
        this.title = title;
        this.parent = parent;
    }
    @Override
    public void draw(DrawContext drawContext) {
        super.draw(drawContext);
        Rectangle rect = getOccupiedAreaBBox();
        PdfDestination dest =
            PdfExplicitDestination.createFitH(
                drawContext.getDocument().getLastPage(),
                rect.getTop());
        PdfOutline outline = parent.addOutline(title);
        outline.addDestination(dest);
    }
}
```

In this case, we override the `draw()` method. We create a `PdfOutline` object with the top-level outline as parent (line 18), and we use the top `y` coordinate of the area occupied by the `Paragraph` as the `top` parameter for an explicit destination that fits the page horizontally (line 14-17) as the destination for that newly created outline (line 19).

If you study both examples carefully, you'll discover that the top-level outline of the example using named destination can be clicked to jump to the title of the novel. This isn't the case in the example in which we create explicit destinations: we only created destinations for the titles of the chapters, not for the title of the novel. The `PdfOutline` objects in an outline tree don't need to be real bookmarks. They don't have to point to a destination on a specific page in the document. They can point to nowhere; they can also be used to trigger an action. We'll make one more bookmark example to demonstrate this. Additionally, we'll change the color and style of the elements in the bookmarks panel.

## Color and style of the outline elements.

In figure 6.10, we have a PDF document with a single blank page.

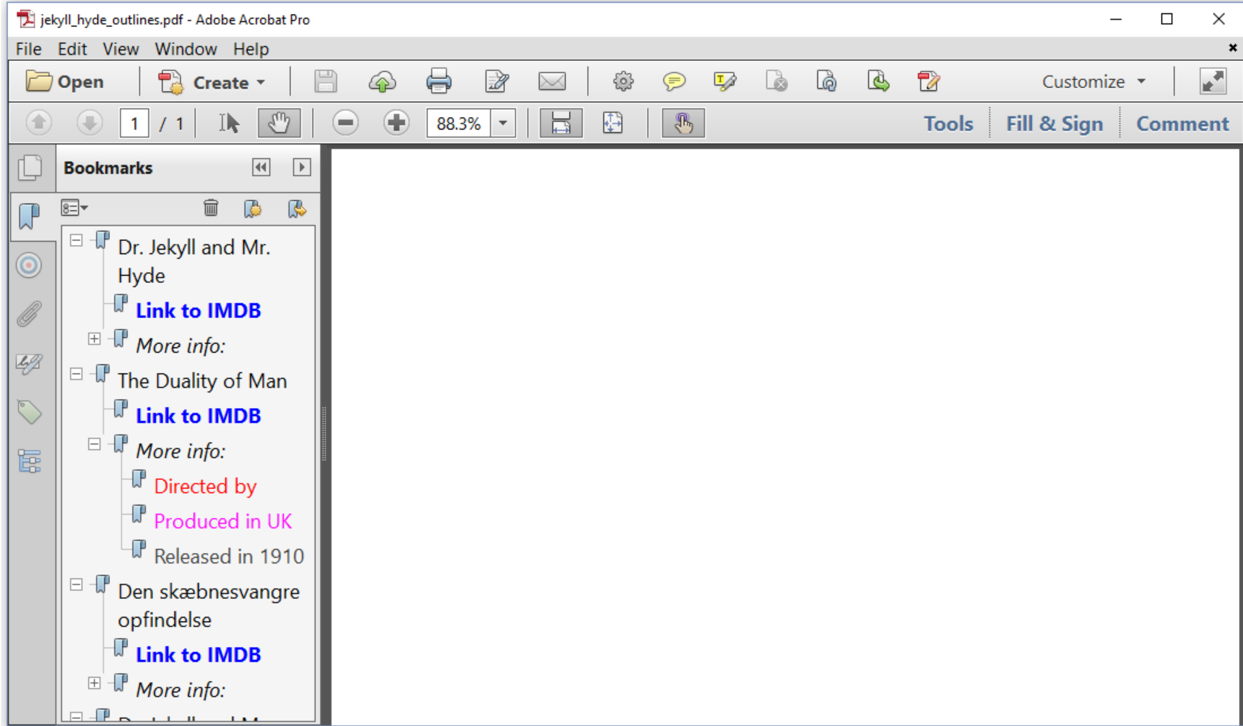



**Figure 6.10: Example of an outline tree without actual bookmarks**

When we open the bookmark panel, we see an outline tree of which all first-level elements are titles of a movie, cartoon or video. These outlines are the parent of two kids:

1. One shows "Link to IMDB" in bold and blue. When we click that outline, an URI action is triggered that brings us to the corresponding web page.
2. The other reads as "More info:" in italic. It is closed by default, but when we open it, we see information in different colors about the director, the country where the movie is produced, and its release data.

None of these `PdfOutline` objects point to a location in the document. The Outlines[101] example shows how this outline tree was built.

[101]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-6#2578-c06e12_outlines.java

```
1  public void createPdf(String dest) throws IOException {
2      PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
3      pdf.addNewPage();
4      pdf.getCatalog().setPageMode(PdfName.UseOutlines);
5      PdfOutline root = pdf.getOutlines(false);
6      List<List<String>> resultSet = CsvTo2DList.convert(SRC, "|");
7      resultSet.remove(0);
8      for (List<String> record : resultSet) {
9          PdfOutline movie = root.addOutline(record.get(2));
10         PdfOutline imdb = movie.addOutline("Link to IMDB");
11         imdb.setColor(Color.BLUE);
12         imdb.setStyle(PdfOutline.FLAG_BOLD);
13         String url = String.format(
14             "http://www.imdb.com/title/tt%s", record.get(0));
15         imdb.addAction(PdfAction.createURI(url));
16         PdfOutline info = movie.addOutline("More info:");
17         info.setOpen(false);
18         info.setStyle(PdfOutline.FLAG_ITALIC);
19         PdfOutline director = info.addOutline("Directed by " + record.get(3));
20         director.setColor(Color.RED);
21         PdfOutline place = info.addOutline("Produced in " + record.get(4));
22         place.setColor(Color.MAGENTA);
23         PdfOutline year = info.addOutline("Released in " + record.get(1));
24         year.setColor(Color.DARK_GRAY);
25     }
26     pdf.close();
27 }
```

Let's go through this code step by step:

- We create a `PdfDocument` (line 2) to which we add a single page (line 3). We change the page mode so that the bookmarks panel is opened by default (line 4). We'll learn more about page mode, layout mode and other viewer preferences in the next chapter.
- We get the root object of the outline tree (line 5). The `boolean` parameter indicates if iText needs to update the outlines. If `true`, the method will read the whole document and create the outline tree. This isn't necessary here, we can just get the cached outline tree. As we have just created the `PdfDocument` there aren't any outlines in that tree yet anyway.
- We create a list with all the records in our Jekyll and Hyde movie database (line 6) and we remove the record with the field names (line 7). We loop over the different records (line 8 - 25).
- Each movie gets its own outline containing its title (line 9).

- We add a first child outline with as title "Link to IMDB" (line 10). We change the color of this title to blue (line 11) and bold (line 12). We add a URI action that jumps to the movie page for that specific movie on IMDB (line 13-15).
- We add a second child outline with as title "More info:" (line 16). By default all the outlines we create are open; in this case, we want the outline to be closed (line 17). We change the style to italic (line 18). Finally we add three children to this outline: the director (line 19) as red text (line 20), the country (line 21) as magenta text (line 22), and the year (line 23) as dark gray text (line 24).
- Finally, we close the `PdfDocument` (line 26).

This example shows how you can easily create an outline tree with different branches, branches of branches, and leaves. It also shows how you can change the color and style of an element in the outline tree, and how you can change the open or closed status of each outline element.

# Summary

This chapter was all about interactive elements that help us navigate through and between documents. We started by experimenting with a series of actions:

- URI actions to navigate to external web pages,
- Named actions to navigate to the first page, previous page, next page, and last page,
- GoTo actions to go to a named destination or an explicit destination inside the document,
- Remote GoTo action to navigate to another PDF document in the same or in a new window,
- JavaScript actions to trigger the execution of PDF-specific JavaScript.

We took a close look at destinations, and how to create them using one of the subclasses of the abstract `PdfDestination` class.

After we learned that links are stored inside a PDF as *annotations*, we looked at some bookmark examples. We learned how to create an outline tree, and we used the `setDestination()` method to jump to a destination inside the document, the `setAction()` method to trigger an action, and none of these to create an inert hierarchical entry in the outline tree.

We already saw a glimpse of the next chapter, when we changed the page mode to make sure the bookmarks panel was opened when opening the document. Viewer preferences will be one of the topics we'll discuss next, but first we'll learn more about the concept of event handling.

# Chapter 7: Handling events; setting viewer preferences and writer properties

This book is meant for developers who want to create PDF documents from scratch in a programmatic way, using source code as opposed to using a template. We started with a chapter about fonts. In the chapters that followed, we discussed the default behavior of every element: `Paragraph`, `Text`, `Image`, and so on. We discovered that these elements can be used in a very intuitive way, but also that we can change their default behavior by creating custom renderer implementations –which isn't always trivial, depending on what you want to achieve. In the previous chapter, we discussed interactivity. We introduced actions and added links and bookmarks that help us navigate through a document.

We'll use this final chapter to introduce a couple of concepts that weren't discussed before. iText creates a new page automatically when elements don't fit the current page, but what if we want to add a watermark, background, header or footer to every page? How do we know when a new page is created? We'll need to look at the `IEventHandler` interface to find out. In the previous chapter, we changed a viewer preference so that the bookmarks panel is open by default. We'll look at some other viewer preferences that can be set. Finally, we'll learn how to change the settings of the `PdfWriter`, for instance to create a PDF of a version that is different from the default PDF version used by iText.

## Implementing the IEventHandler interface

In previous examples, we used the `rotate()` method to switch a page from portrait to landscape. For instance, when we created PDF with tables in chapter 5, we created our `Document` object like this `new Document(pdf, PageSize.A4.rotate())`. In figure 7.1, we also see pages that are rotated, but they are rotated with a different purpose. In chapter 5, we wanted to take advantage of the fact that the width of the page is greater than the height when using landscape orientation. When using the `rotate()` method, it was our purpose to rotate the page, but not its content as is done in figure 7.1.

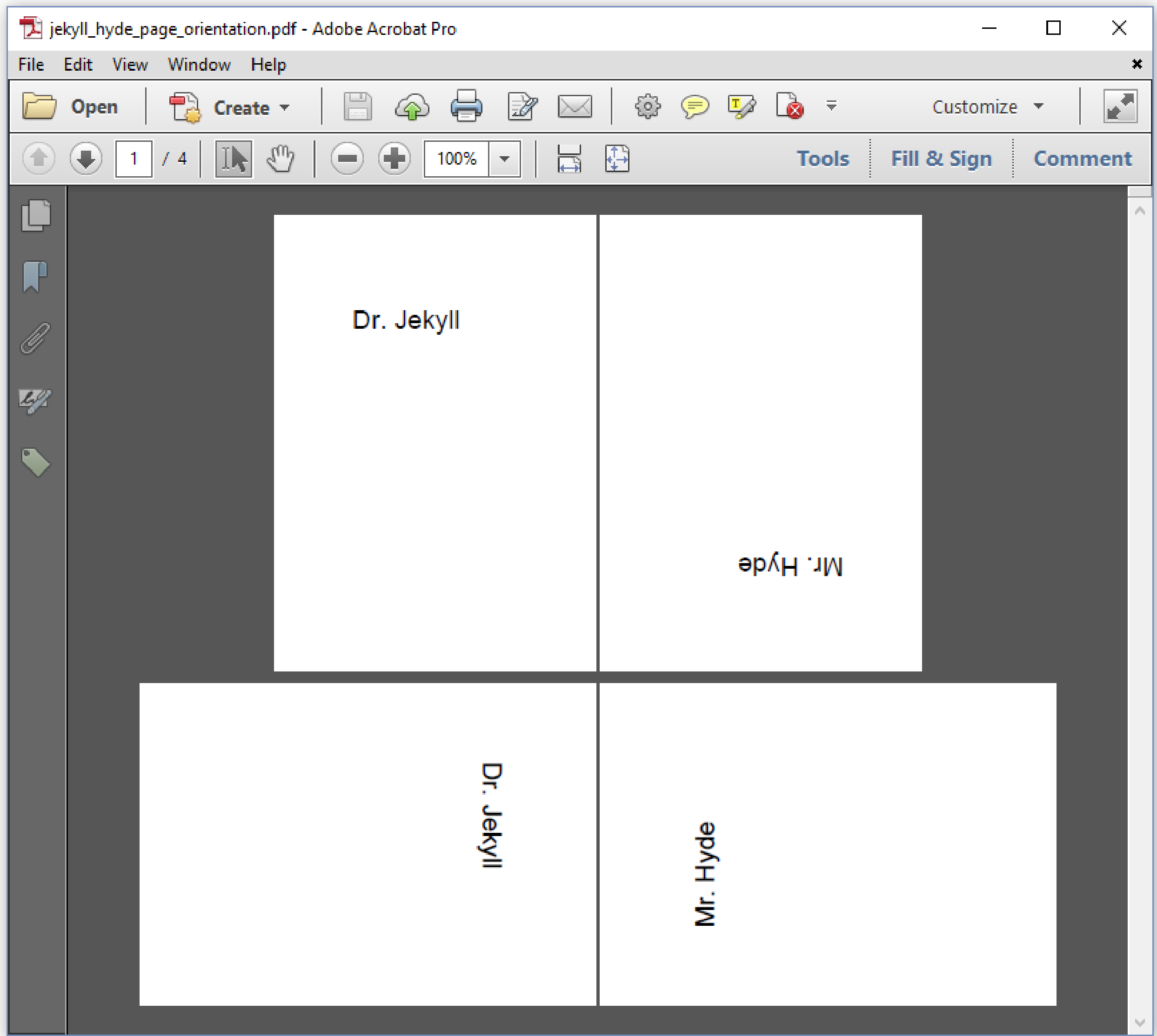



**Figure 7.1: Pages with different orientations**

In the EventHandlers[102] example, we create four A6 pages to which we add content as if the page is in portrait. We change the rotation of the page at the page level in an `IEventHandler`. As defined in the ISO standard for PDF, the rotation of a page needs to be a multiple of 90. This leaves us four possible orientations when we divide the rotation by 360: portrait (rotation 0), landscape (rotation 90), inverted portrait (rotation 180) and seascape (rotation 270).

[102] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2579-c07e01_eventhandlers.java

```
public static final PdfNumber PORTRAIT = new PdfNumber(0);
public static final PdfNumber LANDSCAPE = new PdfNumber(90);
public static final PdfNumber INVERTEDPORTRAIT = new PdfNumber(180);
public static final PdfNumber SEASCAPE = new PdfNumber(270);
```

We create a `PageRotationEventHandler` that allows us to change the rotation of a page, while we are creating a document.

