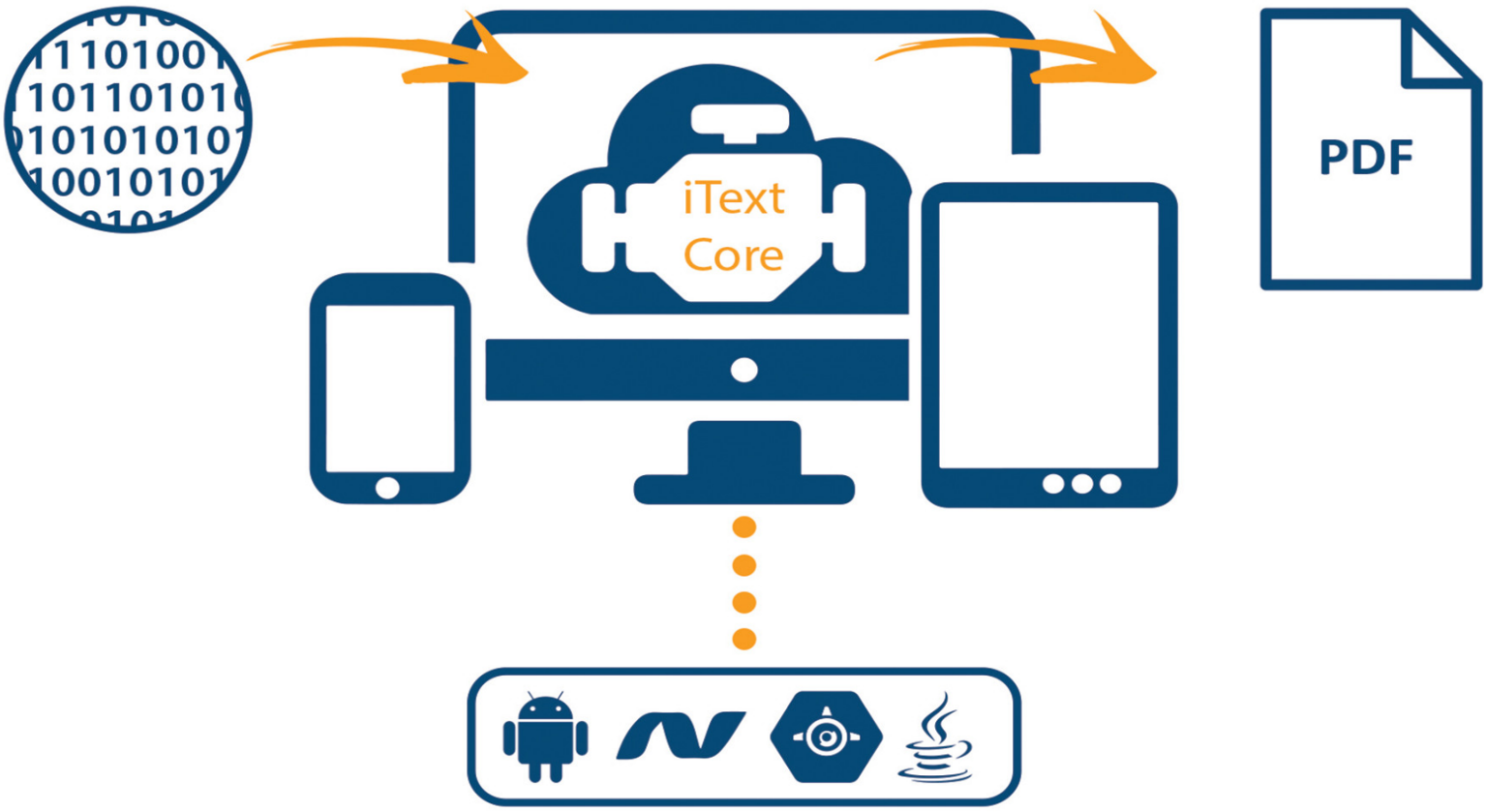
iText7 Jump-Start Tutorial



Bruno Lowagie

iText Software

iText 7: Jump-Start Tutorial

iText Software

This book is for sale at http://leanpub.com/itext7jump-starttutorial

This version was published on 2016-05-19

Contents

We announced iText 7 at the Great Indian Developer Summit in Bangalore on April 26, 2016. We released the first version of iText 7 in May. This tutorial is the first manual on how to use iText 7. It's not the *ultimate resource* the way "iText in Action" was for iText2 and "iText in Action - Second Edition" for iText5. It's called a Jump-Start Tutorial because it gives you a quick overview of the basic iText functionality, limited to PDF creation and manipulation. This allows new iText users to discover what's possible, whereas seasoned iText users will spot what's different compared to iText5.

iText 7 brings:

- a complete revision of all main classes and interfaces to make them more logical for users on one hand, while keeping them compatible with iText 5 as much as possible on the other hand,
- a completely new layout module, which surpasses the abilities of the iText 5 `ColumnText` object, and enables generation of complex PDF layouts,
- a complete rewrite of the font classes, enabling advanced typography.