```
protected class PageRotationEventHandler implements IEventHandler {
    protected PdfNumber rotation = PORTRAIT;
    public void setRotation(PdfNumber orientation) {
        this.rotation = orientation;
    }
    @Override
    public void handleEvent(Event event) {
        PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
        docEvent.getPage().put(PdfName.Rotate, rotation);
    }
}
```

The default orientation will be portrait (line 2), but we can change this default using the `setRotation()` method (line 4-6). We override the `handleEvent()` method that is triggered when an event occurs. We can get the `PdfPage` instance of the page on which the event is triggered from the `PdfDocumentEvent`. This `PdfPage` object represents the page dictionary. One of the possible entries of a page dictionary, is its rotation. We change this entry to the current value of `rotation` (line 9) every time the event is triggered.

The following snippet shows how we can introduce this event handler in the PDF creation process.

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
    pdf.getCatalog().setPageLayout(PdfName.TwoColumnLeft);
    PageRotationEventHandler eventHandler =
        new PageRotationEventHandler();
    pdf.addEventHandler(
        PdfDocumentEvent.START_PAGE, eventHandler);
    Document document = new Document(pdf, PageSize.A8);
    document.add(new Paragraph("Dr. Jekyll"));
    eventHandler.setRotation(INVERTEDPORTRAIT);
    document.add(new AreaBreak());
    document.add(new Paragraph("Mr. Hyde"));
    eventHandler.setRotation(LANDSCAPE);
```

```
14      document.add(new AreaBreak());
15      document.add(new Paragraph("Dr. Jekyll"));
16      eventHandler.setRotation(SEASCAPE);
17      document.add(new AreaBreak());
18      document.add(new Paragraph("Mr. Hyde"));
19      document.close();
20  }
```

We create an instance of the `PageRotationEventHandler` (line 4-5). We declare this `eventHandler` as an event that needs to be triggered every time a new page is started (`PdfDocumentEvent.START_PAGE`) in the `PdfDocument` (line 6-7). We create a PDF with tiny pages (line 8). We add a first paragraph (line 9) on a page that will use the default orientation. The `START_PAGE` event has already happened, when we change this default to inverted portrait (line 10). Only when a new page is created, after introducing a page break (line 11), the new orientation will become active. In this example, we repeat this a couple of times to demonstrate every possible page orientation.

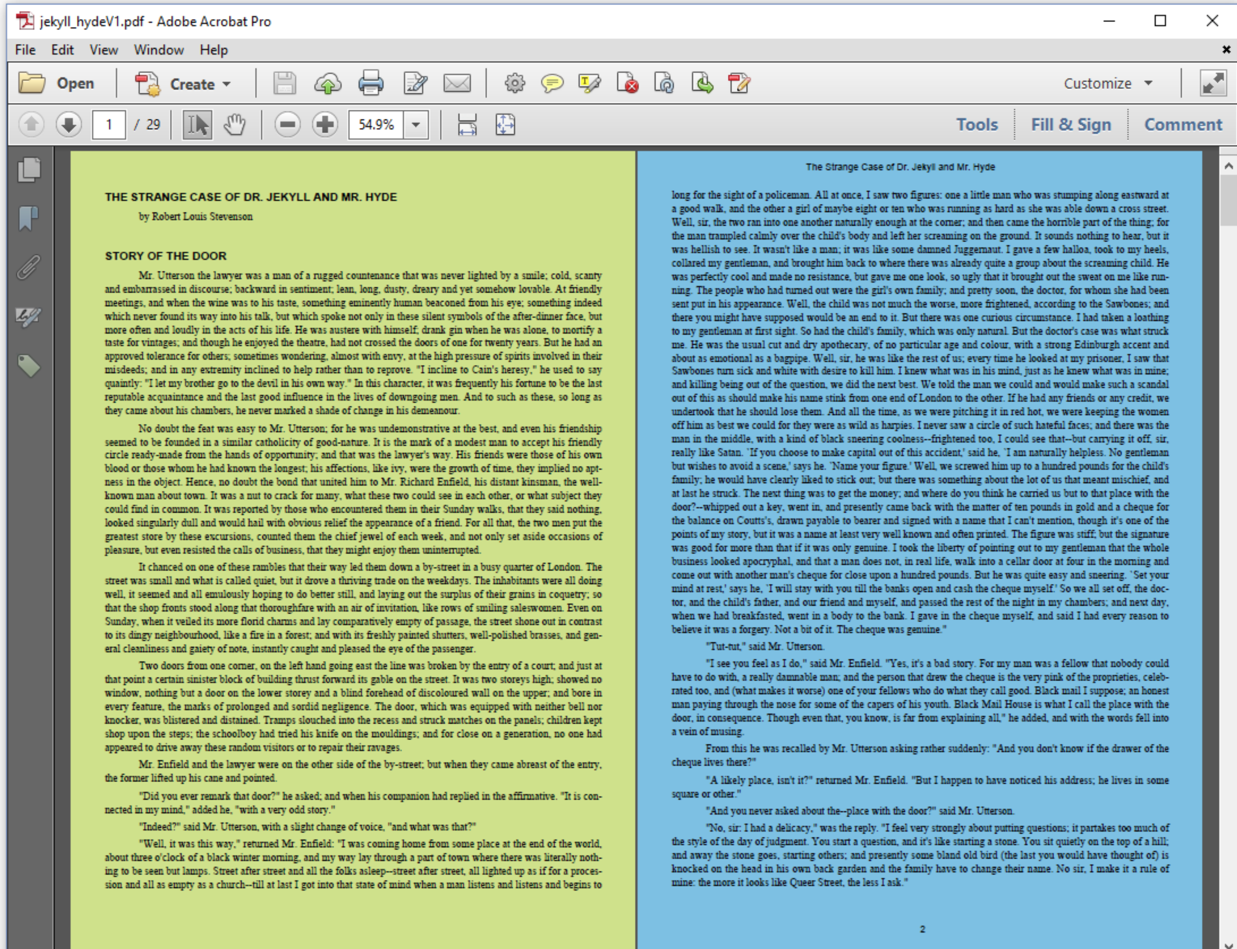
There are four types of events that can be triggered:

- `START_PAGE`– triggered when a new page is started,
- `END_PAGE`– triggered right before a new page is started,
- `INSERT_PAGE`– triggered when a page is inserted, and
- `REMOVE_PAGE`– triggered when a page is removed.

We'll try all of these types in the next handful of examples.

# Adding a background and text to every page

We have created many documents in which we rendered a novel by Robert Louis Stevenson to PDF. We reused the code of one of these examples to create the PDF shown in figure 7.2, and we introduced an event handler to create a lime-colored background for the odd pages and a blue-colored background for the even pages. Starting on page 2, we also added a running header with the title of the novel and a footer with the page number.



THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and laying out the surplus of their grains in coquetry; so that the shop fronts stood along that thoroughfare with an air of invitation, like rows of smiling saleswomen. Even on Sunday, when it veiled its more florid charms and lay comparatively empty of passage, the street shone out in contrast to its dingy neighbourhood, like a fire in a forest; and with its freshly painted shutters, well-polished brasses, and general cleanliness and gaiety of note, instantly caught and pleased the eye of the passenger.

Two doors from one corner, on the left hand going east the line was broken by the entry of a court; and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept shop upon the steps; the schoolboy had tried his knife on the mouldings; and for close on a generation, no one had appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to

The Strange Case of Dr. Jekyll and Mr. Hyde

long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the rest of us; every time he looked at my prisoner, I saw that Sawbones turn sick and white with desire to kill him. I knew what was in his mind, just as he knew what was in mine; and killing being out of the question, we did the next best. We told the man we could and would make such a scandal out of this as should make his name stink from one end of London to the other. If he had any friends or any credit, we undertook that he should lose them. And all the time, as we were pitching it in red hot, we were keeping the women off him as best we could for they were as wild as harpies. I never saw a circle of such hateful faces; and there was the man in the middle, with a kind of black sneering coolness--frightened too, I could see that--but carrying it off, sir, really like Satan. 'If you choose to make capital out of this accident,' said he, 'I am naturally helpless. No gentleman but wishes to avoid a scene,' says he. 'Name your figure.' Well, we screwed him up to a hundred pounds for the child's family; he would have clearly liked to stick out; but there was something about the lot of us that meant mischief, and at last he struck. The next thing was to get the money; and where do you think he carried us but to that place with the door?--whipped out a key, went in, and presently came back with the matter of ten pounds in gold and a cheque for the balance on Coutts's, drawn payable to bearer and signed with a name that I can't mention, though it's one of the points of my story, but it was a name at least very well known and often printed. The figure was stiff; but the signature was good for more than that if it was only genuine. I took the liberty of pointing out to my gentleman that the whole business looked apocryphal, and that a man does not, in real life, walk into a cellar door at four in the morning and come out with another man's cheque for close upon a hundred pounds. But he was quite easy and sneering. 'Set your mind at rest,' says he, 'I will stay with you till the banks open and cash the cheque myself.' So we all set off, the doctor, and the child's father, and our friend and myself, and passed the rest of the night in my chambers; and next day, when we had breakfasted, went in a body to the bank. I gave in the cheque myself, and said I had every reason to believe it was a forgery. Not a bit of it. The cheque was genuine."

"Tut-tut," said Mr. Utterson.

"I see you feel as I do," said Mr. Enfield. "Yes, it's a bad story. For my man was a fellow that nobody could have to do with, a really damnable man; and the person that drew the cheque is the very pink of the proprieties, celebrated too, and (what makes it worse) one of your fellows who do what they call good. Black mail I suppose; an honest man paying through the nose for some of the capers of his youth. Black Mail House is what I call the place with the door, in consequence. Though even that, you know, is far from explaining all," he added, and with the words fell into a vein of musing.

From this he was recalled by Mr. Utterson asking rather suddenly: "And you don't know if the drawer of the cheque lives there?"

"A likely place, isn't it?" returned Mr. Enfield. "But I happen to have noticed his address; he lives in some square or other."

"And you never asked about the--place with the door?" said Mr. Utterson.

"No, sir: I had a delicacy," was the reply. "I feel very strongly about putting questions; it partakes too much of the style of the day of judgment. You start a question, and it's like starting a stone. You sit quietly on the top of a hill; and away the stone goes, starting others; and presently some bland old bird (the last you would have thought of) is knocked on the head in his own back garden and the family have to change their name. No sir, I make it a rule of mine: the more it looks like Queer Street, the less I ask."

2

**Figure 7.2: Colored background and running header**

For this TextWatermark[103] example, we added an END_PAGE event for a change.

```
1 pdf.addEventHandler(
2     PdfDocumentEvent.END_PAGE,
3     new TextWatermark());
```

This choice for the END_PAGE event has an impact on the TextWatermark class.

[103] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2580-c07e02_textwatermark.java

```
protected class TextWatermark implements IEventHandler {
    Color lime, blue;
    PdfFont helvetica;
    protected TextWatermark() throws IOException {
        helvetica = PdfFontFactory.createFont(FontConstants.HELVETICA);
        lime = new DeviceCmyk(0.208f, 0, 0.584f, 0);
        blue = new DeviceCmyk(0.445f, 0.0546f, 0, 0.0667f);
    }
    @Override
    public void handleEvent(Event event) {
        PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
        PdfDocument pdf = docEvent.getDocument();
        PdfPage page = docEvent.getPage();
        int pageNumber = pdf.getPageNumber(page);
        Rectangle pageSize = page.getPageSize();
        PdfCanvas pdfCanvas = new PdfCanvas(
            page.newContentStreamBefore(), page.getResources(), pdf);
        pdfCanvas.saveState()
            .setFillColor(pageNumber % 2 == 1 ? lime : blue)
            .rectangle(pageSize.getLeft(), pageSize.getBottom(),
                pageSize.getWidth(), pageSize.getHeight())
            .fill().restoreState();
        if (pageNumber > 1) {
            pdfCanvas.beginText()
                .setFontAndSize(helvetica, 10)
                .moveText(pageSize.getWidth() / 2 - 120, pageSize.getTop() - 20)
                .showText("The Strange Case of Dr. Jekyll and Mr. Hyde")
                .moveText(120, -pageSize.getTop() + 40)
                .showText(String.valueOf(pageNumber))
                .endText();
        }
        pdfCanvas.release();
    }
}
```

We create color objects (line 2) and a font (line 3) in the constructor (line 4-8), so that we can reuse these objects every time the event is triggered.

The `PdfDocumentEvent` (line 11) gives us access to the `PdfDocument` (line 12) and the `PdfPage` (line 13) on which the event was triggered. We get the current page number (line 14) and the page size (line 15) from the `PdfPage`. In this example, we will add all the content using low-level PDF functionality. We need a `PdfCanvas` object to do this (line 16-17). We draw the background using a `rectangle()` and `fill()` method (line 18-22). For pages with page number greater than 1 (line 23), We create

a text object marked by `beginText()` and `endText()` with two snippets of text that are positioned using the `moveText()` method and added with `showText()` method (line 24-30).

As we add this content after the current page has been completed and right before a new page is created, we have to be careful not to overwrite already existing content. For instance: to create a colored background, we draw an opaque rectangle. If we do this *after* we have added content to the page, this content won't be visible anymore: it will be covered by the opaque rectangle. We can solve this by creating the `PdfCanvas` using the `page.newContentStreamBefore()` method. This will allow us to write PDF syntax to a content stream that will be parsed *before* the rest of the content of the page is parsed.

In iText 5, we used page events to add content when a specific event occurred. It was forbidden to add content in an `onStartPage()` event. One could only add content to a page using the `onEndPage()` method. This often led to confusion among developers who assumed that headers needed to be added in the `onStartPage()` method, whereas footers needed to be added in the `onEndPage()` method. Although this was a misconception, we fixed this problem anyway. Actually, in the case of this example, it would probably be a better idea to add the back ground in the `START_PAGE` event. We can use `page.getLastContentStream()` to create the content stream needed for the `PdfCanvas` object.

In the next example, we'll add a header using a `START_PAGE` event and a footer using an `END_PAGE` event. The footer will show the page number as well as the total number of pages.

## Solving the "Page X of Y" problem

In figure 7.3, we see a running header that starts on page 2. We also see a footer formatted as "page X of Y" where X is the current page and Y the total number of pages.

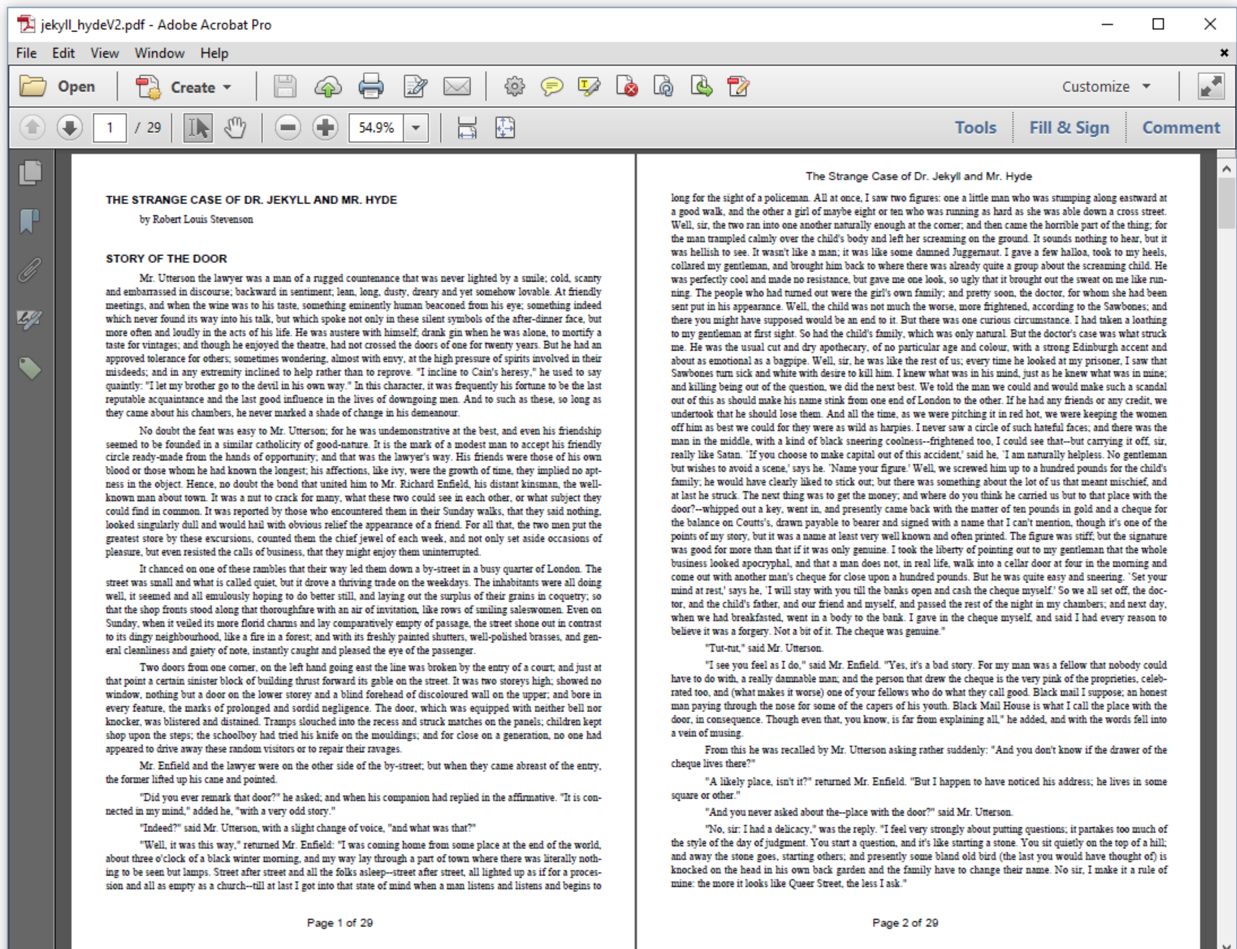



**Figure 7.3: Page X of Y footer**

The event handlers in the PageXofY[104] example are added like this:

```
1 PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2 pdf.addEventHandler(PdfDocumentEvent.START_PAGE,
3     new Header("The Strange Case of Dr. Jekyll and Mr. Hyde"));
4 PageXofY event = new PageXofY(pdf);
5 pdf.addEventHandler(PdfDocumentEvent.END_PAGE, event);
```

Instead of using low-level PDF operators to create the text object, we use the `showTextAligned()` method that was introduced when we talked about the `Canvas` object. See for instance the `handleEvent` implementation of the `Header` class.

[104] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2581-c07e03_pagexofy.java

```
1  protected class Header implements IEventHandler {
2      String header;
3      public Header(String header) {
4          this.header = header;
5      }
6      @Override
7      public void handleEvent(Event event) {
8          PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
9          PdfDocument pdf = docEvent.getDocument();
10         PdfPage page = docEvent.getPage();
11         if (pdf.getPageNumber(page) == 1) return;
12         Rectangle pageSize = page.getPageSize();
13         PdfCanvas pdfCanvas = new PdfCanvas(
14             page.getLastContentStream(), page.getResources(), pdf);
15         Canvas canvas = new Canvas(pdfCanvas, pdf, pageSize);
16         canvas.showTextAligned(header,
17             pageSize.getWidth() / 2,
18             pageSize.getTop() - 30, TextAlignment.CENTER);
19     }
20 }
```

This time, we use the `getLastContentStream()` method (line 14). As we use this class to create a `START_PAGE` event, the header will be the first thing that is written in the total content stream of the page.

The "Page X of Y" footer confronts us with a problem we've already solved once in chapter 2. In the JekyllHydeV8[105] example, we wanted to add the total number of pages of the document on the first page. However, at the moment we wrote that first page, we didn't know the total number of pages in advance. We used a placeholder instead of the final number, and we instructed iText not to flush any content to the `OutputStream` until all pages were created. At that moment, we used a `TextRenderer` to replace the place holder with the total number of pages, and we recreated the layout using the `relayout()` method.

There is one major disadvantage with this approach: it requires that we keep a lot of content in memory before we flush it to the `OutputStream`. The more pages, the more we'll risk an `OutOfMemoryException`. We can solve this problem by using a `PdfFormXObject` as placeholder.

A *form XObject* is a snippet of PDF syntax stored in a separate stream that is external to the content stream of a page. It can be referred to from different pages. If we create one form XObject as a placeholder, and we add it to multiple pages, we have to update it only once, and that change will

[105] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-2#1902-c02e12_jekyllhydev8.java

be reflected on every page. We can update the content of a form XObject as long as it hasn't been written to the `OutputStream`. This is what we'll do in the `PageXofY` class.

```
protected class PageXofY implements IEventHandler {
    protected PdfFormXObject placeholder;
    protected float side = 20;
    protected float x = 300;
    protected float y = 25;
    protected float space = 4.5f;
    protected float descent = 3;
    public PageXofY(PdfDocument pdf) {
        placeholder =
            new PdfFormXObject(new Rectangle(0, 0, side, side));
    }
    @Override
    public void handleEvent(Event event) {
        PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
        PdfDocument pdf = docEvent.getDocument();
        PdfPage page = docEvent.getPage();
        int pageNumber = pdf.getPageNumber(page);
        Rectangle pageSize = page.getPageSize();
        PdfCanvas pdfCanvas = new PdfCanvas(
            page.getLastContentStream(), page.getResources(), pdf);
        Canvas canvas = new Canvas(pdfCanvas, pdf, pageSize);
        Paragraph p = new Paragraph()
            .add("Page ").add(String.valueOf(pageNumber)).add(" of");
        canvas.showTextAligned(p, x, y, TextAlignment.RIGHT);
        pdfCanvas.addXObject(placeholder, x + space, y - descent);
        pdfCanvas.release();
    }
    public void writeTotal(PdfDocument pdf) {
        Canvas canvas = new Canvas(placeholder, pdf);
        canvas.showTextAligned(String.valueOf(pdf.getNumberOfPages()),
            0, descent, TextAlignment.LEFT);
    }
}
```

We define a member-variable name `placeholder` in line 2, and we initialize this `PdfFormXObject` in the constructor of our `IEventHandler` implementation (line 9-10). The other member-variables in line 3-7 are there for our convenience. They reflect the dimension of the placeholder (`side` is the side of the square that defines the placeholder), the position of the footer (`x` and `y`), the space between the "Page X of" and "Y" part (`space`) of the footer, and the space we will allow under the baseline of the "Y" value (`descent`).

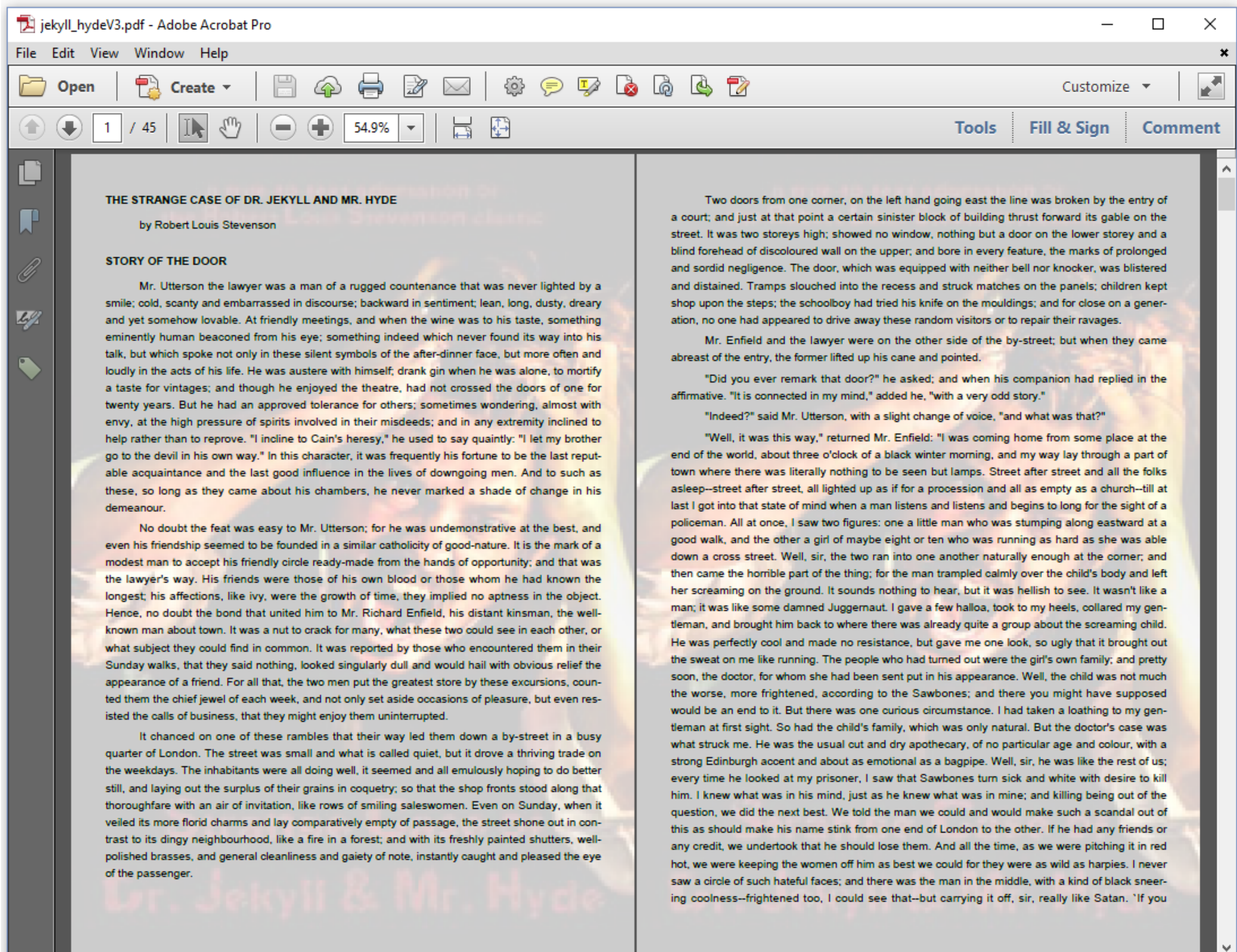
Lines 14 to 21 are identical to what we had in the `Header` class. We create the “Page X of” part of the footer in line 21 and 22. We add this `Paragraph` to the left of the coordinates `x` and `y` (line 24). We add the placeholder at the coordinates `x + space` and `y - descent`. We release the `Canvas`, but we don’t release the `placeholder` yet. Once the complete document is generated, we call the `writeTotal()` method, right before we close the document.

```
1 document.add(div);
2 event.writeTotal(pdf);
3 document.close();
```

In this `writeTotal()` method, we add the total number of pages at the coordinate `x = 0; y = descent` (line 30-31). This way, the “Page X of Y” text will always be nicely aligned with ‘x’ and ‘y’ as the coordinate that has “Page X of” to the left and “Y” to the right.

## Adding a transparent background image

In figure 7.4, we’ve added a transparent image in the background of each page of text. You could use this technique to add watermarks to a document.



THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE

by Robert Louis Stevenson

STORY OF THE DOOR

Mr. Utterson the lawyer was a man of a rugged countenance that was never lighted by a smile; cold, scanty and embarrassed in discourse; backward in sentiment; lean, long, dusty, dreary and yet somehow lovable. At friendly meetings, and when the wine was to his taste, something eminently human beaconed from his eye; something indeed which never found its way into his talk, but which spoke not only in these silent symbols of the after-dinner face, but more often and loudly in the acts of his life. He was austere with himself; drank gin when he was alone, to mortify a taste for vintages; and though he enjoyed the theatre, had not crossed the doors of one for twenty years. But he had an approved tolerance for others; sometimes wondering, almost with envy, at the high pressure of spirits involved in their misdeeds; and in any extremity inclined to help rather than to reprove. "I incline to Cain's heresy," he used to say quaintly: "I let my brother go to the devil in his own way." In this character, it was frequently his fortune to be the last reputable acquaintance and the last good influence in the lives of downgoing men. And to such as these, so long as they came about his chambers, he never marked a shade of change in his demeanour.

No doubt the feat was easy to Mr. Utterson; for he was undemonstrative at the best, and even his friendship seemed to be founded in a similar catholicity of good-nature. It is the mark of a modest man to accept his friendly circle ready-made from the hands of opportunity; and that was the lawyer's way. His friends were those of his own blood or those whom he had known the longest; his affections, like ivy, were the growth of time, they implied no aptness in the object. Hence, no doubt the bond that united him to Mr. Richard Enfield, his distant kinsman, the well-known man about town. It was a nut to crack for many, what these two could see in each other, or what subject they could find in common. It was reported by those who encountered them in their Sunday walks, that they said nothing, looked singularly dull and would hail with obvious relief the appearance of a friend. For all that, the two men put the greatest store by these excursions, counted them the chief jewel of each week, and not only set aside occasions of pleasure, but even resisted the calls of business, that they might enjoy them uninterrupted.

It chanced on one of these rambles that their way led them down a by-street in a busy quarter of London. The street was small and what is called quiet, but it drove a thriving trade on the weekdays. The inhabitants were all doing well, it seemed and all emulously hoping to do better still, and laying out the surplus of their grains in coquetry; so that the shop fronts stood along that thoroughfare with an air of invitation, like rows of smiling saleswomen. Even on Sunday, when it veiled its more florid charms and lay comparatively empty of passage, the street shone out in contrast to its dingy neighbourhood, like a fire in a forest; and with its freshly painted shutters, well-polished brasses, and general cleanliness and gaiety of note, instantly caught and pleased the eye of the passenger.

Two doors from one corner, on the left hand going east the line was broken by the entry of a court; and just at that point a certain sinister block of building thrust forward its gable on the street. It was two storeys high; showed no window, nothing but a door on the lower storey and a blind forehead of discoloured wall on the upper; and bore in every feature, the marks of prolonged and sordid negligence. The door, which was equipped with neither bell nor knocker, was blistered and distained. Tramps slouched into the recess and struck matches on the panels; children kept shop upon the steps; the schoolboy had tried his knife on the mouldings; and for close on a generation, no one had appeared to drive away these random visitors or to repair their ravages.

Mr. Enfield and the lawyer were on the other side of the by-street; but when they came abreast of the entry, the former lifted up his cane and pointed.

"Did you ever remark that door?" he asked; and when his companion had replied in the affirmative. "It is connected in my mind," added he, "with a very odd story."

"Indeed?" said Mr. Utterson, with a slight change of voice, "and what was that?"

"Well, it was this way," returned Mr. Enfield: "I was coming home from some place at the end of the world, about three o'clock of a black winter morning, and my way lay through a part of town where there was literally nothing to be seen but lamps. Street after street and all the folks asleep--street after street, all lighted up as if for a procession and all as empty as a church--till at last I got into that state of mind when a man listens and listens and begins to long for the sight of a policeman. All at once, I saw two figures: one a little man who was stumping along eastward at a good walk, and the other a girl of maybe eight or ten who was running as hard as she was able down a cross street. Well, sir, the two ran into one another naturally enough at the corner; and then came the horrible part of the thing; for the man trampled calmly over the child's body and left her screaming on the ground. It sounds nothing to hear, but it was hellish to see. It wasn't like a man; it was like some damned Juggernaut. I gave a few halloa, took to my heels, collared my gentleman, and brought him back to where there was already quite a group about the screaming child. He was perfectly cool and made no resistance, but gave me one look, so ugly that it brought out the sweat on me like running. The people who had turned out were the girl's own family; and pretty soon, the doctor, for whom she had been sent put in his appearance. Well, the child was not much the worse, more frightened, according to the Sawbones; and there you might have supposed would be an end to it. But there was one curious circumstance. I had taken a loathing to my gentleman at first sight. So had the child's family, which was only natural. But the doctor's case was what struck me. He was the usual cut and dry apothecary, of no particular age and colour, with a strong Edinburgh accent and about as emotional as a bagpipe. Well, sir, he was like the rest of us; every time he looked at my prisoner, I saw that Sawbones turn sick and white with desire to kill him. I knew what was in his mind, just as he knew what was in mine; and killing being out of the question, we did the next best. We told the man we could and would make such a scandal out of this as should make his name stink from one end of London to the other. If he had any friends or any credit, we undertook that he should lose them. And all the time, as we were pitching it in red hot, we were keeping the women off him as best we could for they were as wild as harpies. I never saw a circle of such hateful faces; and there was the man in the middle, with a kind of black sneering coolness--frightened too, I could see that--but carrying it off, sir, really like Satan. `If you

**Figure 7.4: Transparent background image**

Let's take a look at the `TransparentImage` class in the ImageWatermark[106] example.

```
1   protected class TransparentImage implements IEventHandler {
2       protected PdfExtGState gState;
3       protected Image img;
4       public TransparentImage(Image img) {
5           this.img = img;
6           gState = new PdfExtGState().setFillOpacity(0.2f);
7       }
8       @Override
9       public void handleEvent(Event event) {
10          PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
11          PdfDocument pdf = docEvent.getDocument();
12          PdfPage page = docEvent.getPage();
```

[106] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2582-c07e04_imagewatermark.java

```
13          Rectangle pageSize = page.getPageSize();
14          PdfCanvas pdfCanvas = new PdfCanvas(
15              page.getLastContentStream(), page.getResources(), pdf);
16          pdfCanvas.saveState().setExtGState(gState);
17          Canvas canvas = new Canvas(pdfCanvas, pdf, page.getPageSize());
18          canvas.add(img
19              .scaleAbsolute(pageSize.getWidth(), pageSize.getHeight()));
20          pdfCanvas.restoreState();
21          pdfCanvas.release();
22      }
23  }
```

Note that we store the `Image` object as a member-variable; this way, we can use it as many times we want and the bytes of the image will be added to the PDF document only once.