It's our intention to release a series of separate tutorials zooming in on the different aspects of iText 7 in more detail, such as an in-depth overview of the high-level objects, an adapted "ABC of PDF" handbook, a manual dedicated to AcroForms and nothing but AcroForms. We'll also need tutorials for functionality that isn't covered in this Jump-Start Tutorial, such as digital signature technology and text extraction.

Follow @iText on Twitter[1] to be kept up-to-date for new releases of tutorial videos and documentation.

[1] https://twitter.com/iText

Before we start: installing iText 7

All the examples explained in this tutorial are available on our web site from this URL: http://developers.itextpdf.com 7-jump-start-tutorial/examples[2]

Before we start using iText 7, we need to install the necessary iText jars. The best way to do this, is by importing the jars from the Central Maven Repository. We have made some simple videos explaining how to do this using different IDEs:

- How to import iText 7 in Eclipse to create a Hello World PDF?[3]
- How to import iText 7 in Netbeans to create a Hello World PDF?[4]

In these tutorials, we only define the `kernel` and the `layout` projects as dependencies. Maven also automatically imports the `io` jar because the `kernel` packages depend on the `io` packages.

This is the complete list of dependencies that you'll need to define if you want to run all the examples in this tutorial:

```
<dependencies>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>kernel</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>io</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>layout</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
```

[2] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples
[3] https://www.youtube.com/watch?v=sxArv-GskLc&
[4] https://www.youtube.com/watch?v=VcOi99zW7O4

```
    </dependency>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>forms</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>pdfa</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>pdftest</artifactId>
        <version>7.0.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.18</version>
    </dependency>
</dependencies>
```

Every dependency corresponds with a jar in Java and with a DLL in C#.

- `kernel` and `io`: contain low-level functionality.
- `layout`: contains high-level functionality.
- `forms`: needed for all the AcroForm examples.
- `pdfa`: needed for PDF/A-specific functionality.
- `pdftest`: needed for the examples that are also a test.

In this tutorial, we won't use the following modules that are also available:

- `barcodes`: use this if you want to create bar codes.
- `hyph`: use this if you want text to be hyphenated.
- `font-asian`: use this is you need CJK functionality (Chinese / Japanese / Korean)
- `sign`: use this if you need support for digital signatures.

All the jars listed above are available under the AGPL license. Additional iText 7 functionality is available through AddOns, which are delivered as jars under a commercial license. If you want to use any of these AddOns, or if you want to use iText 7 with your proprietary code, you need to obtain a commercial license key for iText 7 (see the legal section of our web site[5]).

You can import such a license key using the license-key module. You can get the license-key jar like this:

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

Some functionality in iText is closed source. For instance, if you want to use **PdfCalligraph**, you need the `typography` module. This module won't work without an official license key.

[5] http://itextpdf.com/legal

Chapter 1: Introducing basic building blocks

When I created iText back in the year 2000, I tried to solve two very specific problems.

1. In the nineties, most PDF documents were created manually on the Desktop using tools like Adobe Illustrator or Acrobat Distiller. I needed to serve PDFs automatically in unattended mode, either in a batch process, or (preferably) on the fly, served to the browser by a web application. These PDF documents couldn't be produced manually due to the volume (the high number of pages and files) and because the content wasn't available in advance (it needed to be calculated, based on user input and/or real-time results from database queries). In 1998, I wrote a first PDF library that solved this problem. I deployed this library on a web server and it created thousands of PDF documents in a server-side Java application.
2. I soon faced a second problem: a developer couldn't use my first PDF library without consulting the PDF specification. As it turned out, I was the only member in my team who understood my code. This also meant that I was the only person who could fix my code if something broke. That's not a healthy situation in a software project. I solved this problem by rewriting my first PDF library from scratch, ensuring that knowing the PDF specification inside-out became optional instead of mandatory. I achieved this by introducing the concept of a `Document` to which a developer can add `Paragraph`, `List`, `Image`, `Chunk`, and other high-level objects. By combining these intuitive building blocks, a developer could easily create a PDF document programmatically. The code to create a PDF document became easier to read and easier to understand, especially by people who didn't write that code.