Creating a new `Image` instance of the same image in the `handleEvent` would result in a bloated PDF document. The same image bytes would be added to the document as many times as there are pages. This was already explained in chapter 3.

We also reuse the `PdfExtGState` object. This is a *graphics state* object that is external to the content stream. We use it to set the fill opacity to 20%.

In this example, we use a mix of `PdfCanvas` and `Canvas`. We use `PdfCanvas` to save, change, and restore the graphics state. We use `Canvas` to add the image resized to the dimensions of the page.

In this example, we didn't want the background image to appear for the table of contents. See figure 7.5.

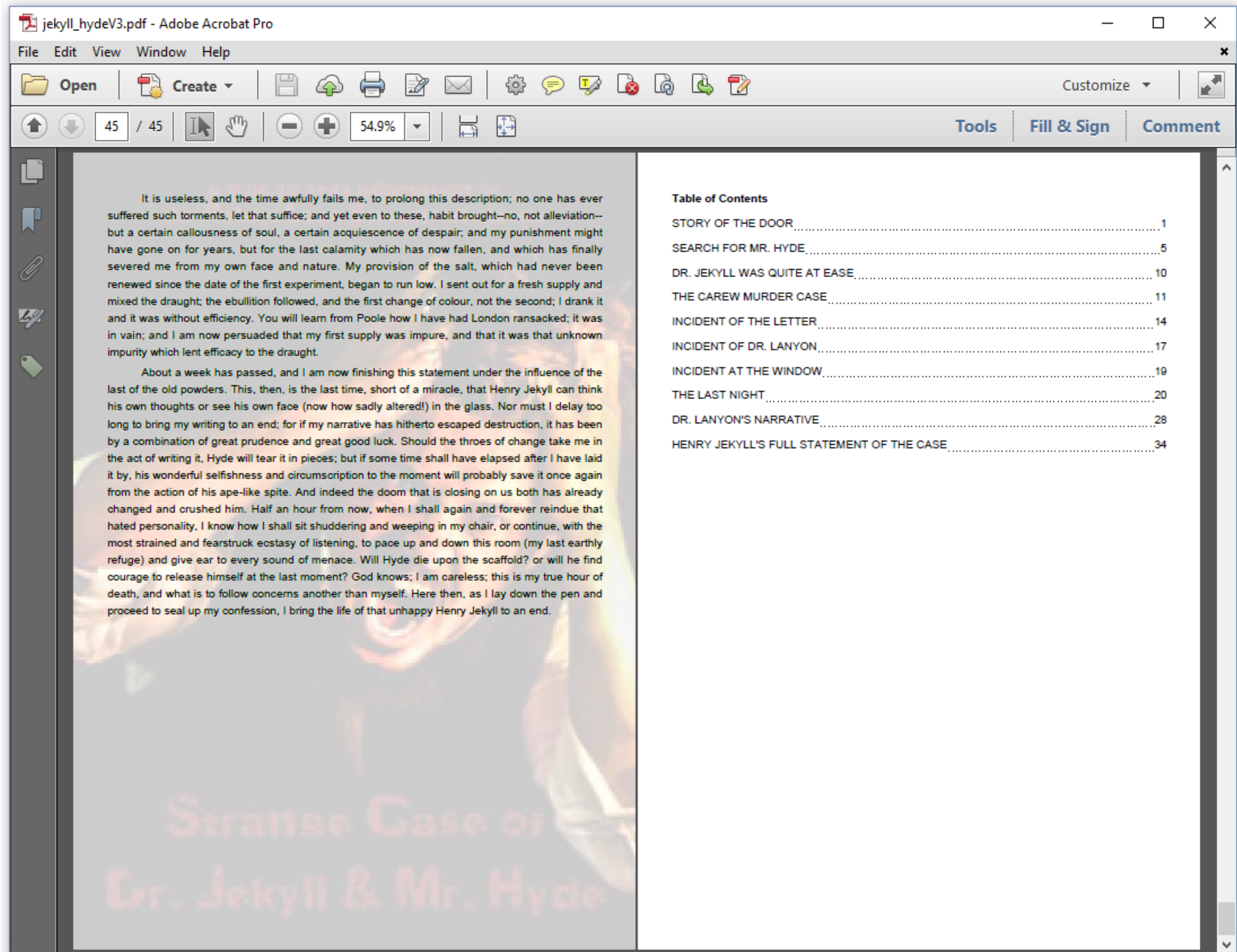



**Figure 7.5: Removing a specific handler**

We achieve this by removing the event handler, right before we add the table of contents.

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  Image img = new Image(ImageDataFactory.create(IMG));
3  IEventHandler handler = new TransparentImage(img);
4  pdf.addEventHandler(PdfDocumentEvent.START_PAGE, handler);
5  Document document = new Document(pdf);
6  ... // Code that adds the text of the novel
7  pdf.removeEventHandler(
8      PdfDocumentEvent.START_PAGE, handler);
9  document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
10 ... // code that adds the TOC
11 document.close();
```

We can remove a specific handler, using the `removeEventHandler()` method. We can remove all handlers using the `removeAllHandlers()` method. That's what we're going to do in the next example.

# Insert and remove page events

To obtain the PDF shown in figure 7.6, we took an existing PDF generated by one of the examples in the previous chapter. We inserted one page to be the new page 1. We removed all pages starting with the third chapter. As you can see, the bookmarks were updated accordingly.

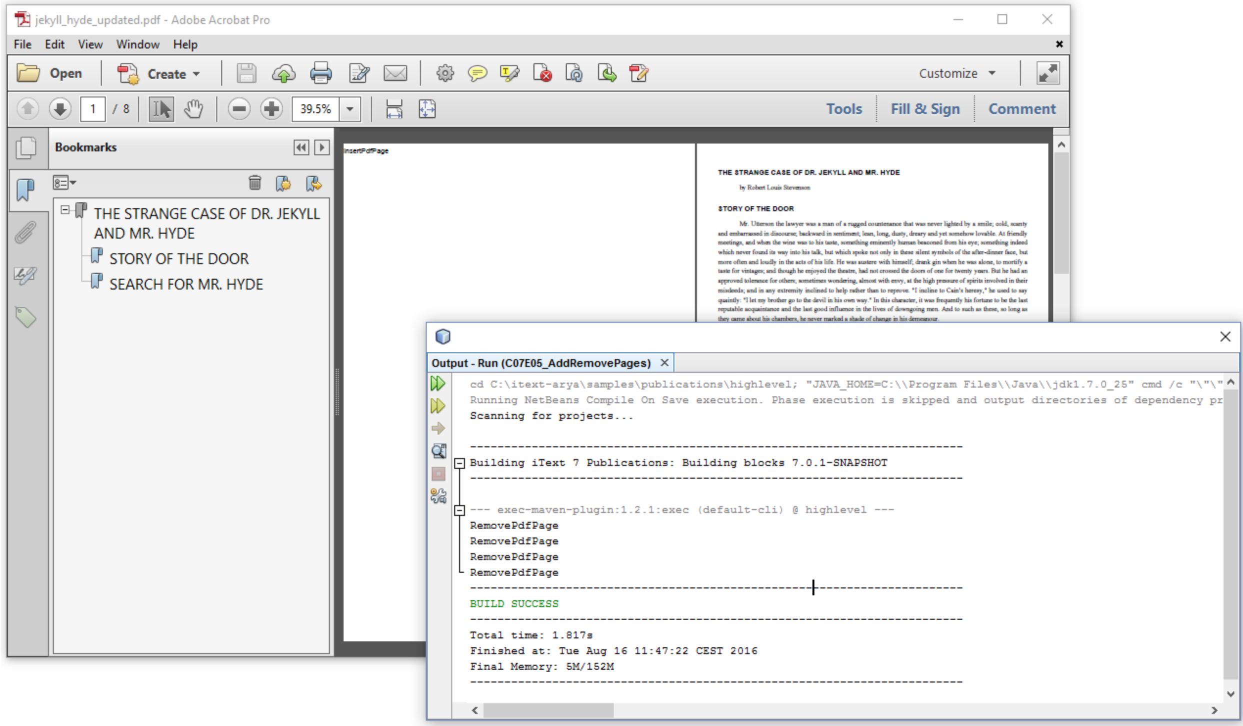



Figure 7.6: Insert and remove page events

The AddRemovePages[107] example uses the `INSERT_PAGE` event to add content to the inserted page, and the `REMOVE_PAGE` method to write something to the `System.out`. At some point, we remove all handlers.

```
1  public void manipulatePdf(String src, String dest) throws IOException {
2      PdfReader reader = new PdfReader(src);
3      PdfWriter writer = new PdfWriter(dest);
4      PdfDocument pdf = new PdfDocument(reader, writer);
5      pdf.addEventHandler(
6          PdfDocumentEvent.INSERT_PAGE, new AddPageHandler());
7      pdf.addEventHandler(
8          PdfDocumentEvent.REMOVE_PAGE, new RemovePageHandler());
9      pdf.addNewPage(1, PageSize.A4);
10     int total = pdf.getNumberOfPages();
```

[107] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2583-c07e05_addremovepages.java

```
11      for (int i = 9; i <= total; i++) {
12          pdf.removePage(9);
13          if (i == 12)
14              pdf.removeAllHandlers();
15      }
16      pdf.close();
17  }
18  protected class AddPageHandler implements IEventHandler {
19      @Override
20      public void handleEvent(Event event) {
21          PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
22          PdfDocument pdf = docEvent.getDocument();
23          PdfPage page = docEvent.getPage();
24          PdfCanvas pdfCanvas = new PdfCanvas(page);
25          Canvas canvas = new Canvas(pdfCanvas, pdf, page.getPageSize());
26          canvas.add(new Paragraph().add(docEvent.getType()));
27      }
28  }
29  protected class RemovePageHandler implements IEventHandler {
30      @Override
31      public void handleEvent(Event event) {
32          PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
33          System.out.println(docEvent.getType());
34      }
35  }
```

In this example, we have an `AddPageHandler` (line 18-28) and a `RemovePageHandler` (line 29-35). We declare these handlers to the `PdfDocument` as `INSERT_PAGE` and `REMOVE_PAGE` event respectively (line 5-8). The `AddPageHandler` will be triggered only once, when we add a new page (line 9). The remove page will be triggered four times. We remove all pages from page 9 to the total number of pages. We do this by removing page 9 over and over again (line 12), until no pages are left. As soon as we've removed page 12, we remove all handlers (line 13-14), which means that the event is triggered after we removed pages 9, 10, 11, and 12.

In the next example, we're going to define page labels.

# Page labels

Figure 7.7 shows a document with 38 pages. In the toolbar above the document, Adobe Acrobat shows that we're on page "i" or page 1 of 38. We have opened the Page Thumbnails panel to see a thumbnail for each page. We see that the first three pages are number i, ii, iii. Then we have 34 pages numbered from 1 to 34. Finally, we have a page with page label TOC.

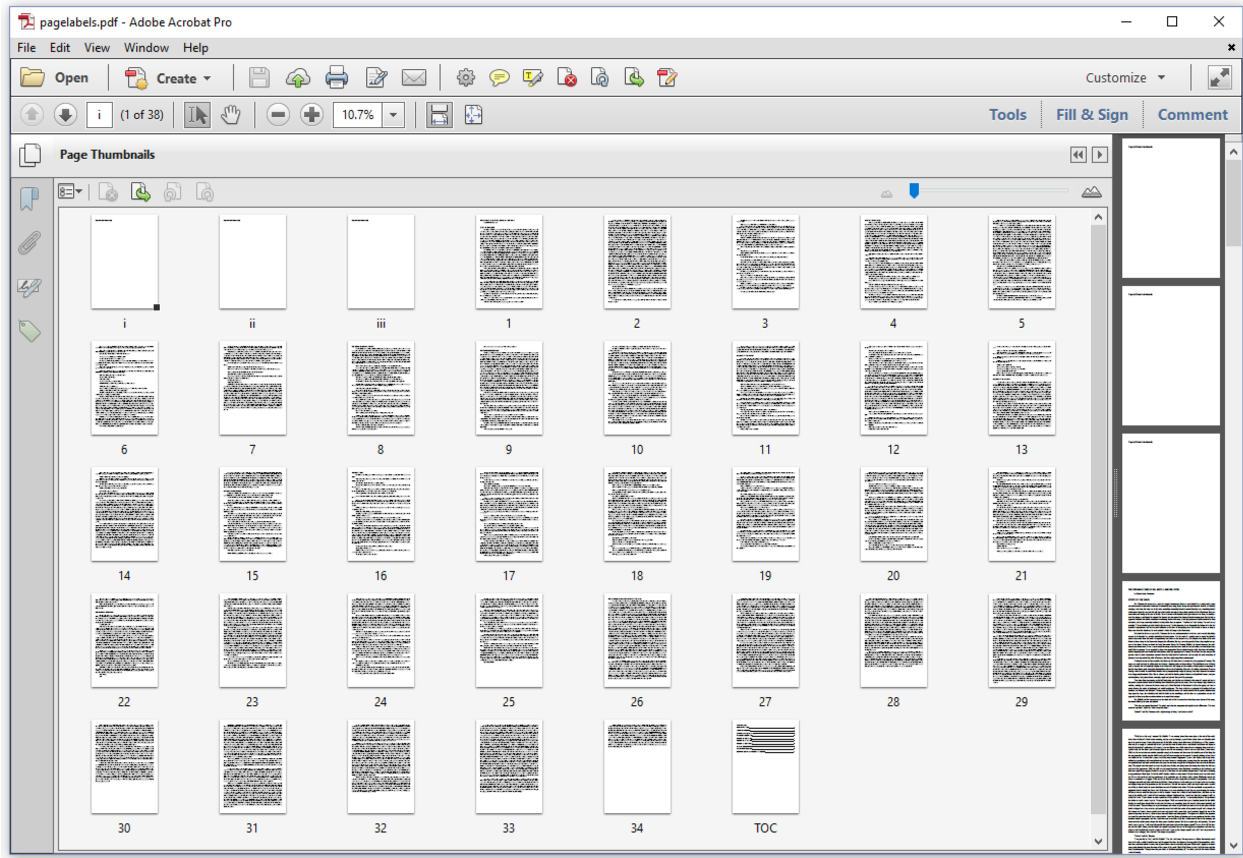



**Figure 7.7: Page labels**

These page labels aren't part of the actual content. For instance: you won't see them when you print the document. They are only visible in the PDF viewer –that is: if the PDF viewer supports page labels. The PDF in figure 7.7 was created using the PageLabels[108] example.

```
PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
PdfPage page = pdf.addNewPage();
page.setPageLabel(PageLabelNumberingStyleConstants
        .LOWERCASE_ROMAN_NUMERALS, null);
Document document = new Document(pdf);
document.add(new Paragraph().add("Page left blank intentionally"));
... // add some more pages left blank intentionally
page = pdf.getLastPage();
page.setPageLabel(PageLabelNumberingStyleConstants
    .DECIMAL_ARABIC_NUMERALS, null, 1);
... // add content of the novel
document.add(new AreaBreak(AreaBreakType.NEXT_PAGE));
p = new Paragraph().setFont(bold)
```

[108] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2584-c07e06_pagelabels.java

```
14     .add("Table of Contents").setDestination("toc");
15 document.add(p);
16 page = pdf.getLastPage();
17 page.setPageLabel(null, "TOC", 1);
18 ... // add table of contents
19 document.close();
```

We change the page label style three times in this code snippet:

1. We create the first page in line 2 and we set the page label style for this page to `LOWERCASE_ROMAN_NUMERALS`. We don't define a prefix. All the pages that follow this first page will be numbered like this: i, ii, iii, iv, v,... until we change the page label style. This happens in line 9.
2. In line 8, we get the last page that was added so far, and we change the page labels to `DECIMAL_ARABIC_NUMERALS` in line 9. Once more we don't define a prefix, and we tell the current page to start the page count with 1. We didn't really have to do this, because the page count always restart when you change the page labels. You can use this method if you don't want that to happen. For instance: we could pass 4 instead of 1 if we want the first page that follows the pages with Roman numerals to be page 4.
3. In line 17, we change the page label style to `null`. This means that no numbering will be used, even if we pass a value for the first page after the page label change. In this case, we do pass a prefix. That's the page label we see when we reach the table of contents of our document. The prefix can be combined with a page number, for instance if you have Arabic numerals for page numbers and the prefix is "X-", then the pages will be numbered as "X-1", "X-2", "X-3", and so on.

In this example, we had to manually open the Page Thumbnails panel to see the thumbnail overview of all the pages. We could have instructed the document to open that panel by default. In the next example, we'll change the page display and the page mode.

# Page display and page mode

The file `page_mode_page_layout.pdf` is almost identical to the file with the page labels we created in the previous example, but when we open it, we see that the panel with the page thumbnails is open by default. This is the *page mode*. We also see that the first page only takes half of the space that is available horizontally and that it's pushed to the right. At the bottom, we see that the second and third page are shown next to each other. This is the *page layout*.

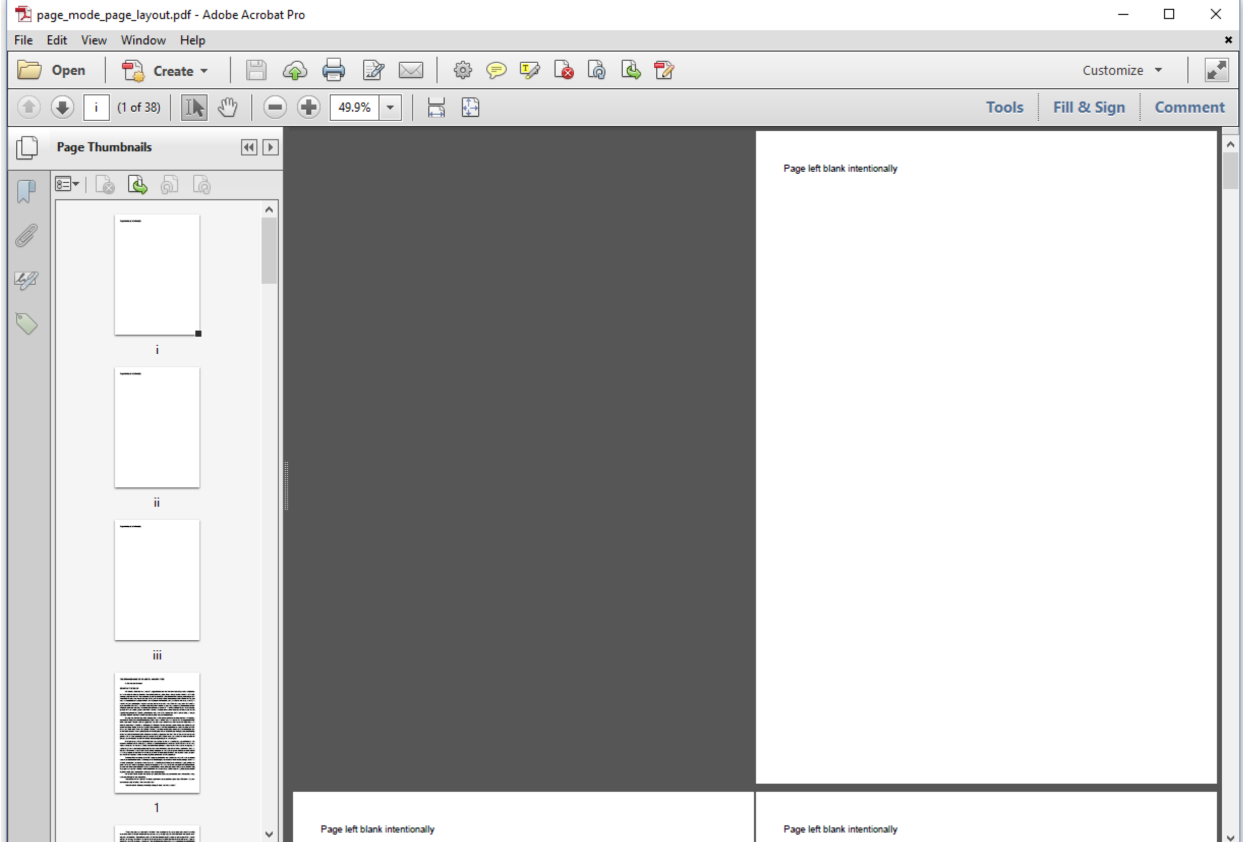



**Figure 7.8: Page layout and page mode**

The PageLayoutPageMode[109] example is identical to the previous example, except for the following lines.

```
pdf.getCatalog().setPageLayout(PdfName.TwoColumnRight);
pdf.getCatalog().setPageMode(PdfName.UseThumbs);
```

We get the *catalog* from the `PdfDocument`. The catalog is also known as the root dictionary of the PDF file. It's the first object that is read when a parser reads a PDF document.

We can set the page layout for the document with the `setPageLayout()` method using one of the following parameters:

- `PdfName.SinglePage`— Display one page at a time.
- `PdfName.OneColumn`— Display the pages in one column.
- `PdfName.TwoColumnLeft`— Display the pages in two columns, with the odd-numbered pages on the left.

[109] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2585-c07e07_pagelayoutpagemode.java

- `PdfName.TwoColumnRight`— Display the pages in two columns, with the odd-numbered pages on the right.
- `PdfName.TwoPageLeft`— Display the pages two at a time, with the odd-numbered pages on the left.
- `PdfName.TwoPageRight`— Display the pages two at a time, with the odd-numbered pages on the right.

We can set the page mode for the document with the `setPageMode()` method using one of the following parameters

- `PdfName.UseNone`— No panel is visible by default.
- `PdfName.UseOutlines`— The bookmarks panel is visible, showing the outline tree.
- `PdfName.UseThumbs`— A panel with pages visualized as thumbnails is visible.
- `PdfName.FullScreen`— The document is shown in full screen mode.
- `PdfName.UseOC`— The panel with the optional content structure is open.
- `PdfName.UseAttachments`— The attachments panel is visible.

We haven't discussed optional content yet, nor attachments. That's something we'll save for another tutorial.

When we use `PdfName.FullScreen`, the PDF will try to open in full screen mode. Many viewers won't do this without showing a warning first.

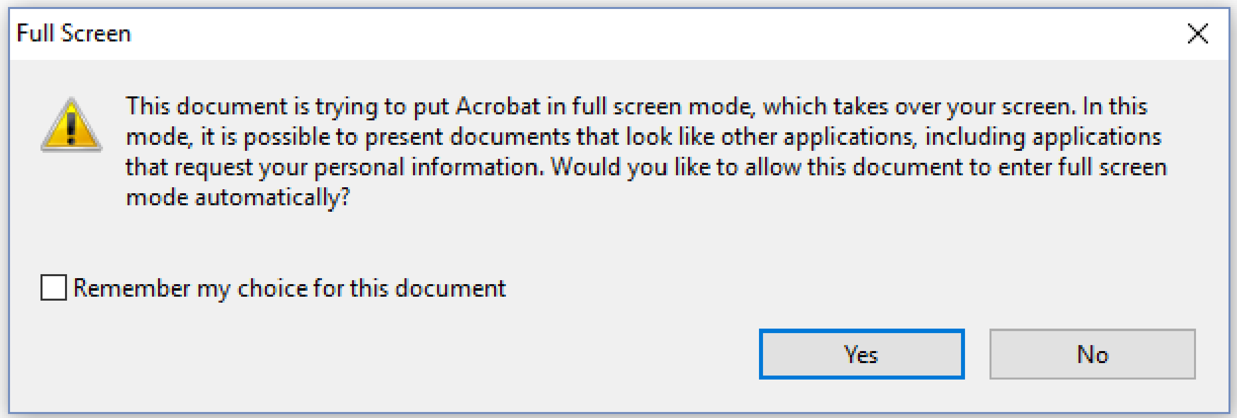



**Figure 7.9: Warning before switching to full screen mode**

The warning shown in figure 7.9 was triggered by the PDF created with the FullScreen[110] example.

[110] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2586-c07e08_fullscreen.java

```
pdf.getCatalog().setPageMode(PdfName.FullScreen);
PdfViewerPreferences preferences = new PdfViewerPreferences();
preferences.setNonFullScreenPageMode(
    PdfViewerPreferencesConstants.USE_THUMBS);
pdf.getCatalog().setViewerPreferences(preferences);
```

In this example, we also create a `PdfViewerPreferences` instance (line 2). We set the viewer preference that tells the viewer what to do when we exit full screen mode. The possible values for the `setNonFullScreenPageMode()` are:

- `PdfViewerPreferencesConstants.USE_NONE`— No panel is opened when we return from full screen mode.
- `PdfViewerPreferencesConstants.USE_OUTLINES`— The bookmarks panel is visible, showing the outline tree.
- `PdfViewerPreferencesConstants.USE_THUMBS`— A panel with pages visualized as thumbnails is visible.
- `PdfViewerPreferencesConstants.USE_OC`— The panel with the optional content structure is open.

We used `PdfViewerPreferencesConstants.USE_THUMBS` which means that we see the PDF as shown in figure 7.10.

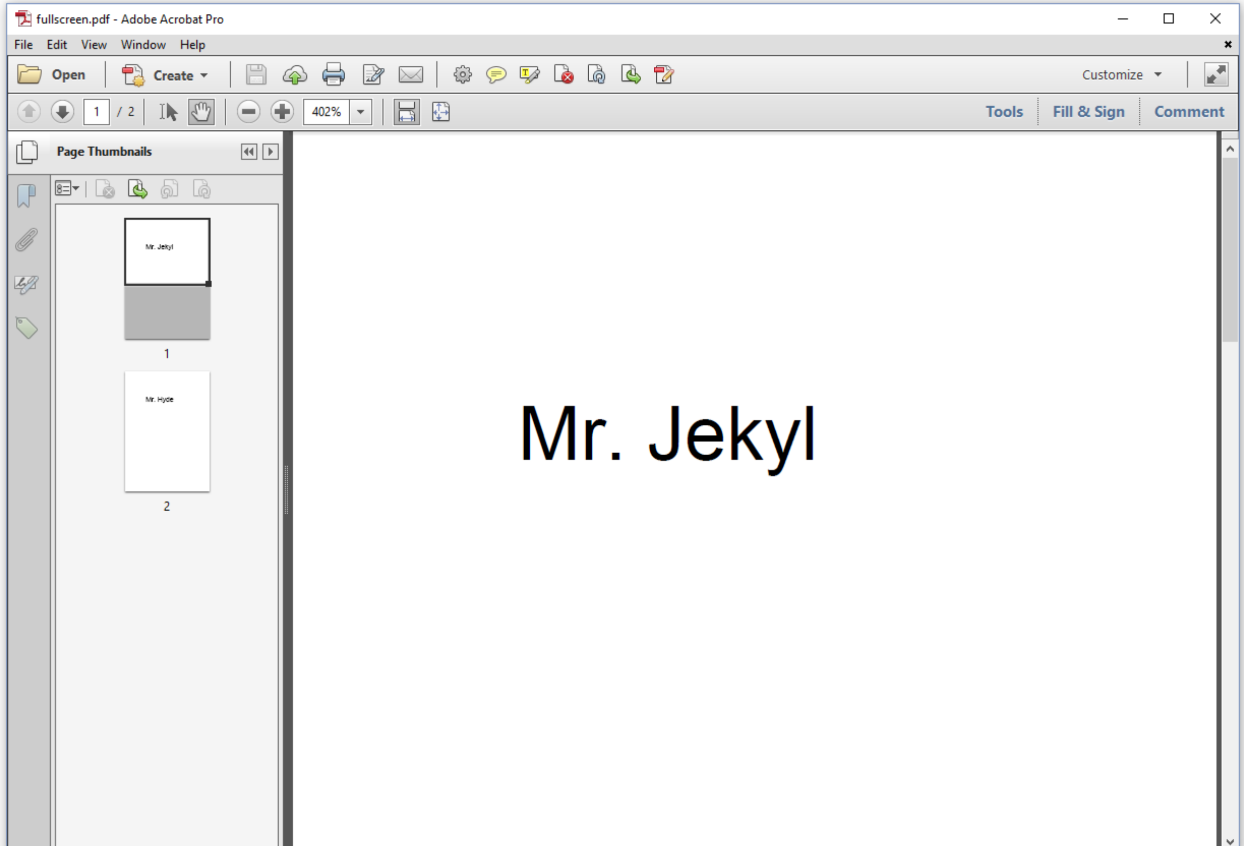



**Figure 7.10: Viewer after exiting full screen mode**

Let's take a look at some other viewer preferences that are available in the PDF specification.

# Viewer preferences

When we open the PDF shown in figure 7.11, we don't see a menu bar, we don't see a tool bar, we see the title of the document in the top bar, and so on.

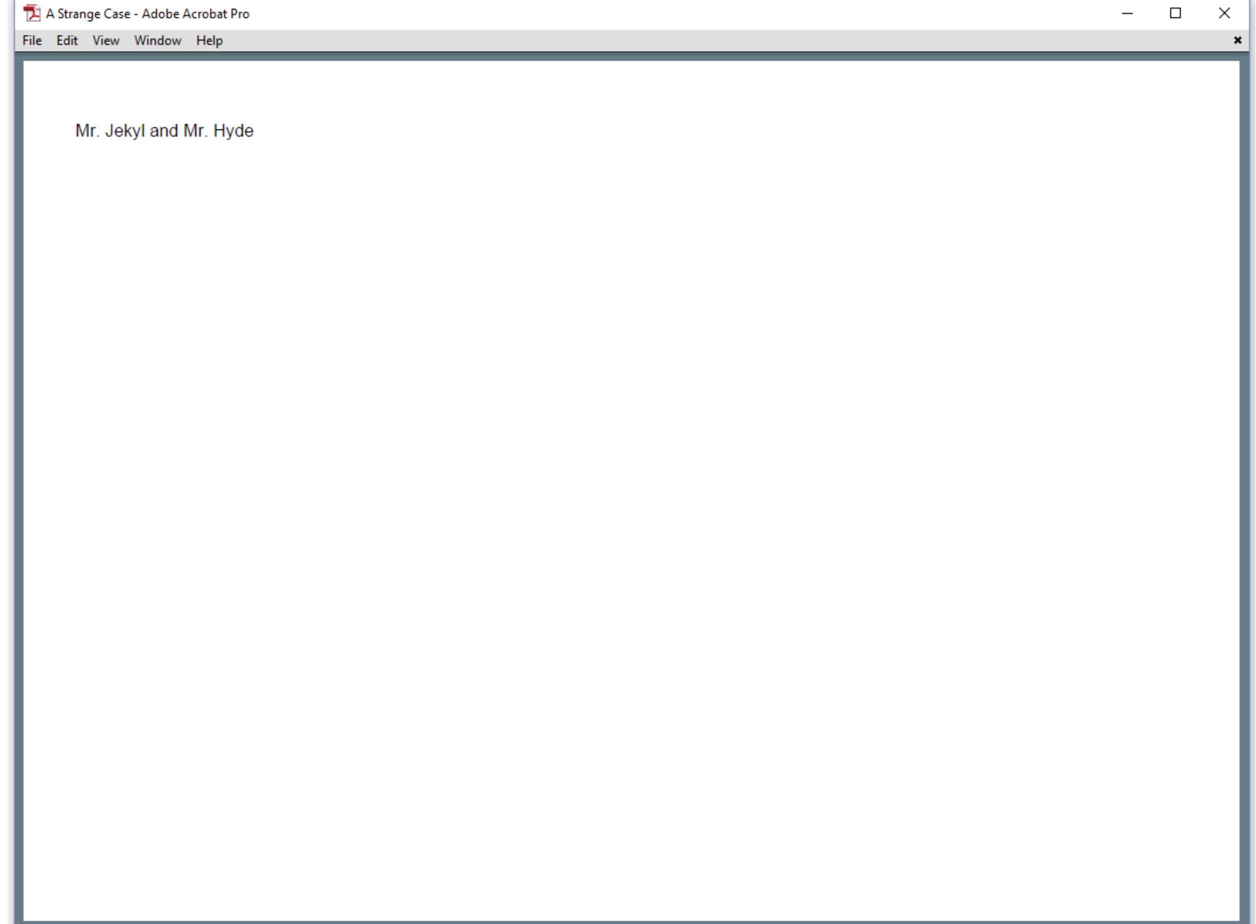



**Figure 7.11: Different viewer preferences at work in one document**

The ViewerPreferences[111] example shows us which viewer preferences have been set for this document.

```
1  public void createPdf(String dest) throws IOException {
2      PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
3      PdfViewerPreferences preferences = new PdfViewerPreferences();
4      preferences.setFitWindow(true);
5      preferences.setHideMenubar(true);
6      preferences.setHideToolbar(true);
7      preferences.setHideWindowUI(true);
8      preferences.setCenterWindow(true);
9      preferences.setDisplayDocTitle(true);
10     pdf.getCatalog().setViewerPreferences(preferences);
11     PdfDocumentInfo info = pdf.getDocumentInfo();
```

[111] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2587-c07e09_viewerpreferences.java

```
12      info.setTitle("A Strange Case");
13      Document document = new Document(pdf, PageSize.A4.rotate());
14      document.add(new Paragraph("Mr. Jekyl and Mr. Hyde"));
15      document.close();
16  }
```

All of these preferences expect `true` or `false` as a parameter; `false` being the default value if no preference is chosen.

- Line 4: with `setFitWindow()`, we tell the viewer to resize the document's window to fit the size of the first displayed page.
- Line 5: with `setHideMenubar()`, we tell the viewer to hide the menu bar; that is the bar with menu items such as *File*, *Edit*, *View*,…
- Line 6: with `setHideToolbar()`, we tell the viewer to hide the tool bar; that is the bar with the icons that give us direct access to some features also available through the menu items.
- Line 7: with `setHideWindowUI()`, we tell the viewer to hide all user interface elements such as scroll bars and other navigational controls.
- Line 8: with `setCenterWindow()`, we tell the viewer to position the document's window in the center of the screen.
- Line 9: with `setDisplayDocTitle()`, we tell the viewer to show the title of the document in the title bar.

Setting the title in the title bar requires that we define a title in the metadata. We do this in line 11-12. We'll have a closer look at metadata in a moment.

You can also use the `PdfViewerPreferences` class to define the predominant reading order of text using the `setDirection()` method, the view area using the `setViewArea()` and `setViewClip()` method. We won't do that in this tutorial, we'll skip to some printer preferences.

# Printer preferences

The mechanism of viewer preferences can also be used to set some printer preferences. For instance: we can select the area that will be printed by default using the `setPrintArea()` and the `setPrintClip()` method. Specific printer settings can be selected using the `setDuplex()` and `setPickTrayByPDFSize()` method. You can select a default page range that needs to be printed using the `setPrintPageRange()` method.

Figure 7.12 shows the default settings in the Print Dialog after using the `setPrintScaling()` and `setNumCopies()` method.

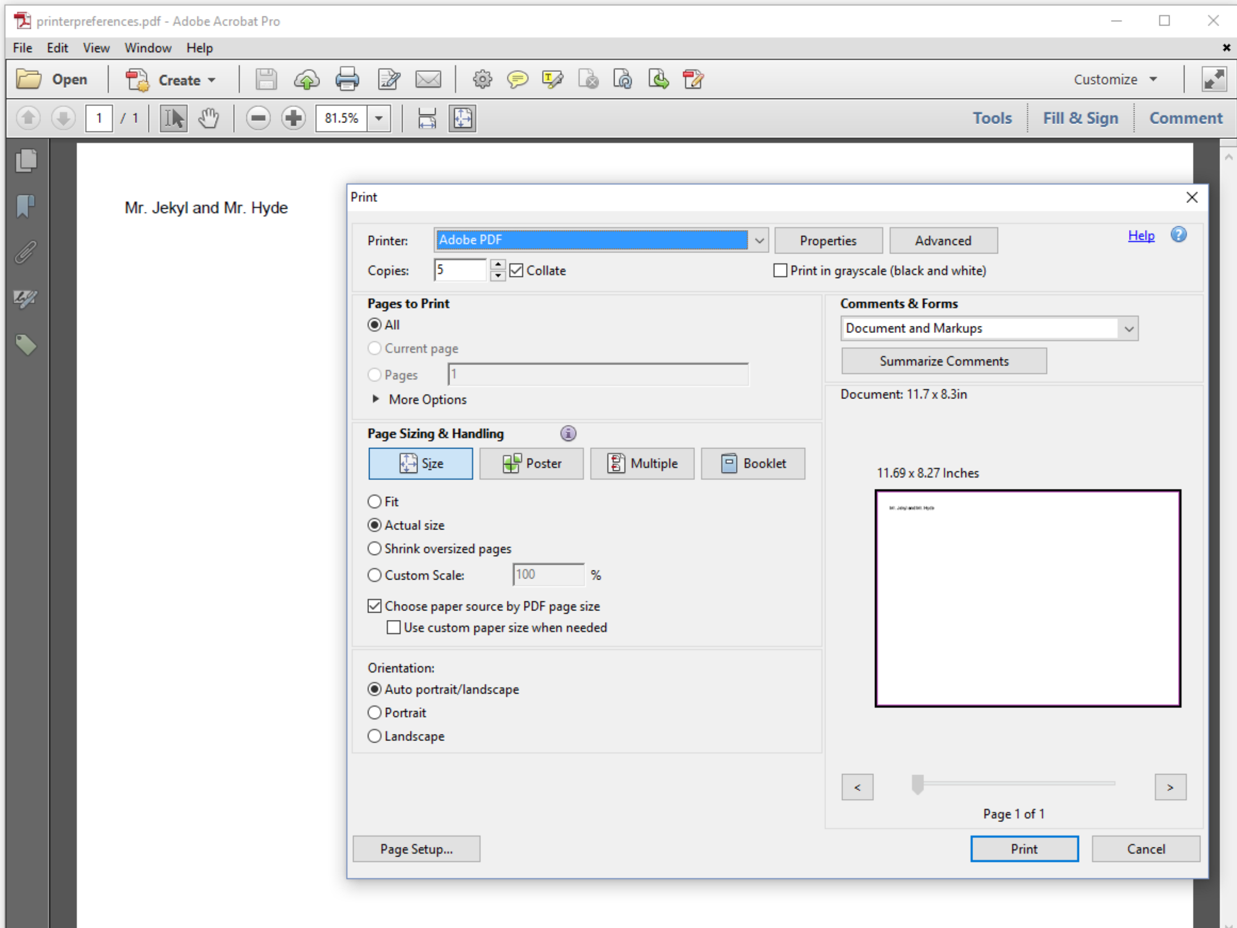



**Figure 7.12: Printer preferences**

The values in the screen shot correspond with the code of the PrinterPreferences[112] example.

```
1  PdfViewerPreferences preferences = new PdfViewerPreferences();
2  preferences.setPrintScaling(
3      PdfViewerPreferencesConstants.NONE);
4  preferences.setNumCopies(5);
5  pdf.getCatalog().setViewerPreferences(preferences);
```

Although PDF viewers nowadays offer many print-scaling options, the PDF specification only allows you to choose between `NONE` (no print scaling; the actual size of the document is preserved) and `APP_DEFAULT` (the default scaling of the viewer application). We set the number of copies to 5, which is reflected in the Print Dialog.

[112] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2588-c07e10_printerpreferences.java

As you can see, setting a printer preference doesn't *enforce* the preference that is chosen. For instance, if we set the number of copies to 5, a user can easily change this to any other number in the dialog. In PDF 2.0, an extra viewer preference, named `/Enforce` will be introduced. Its value will be an array. A PDF 2.0 viewer should check the entries and enforce all the viewer preferences that are present in that array. The Draft specification only provides a way to enforce the print scaling so far. We haven't implemented this `/Enforce` preference in iText yet, but we'll do so as soon as the PDF 2.0 standard aka ISO-32000-2 is released.

Once in a while, we get the question if it's possible to set a viewer preference to open a document at a certain page. It's possible to jump to a specific page when opening a document, but that isn't achieved using a viewer preference. We need an open action to do this.

# Open action and additional actions

The PDF document shown in figure 7.13 jumps straight to the last page when we open it. But there's more: when we leave the last page, we get a message saying "Goodbye last page!"

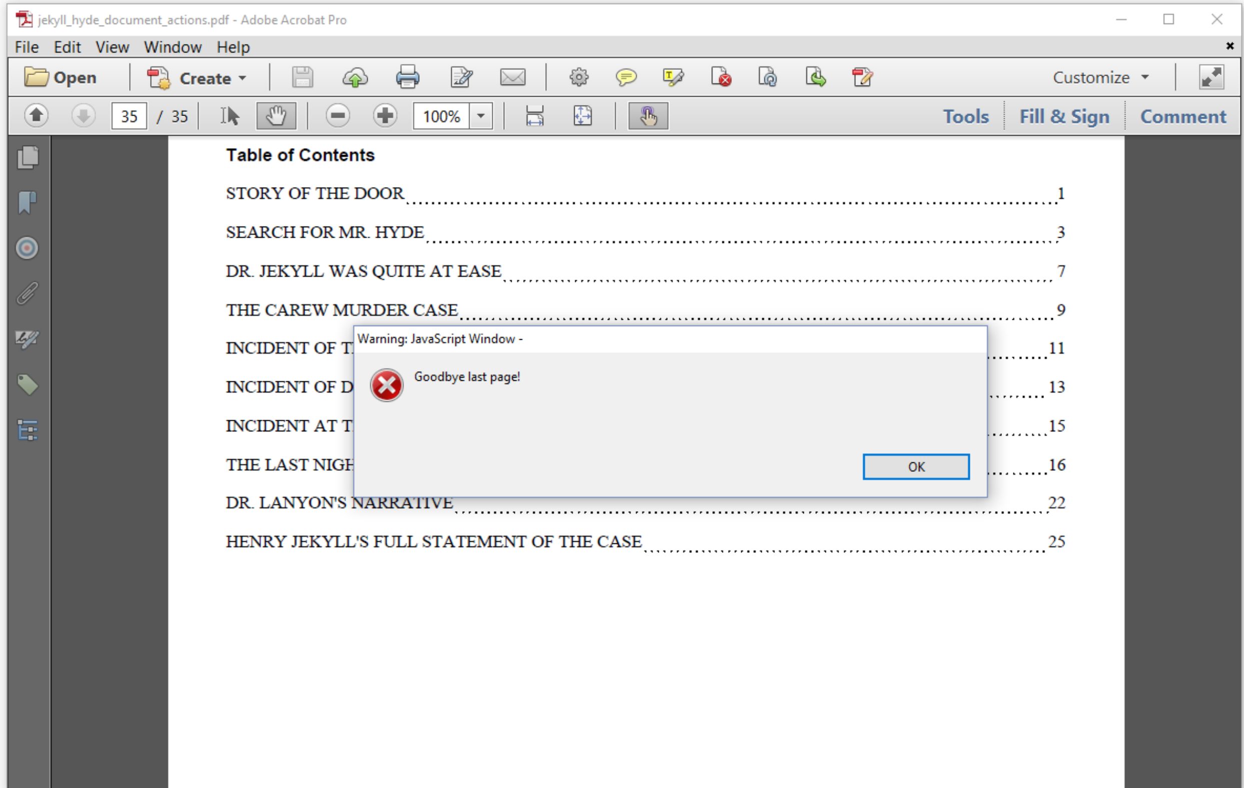



**Figure 7.13: Document opens on last page and page says goodbye when we leave it**

When we go to the first page, the document shows another alert: "This is where it starts!"

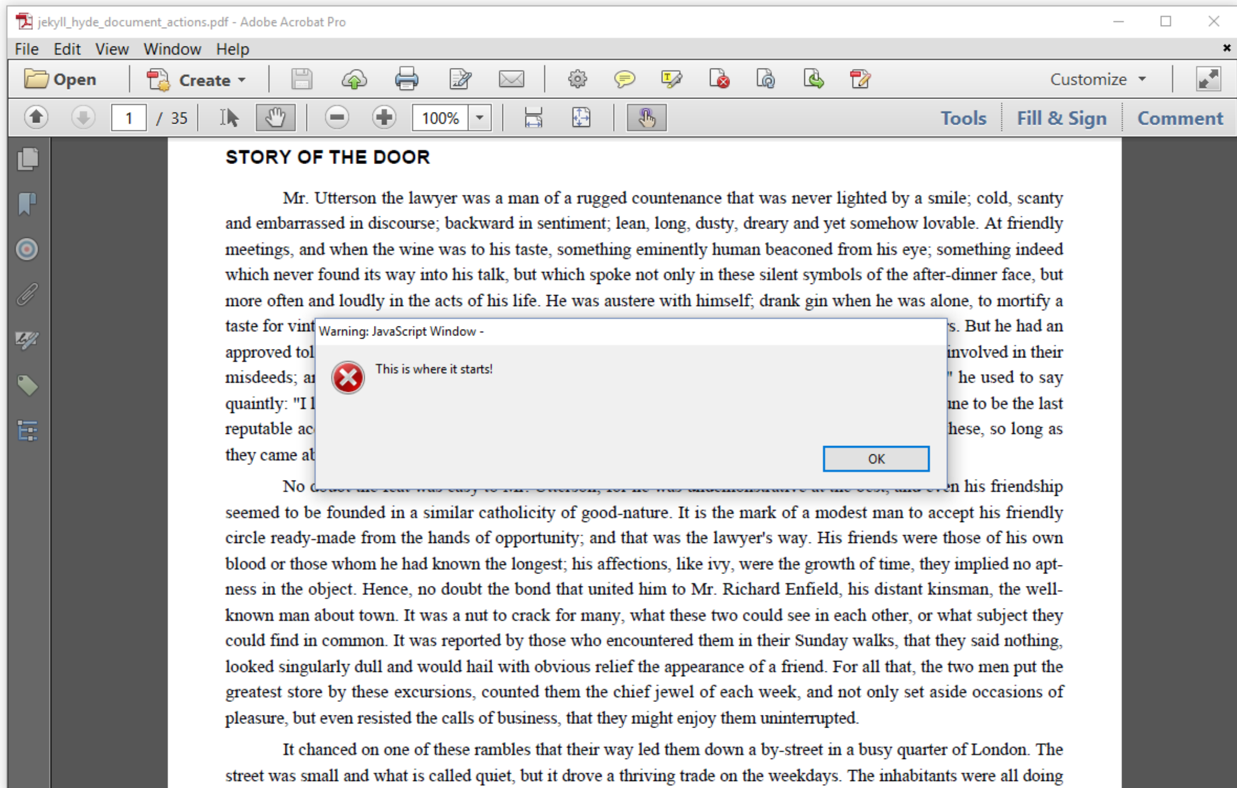



**Figure 7.14: The first page says ”This is where it starts!”**

Finally, when we close the document, it says: “Thank you for reading”.

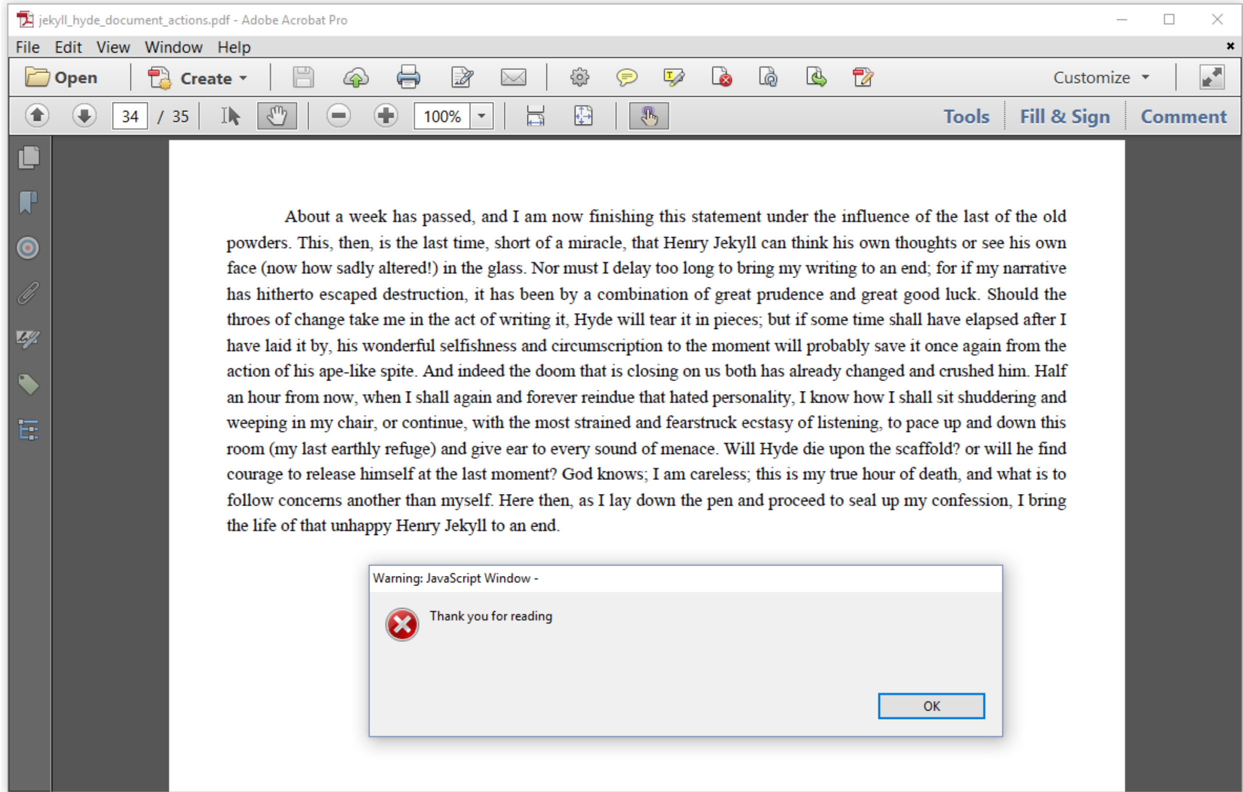



**Figure 7.15: The document says "Thank you for reading" upon closing it**

The action that jumps to the last page is an *open action*; all the other actions are *additional actions* with respect to events triggered on the document or on a page. The Actions[113] example shows us what it's all about.

```
 1 public void createPdf(String dest) throws IOException {
 2     PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
 3     pdf.getCatalog().setOpenAction(
 4             PdfDestination.makeDestination(new PdfString("toc")));
 5     pdf.getCatalog().setAdditionalAction(PdfName.WC,
 6         PdfAction.createJavaScript("app.alert('Thank you for reading');"));
 7     pdf.addNewPage().setAdditionalAction(PdfName.O,
 8         PdfAction.createJavaScript("app.alert('This is where it starts!');"));
 9     Document document = new Document(pdf);
10     PdfPage page = pdf.getLastPage();
11     page.setAdditionalAction(PdfName.C,
12         PdfAction.createJavaScript("app.alert('Goodbye last page!');"));
13     document.close();
14 }
```

[113] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2589-c07e11_actions.java

Let's start with the open action (line 3-4). This action is added to the catalog using the `setOpenAction()` method. This method accepts an instance of the `PdfDestination` class –in this case a link to a named destination– or of the `PdfAction` class.

The next action is an additional action for the document (line 5-6). This action is also added to the catalog, using the `setAdditionalAction()` method. The second parameter has to be a `PdfAction` object. The first parameter is one of the following names:

- `PdfName.WC`– which stands for *Will Close*. This action will be performed right before closing a document.
- `PdfName.WS`– which stands for *Will Save*. This action will be performed right before saving a document. Note that this will only work for viewers that allow you to save a document; and that *save* isn't the same as *save as* in this context.
- `PdfName.DS`– which stands for *Did Save*. This action will be performed right after saving a document. Note that this will only work for viewers that allow you to save a document; and that *save* isn't the same as *save as* in this context.
- `PdfName.WP`– which stands for *Will Print*. This action will be performed right before printing a document.
- `PdfName.DP`– which stands for *Did Print*. This action will be performed right after printing a document.

The next two additional actions are actions that are added to a `PdfPage` object (line 7-8; line 11-12). The parameters of this `setAdditionalAction()` method are again an instance of the `PdfAction` class as the second parameter, but the first parameter has to be one of the following names:

- `PdfName.O`– the action will be performed when the page is opened, for instance when a user navigates to it from the next or previous page, or by clicking a link. If this page is the first page that opens when opening a document, and if there's also an open action, the open action will be triggered first.
- `PdfName.C`– the action will be performed when the page is closed, for instance when a user navigates away from it by going to the next or previous page, or by clicking a link that moves away from this page.

There are more types of additional actions, especially in the context of interactive forms. Those actions are out of the scope of this tutorial and will be discussed in a tutorial about forms. We'll finish this chapter by looking at some writer properties.

## Writer properties

In one of the previous examples, we added some metadata to a `PdfDocumentInfo` object. We obtained this object from the `PdfDocument` using the `getDocumentInfo()` method. This `PdfDocumentInfo`

object corresponds with the info dictionary of the PDF; that's a dictionary containing metadata in the form of key-value pairs. This is how metadata was originally stored inside a PDF, but soon it turned out that it was a better idea to store metadata as XML inside a PDF file. This is shown in figure 7.16.

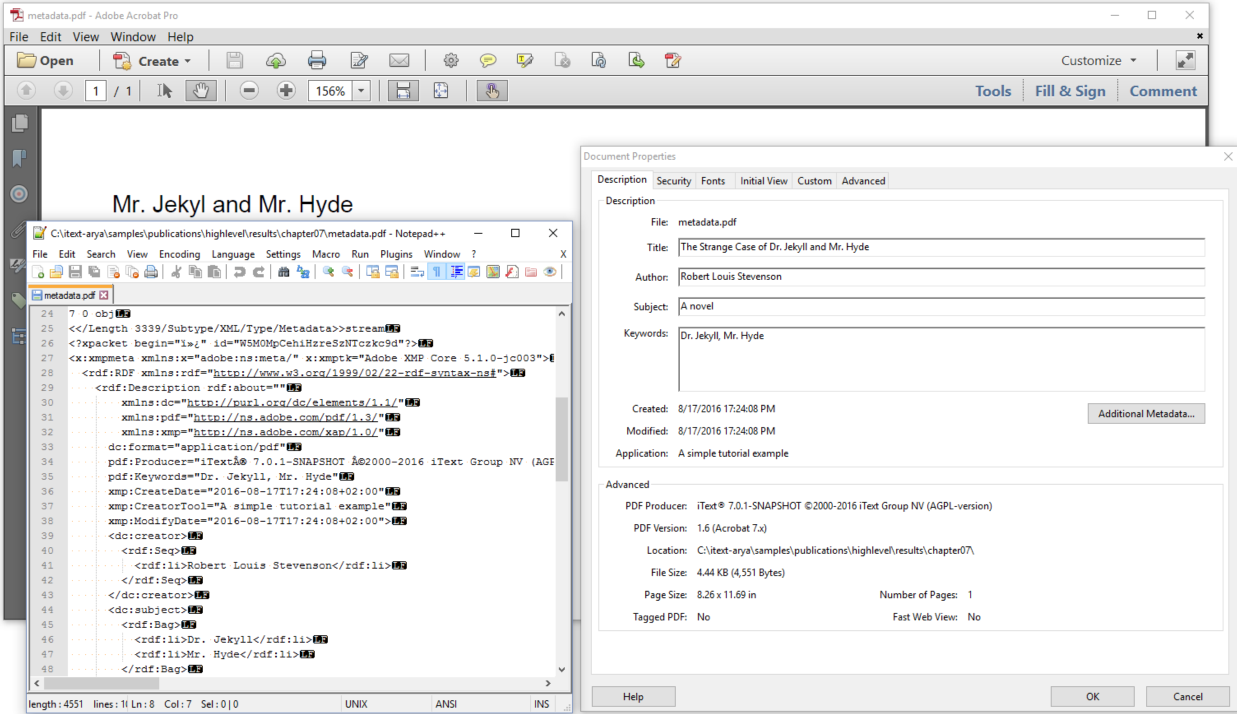



**Figure 7.16: PDF and metadata**

The XML is added as an uncompressed stream, which allows software that doesn't understand PDF syntax to extract the XML stream anyway and interpret it. The format of the XML is defined in the eXtensible Metadata Platform (XMP) standard. This standard allows much more flexibility than a simple key-value pair dictionary.

## XMP metadata

When you create a document, you can create your own XMP metadata using the `XMPMeta` class and then add this metadata to the `PdfDocument` by passing it as a parameter with the `setXmpMetadata()` method. But you can also ask iText to create the metadata automatically, based on the entries in the info dictionary. That's what we did in the Metadata[114] example.

[114] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2590-c07e12_metadata.java

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(
        new PdfWriter(dest,
            new WriterProperties()
                .addXmpMetadata()
                .setPdfVersion(PdfVersion.PDF_1_6)));
    PdfDocumentInfo info = pdf.getDocumentInfo();
    info.setTitle("The Strange Case of Dr. Jekyll and Mr. Hyde");
    info.setAuthor("Robert Louis Stevenson");
    info.setSubject("A novel");
    info.setKeywords("Dr. Jekyll, Mr. Hyde");
    info.setCreator("A simple tutorial example");
    Document document = new Document(pdf);
    document.add(new Paragraph("Mr. Jekyl and Mr. Hyde"));
    document.close();
}
```

We create a `WriterProperties` object (line 4) that is used as a second parameter of the `PdfWriter` class (line 3). We use the `addXmpMetadata()` method (line 5) to instruct iText to create an XMP stream based on the metadata added to the `PdfDocumentInfo` object: the title (line 8), the author (line 9), the subject (line 10), the keywords (line 11), and the creator application (line 12). The producer, creation time and modification time are set automatically. You can't change them.

iText 5 generated PDF 1.4 files by default. In some cases, this version was changed automatically when you used specific functionality, For instance: when using full compression, the version was changed to PDF 1.5. Full compression means that the cross-reference table and possibly some indirect objects will be compressed. That wasn't possible in PDF 1.4. iText 7 creates PDF 1.7 files (ISO-32000-1) by default. In the previous example, we changed the version to PDF 1.6 using the `setPdfVersion()` method on the `WriterProperties`.

You can also change the compression in the `WriterProperties`.

## Compression

In one of the event handler examples, we created a document with an image as background. The size of this PDF was 134 KB. In figure 7.17, you see another version of a document with the exact same content. The size of that PDF is only 125 KB.

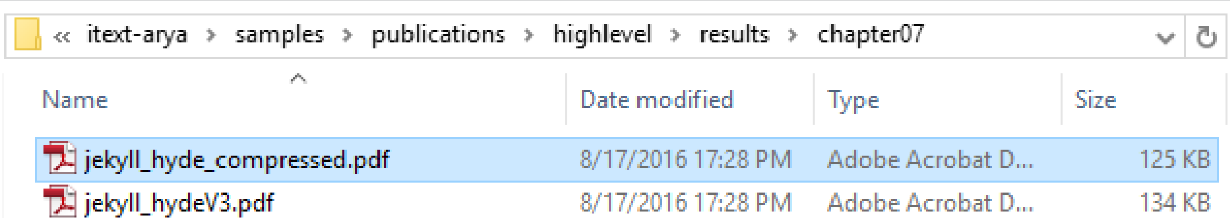



**Figure 7.17: PDF and compression**

This difference in size is caused by the way some content is stored inside the PDF. For small files, without many objects, the effect of full compression won't be significant. All content streams with PDF syntax are compressed by default by iText. Starting with PDF 1.5, more objects can be compressed, but that doesn't always make sense. A fully compressed file can count more bytes than an ordinary PDF 1.4 file if the PDF consists of only a dozen objects. The effect is more significant the more objects are needed in the PDF. If you plan to create large PDF files with many pages and many objects, you should take a look at the Compressed[115] example.

```
1 PdfDocument pdf = new PdfDocument(new PdfWriter(dest,
2     new WriterProperties().setFullCompressionMode(true)));
```

Once again, we use the `WriterProperties` object, now in combination with the `setFullCompressionMode()` method. There's also a `setCompressionLevel()` method that allows you to set a compression level ranging from `0` (best speed) to `9` (best compression), or you can set it to the default value `-1`.

We'll conclude this chapter with a small encryption example.

## Encryption

There are two ways to encrypt a PDF file. You can encrypt a PDF file using the public key of a public/private key pair. In that case, the PDF can only be viewed by the person who has access to the corresponding private key. This is very secure.

Using passwords is another way to encrypt a file. You can define two passwords for a PDF file: an owner password and a user password.

If a PDF is encrypted with an owner password, the PDF can be opened and viewed without that password, but some permissions can be in place. In theory, only the person who knows the owner password can change the permissions.

The concept of protecting a document using only an owner password is flawed. Many tools, including iText, can remove an owner password if there's no user password in place.

[115] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2591-c07e13_compressed.java

If a PDF is also encrypted with a user password, the PDF can't be opened without a password. Figure 7.18 shows what happens when you try to open such a file.

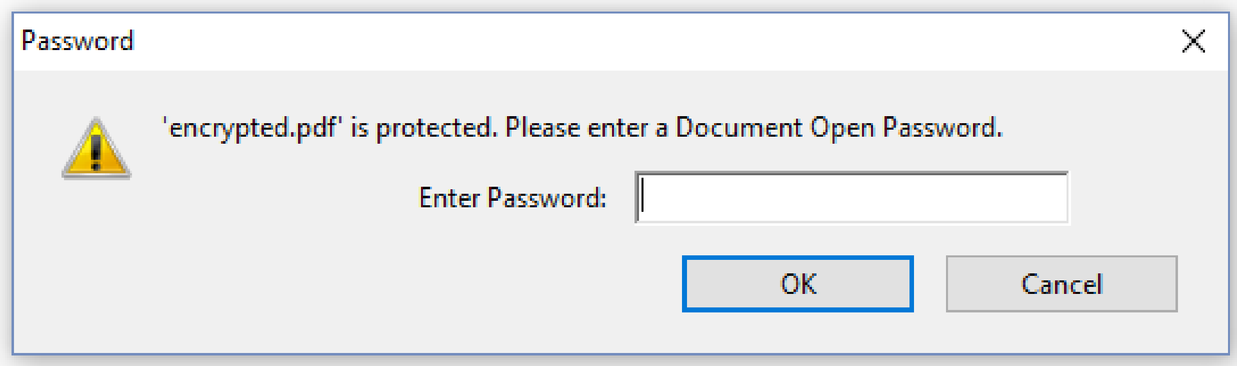



**Figure 7.18: A PDF that requires a password**

The document only opens when we pass one of the two passwords, the user password in which case the permissions will be in place, or the owner password in which case we can change the permissions.

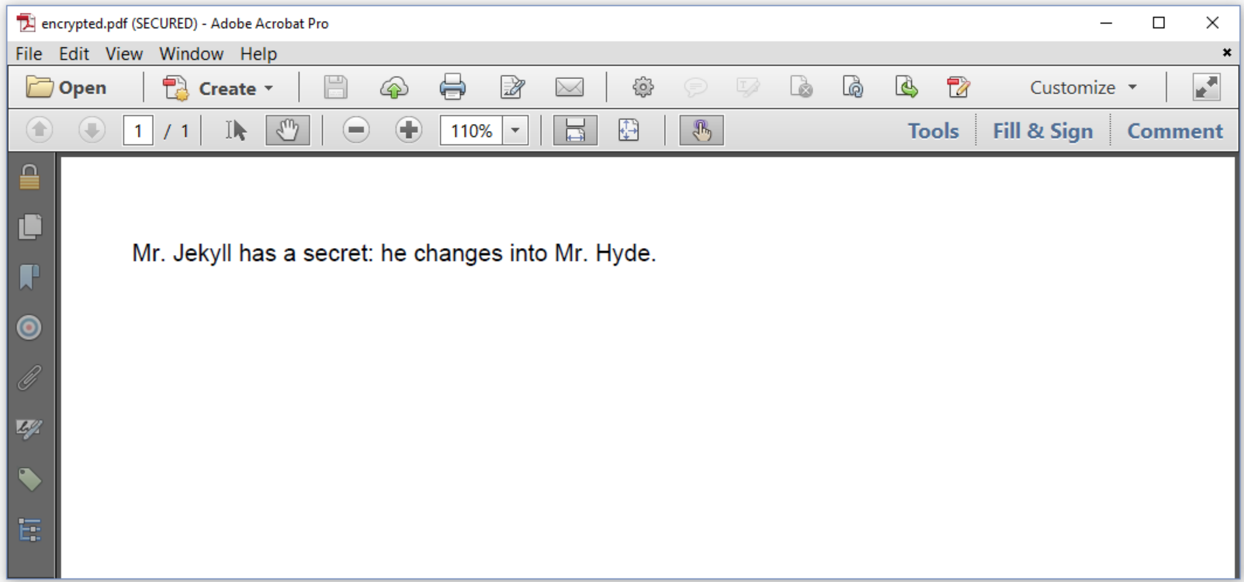



**Figure 7.19: A secured PDF**

As we can see in the Encrypted[116] example, the passwords and the permissions were defined using `WriterProperties`.

[116] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-7#2592-c07e14_encrypted.java

```
byte[] user = "It's Hyde".getBytes();
byte[] owner = "abcdefg".getBytes();
PdfDocument pdf = new PdfDocument(new PdfWriter(dest,
    new WriterProperties().setStandardEncryption(user, owner,
        EncryptionConstants.ALLOW_PRINTING
            | EncryptionConstants.ALLOW_ASSEMBLY,
        EncryptionConstants.ENCRYPTION_AES_256)));
```

We define a user password and an owner password as a byte array (line 1-2). Those are the first two parameters of the `setStandardEncryption()` method. The third parameter can be used to define permissions. In our example, we allow printing and document assembly –that is: splitting and merging. Finally, we define the encryption algorithm: AES 256.

The possible values for the permissions are:

- `ALLOW_DEGRADED_PRINTING`– Allow printing at a low resolution only,
- `ALLOW_PRINTING`– Allow printing at a low as well as at high resolution,
- `ALLOW_SCREENREADERS`– Allows the extraction of text for accessibility purposes,
- `ALLOW_COPY`– Allow copy/paste of text and images,
- `ALLOW_FILL_IN`– Allow filling out interactive form fields,
- `ALLOW_MODIFY_ANNOTATIONS`– Allow the modification of text annotations and filling out interactive form fields,
- `ALLOW_ASSEMBLY`– Allow the insertion, rotation and deletion of pages, as well as the creation of outline items and thumbnail images,
- `ALLOW_MODIFY_CONTENTS`– Allow the modification of the document by operations other than those controlled by `ALLOW_FILL_IN`, `ALLOW_MODIFY_ANNOTATIONS`, and `ALLOW_ASSEMBLY`.

If you want to combine different permissions, always use the "or" (`|`) operator because some permissions overlap. For instance `ALLOW_PRINTING` sets the bit for printing as well as for degraded printing.

iText supports the following encryption algorithms, to be used for the fourth parameter:

- `STANDARD_ENCRYPTION_40`– encrypt using the 40-bit alleged RC4 (ARC4) algorithm,
- `STANDARD_ENCRYPTION_128`– encrypt using the 128-bit alleged RC4 (ARC4) algorithm.
- `AES_128`– encrypt using the 128-bit AES algorithm,
- `AES_256`– encrypt using the 256-bit AES algorithm.

You can also add one of the following extra parameters to the encryption algorithm using an "or" (`|`) operation:

- `DO_NOT_ENCRYPT_METADATA`– if you want to avoid that the metadata will also be encrypted. Encrypting the metadata doesn't make sense if you want the metadata to be accessible for your document management system, but be aware that this option is ignored when using 40-bit ARC encryption.
- `EMBEDDED_FILES_ONLY`– if you want to encrypt the embedded files only, and not the actual PDF document. For instance: if the PDF document itself is a wrapper for other documents, such as a cover note explaining that it's not possible to open the document without having the right credentials at hand immediately. In this case the PDF is an unencrypted wrapper for encrypted documents.

All of these parameters can also be used for the `setPublicKeyEncryption()` method, in which case the first parameter is an array of `Certificate` objects, the second parameter an array with the corresponding permissions, and the third parameter the encryption mode –that is: the encryption algorithm and the option to not encrypt the metadata and to encrypt only the embedded files.

With this last example, we have given away the punch line of The Strange Case of Dr. Jekyll and Mr. Hyde: "Dr. Jekyll has a secret: he changes into Mr. Hyde." But you probably already knew that after all the Jekyll and Hyde examples we've made in this book.

# Summary

In this final chapter of the "iText 7: Building Blocks", we have covered the `IEventHandler` functionality that allows us to take action when specific events –such as starting, ending, inserting, and removing a page– occur. We looked at viewer preferences that allowed us to tell PDF viewers how to present the document when we open it. We could also use the mechanism of viewer preferences to set some printer preferences. Finally, we looked at some writer properties. We discussed these properties in the context of metadata, compression, and encryption.

We've covered a lot of ground in this tutorial. You should now have a clear understanding of the basic building blocks that are available when you want to create a document from scratch. You also know how to establish document interactivity (actions) and navigation (links, destinations, bookmarks). You know the mechanism to handle events, and you can set viewer preferences and writer properties. You're all set to create some really cool PDFs.

Obviously, this is not the definitive guide on iText. In future tutorials, we'll also take a look under the hood, examining the PDF syntax that is used to create the content stream of a page and the structure of a PDF document. We'll spend a tutorial on forms, how to create them and how to use forms as templates. We'll create a tutorial on manipulating existing documents by adding and removing content. We'll also update the old tutorial on digital signatures. We don't have an ETA on those tutorials, but make sure to check our Books[117] page on a regular basis.

**THE END**

[117]http://developers.itextpdf.com/books

# Appendix

## A: AbstractElement methods

| Method | Text / Link | Image | Tab | AreaBreak |
|---|---|---|---|---|
| addStyle() | Yes | Yes | No | No |
| setHyphenation() | Yes | No | No | No |
| setSplitCharacters() | Yes | No | No | No |
| setHorizontalAlignment() | No | Yes | No | No |
| setTextAlignment() | No | No | No | No |
| setHeight() | No | Yes | No | No |
| setWidth() | No | Yes | No | No |
| setWidthPercent() | No | Yes | No | No |
| setBorder() | Yes | Yes* | No | No |
| setBorderLeft() | Yes | Yes* | No | No |
| setBorderRight() | Yes | Yes* | No | No |
| setBorderTop() | Yes | Yes* | No | No |
| setBorderBottom() | Yes | Yes* | No | No |
| setBackgroundColor() | Yes | Yes* | No | No |
| setFont() | Yes | No | No | No |
| setFontSize() | Yes | No | No | No |
| setFontColor() | Yes | No | No | No |
| setBold() | Yes | No | No | No |
| setItalic() | Yes | No | No | No |
| setLineThrough() | Yes | No | No | No |
| setUnderline() | Yes | No | No | No |
| setTextRenderingMode() | Yes | No | No | No |
| setStrokeColor() | Yes | No | No | No |
| setStrokeWidth() | Yes | No | No | No |
| setCharacterSpacing() | Yes | No | No | No |
| setWordSpacing() | Yes | No | No | No |
| setFontKerning() | Typ | No | No | No |
| setFontScript() | Typ | No | No | No |
| setBaseDirection() | Typ | No | No | No |
| setRelativePosition() | Yes | Yes | No | No |
| setFixedPosition() | No | Yes | No | No |
| setAction() | Yes | Yes | No | No |
| setDestination() | Yes | Yes | No | No |

In the `Image` column, some methods have an asterisk next to "Yes". The asterisk means that you may not notice that the method works because the image isn't transparent. For instance: it doesn't make sense to set a background color for an opaque image: the image covers the background completely. The same is true for borders.

# B: BlockElement methods

| Method | Paragraph | Div | List | Table | Cell | LineSeparator |
|---|---|---|---|---|---|---|
| `addStyle()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setHyphenation()` | Yes | Yes | Yes | Yes | Yes | No |
| `setSplitCharacters()` | Yes | Yes | Yes | Yes | Yes | No |
| `setHorizontalAlignment()` | Yes | Yes | Yes | Yes | No* | Yes |
| `setVerticalAlignment()` | No | No | No | No | Yes | No |
| `setRotationAngle()` | Yes | Yes | Yes | No | Yes | No |
| `setTextAlignment()` | Yes | Yes | Yes | Yes | Yes | No |
| `setKeepTogether()` | Yes | Yes | Yes | Yes | Yes | No |
| `setKeepWithNext()` | Yes* | Yes* | Yes* | Yes* | No* | Yes* |
| `setHeight()` | Yes | Yes | Yes | No | Yes | No |
| `setWidth()` | Yes | Yes | Yes | Yes | No* | Yes |
| `setWidthPercent()` | Yes | Yes | Yes | Yes | No* | Yes |
| `setMargin()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setMargins()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setMarginLeft()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setMarginRight()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setMarginTop()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setMarginBottom()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setPadding()` | Yes | Yes | Yes | No | Yes | No |
| `setPaddings()` | Yes | Yes | Yes | No | Yes | No |
| `setPaddingLeft()` | Yes | Yes | Yes | No | Yes | No |
| `setPaddingRight()` | Yes | Yes | Yes | No | Yes | No |
| `setPaddingTop()` | Yes | Yes | Yes | No | Yes | No |
| `setPaddingBottom()` | Yes | Yes | Yes | No | Yes | No |
| `setBorder()` | Yes | Yes | Yes | Yes | Yes** | No |
| `setBorderLeft()` | Yes | Yes | Yes | Yes | Yes** | No |
| `setBorderRight()` | Yes | Yes | Yes | Yes | Yes** | No |
| `setBorderTop()` | Yes | Yes | Yes | Yes | Yes** | No |
| `setBorderBottom()` | Yes | Yes | Yes | Yes | Yes** | No |
| `setBackgroundColor()` | Yes | Yes | Yes | Yes | Yes | Yes |
| `setFont()` | Yes | Yes | Yes | Yes | Yes | No |
| `setFontSize()` | Yes | Yes | Yes | Yes | Yes | No |
| `setFontColor()` | Yes | Yes | Yes | Yes | Yes | No |
| `setBold()` | Yes | Yes | Yes | Yes | Yes | No |
| `setItalic()` | Yes | Yes | Yes | Yes | Yes | No |

| Method | Paragraph | Div | List | Table | Cell | LineSeparator |
|---|---|---|---|---|---|---|
| setLineThrough() | Yes | Yes | Yes | Yes | Yes | No |
| setUnderline() | Yes | Yes | Yes | Yes | Yes | No |
| setTextRenderingMode() | Yes | Yes | Yes | Yes | Yes | No |
| setStrokeColor() | Yes | Yes | Yes | Yes | Yes | No |
| setStrokeWidth() | Yes | Yes | Yes | Yes | Yes | No |
| setSpacingRatio() | Yes | Yes | Yes | Yes | Yes | No |
| setCharacterSpacing() | Yes | Yes | Yes | Yes | Yes | No |
| setWordSpacing() | Yes | Yes | Yes | Yes | Yes | No |
| setFontKerning() | Typ | Typ | Typ | Typ | Typ | No |
| setFontScript() | Typ | Typ | Typ | Typ | Typ | No |
| setBaseDirection() | Typ | Typ | Typ | Typ | Typ | No |
| setRelativePosition() | Yes | Yes | Yes | Yes | Yes | No |
| setFixedPosition() | Yes | Yes | Yes | Yes | Yes | No |
| setAction() | Yes | Yes | Yes | Yes | Yes | Yes |
| setDestination() | Yes | Yes | Yes | Yes | Yes | Yes |

There's an asterisk added to the "Yes" value of `setKeepWithNext()` because this method only works for objects added directly to the `Document`. It won't work for nested objects.

There's an asterisk added to the "No" value for some methods of the `Cell` method because these methods can't be used when the `Cell` is part of a `Table`. But you can also use a `Cell` outside the context of a `Table`. In that case, you can define the width and the horizontal alignment. There are two asterisks next to the `Yes` for the border methods of the `Cell`. Borders are drawn at the level of the `Table`. Changing the border of a `Cell` changes a line in the grid of the `Table`, but you can't use the `setBorder()` methods if you are using a `Cell` outside a `Table`.

# C: RootElement methods

| Method | Document | Canvas |
|---|---|---|
| setHyphenation() | Yes | Yes |
| setSplitCharacters() | Yes | Yes |
| setHorizontalAlignment() | No | No |
| setTextAlignment() | Yes | Yes |
| setHeight() | No | No |
| setWidth() | No | No |
| setWidthPercent() | No | No |
| setBorder() | No | No |
| setBorderLeft() | No | No |
| setBorderRight() | No | No |
| setBorderTop() | No | No |
| setBorderBottom() | No | No |
| setBackgroundColor() | No | No |

| Method | Document | Canvas |
|---|---|---|
| `setFont()` | Yes | Yes |
| `setFontSize()` | Yes | Yes |
| `setFontColor()` | Yes | Yes |
| `setBold()` | Yes | Yes |
| `setItalic()` | Yes | Yes |
| `setLineThrough()` | Yes | Yes |
| `setUnderline()` | Yes | Yes |
| `setTextRenderingMode()` | Yes | Yes |
| `setStrokeColor()` | Yes | Yes |
| `setStrokeWidth()` | Yes | Yes |
| `setCharacterSpacing()` | Yes | Yes |
| `setWordSpacing()` | Yes | Yes |
| `setFontKerning()` | Typ | Typ |
| `setFontScript()` | Typ | Typ |
| `setBaseDirection()` | Typ | Typ |
| `setRelativePosition()` | No | No |
| `setFixedPosition()` | No | No |
| `setDestination()` | No | No |
| `showTextAligned()` | Yes | Yes |
| `showTextAlignedKerned()` | Typ | Typ |
| `add(BlockElement)` | Yes | Yes |
| `add(Image)` | Yes | Yes |
| `add(AreaBreak)` | Yes | No |
| `setMargins()` | Yes | No |
| `setLeftMargin()` | Yes | No |
| `setRightMargin()` | Yes | No |
| `setTopMargin()` | Yes | No |
| `setBottomMargin()` | Yes | No |