This is a jump-start tutorial to iText 7. We won't go into much detail, but let's start with some examples that involve some of these basic building blocks.

Introducing iText's basic building blocks

Many programming tutorials start with a Hello World example. This tutorial isn't any different.

This is the HelloWorld[6] example for iText 7:

[6] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1723-c01e01_helloworld.java

```
OutputStream fos = new FileOutputStream(dest);
PdfWriter writer = new PdfWriter(fos);
PdfDocument pdf = new PdfDocument(writer);
Document document = new Document(pdf);
document.add(new Paragraph("Hello World!"));
document.close();
```

Let's examine this example line by line:

1. In the first line, we create an `OutputStream`. If we wanted to write a web application, we could have created a `ServletOutputStream`; if we wanted to create a PDF document in memory, we could have used a `ByteArrayOutputStream`; but we want to create a PDF file on disk, so we create a `FileOutputStream` that writes a PDF document to the path defined in the `dest` parameter, for instance `results/chapter01/hello_world.pdf`.
2. In the second line, we create a `PdfWriter` instance. `PdfWriter` is an object that can write a PDF file. It doesn't know much about the actual content of the PDF document it is writing. The `PdfWriter` doesn't know what the document is about, it just writes different file parts and different objects that make up a valid document once the file structure is completed.
3. The `PdfWriter` knows what to write because it listens to a `PdfDocument`. The `PdfDocument` object manages the content that is added, distributes that content over different pages, and keeps track of whatever information is relevant for that content. In chapter 7, we'll discover that there are various flavors of `PdfDocument` classes a `PdfWriter` can listen to.
4. Now that we've created a `PdfWriter` and a `PdfDocument`, we're done with all the low-level, PDF-specific code. We create a `Document` that takes the `PdfDocument` as parameter. Now that we have the `document` object, we can forget that we're creating PDF.
5. We create a `Paragraph` containing the text `"Hello World"` and we add that paragraph to the `document` object.
6. We close the `document`. Our PDF has been created.

Figure 1.1 shows what the resulting PDF document looks like:

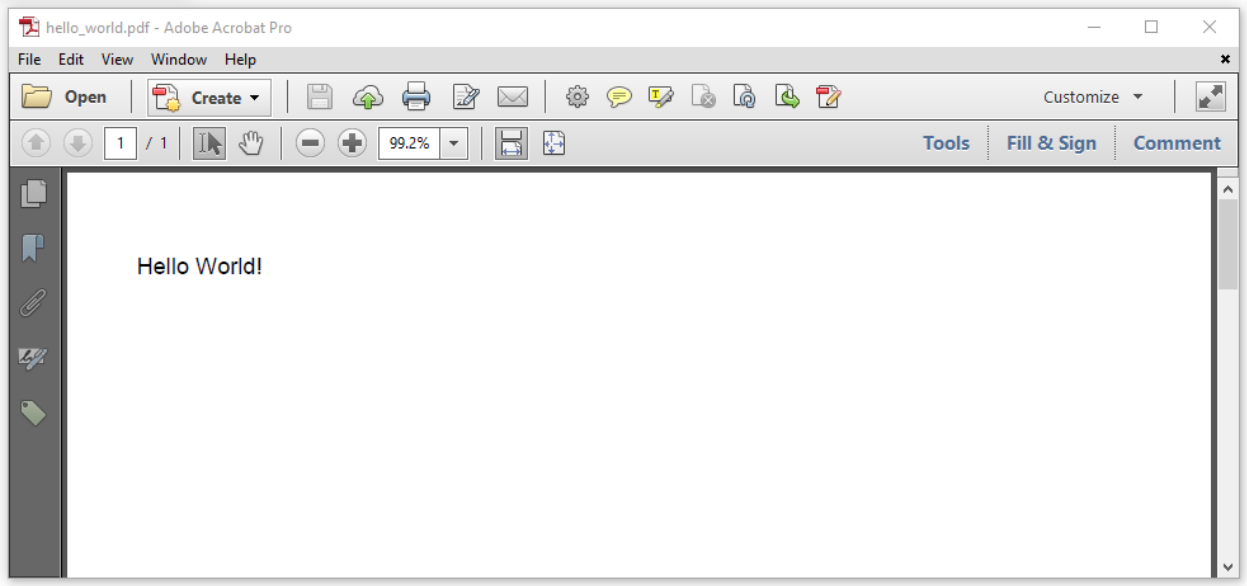


Figure 1.1: Hello World example

Let's add some complexity. Let's pick a different font and let's organize some text as a list; see Figure 1.2.

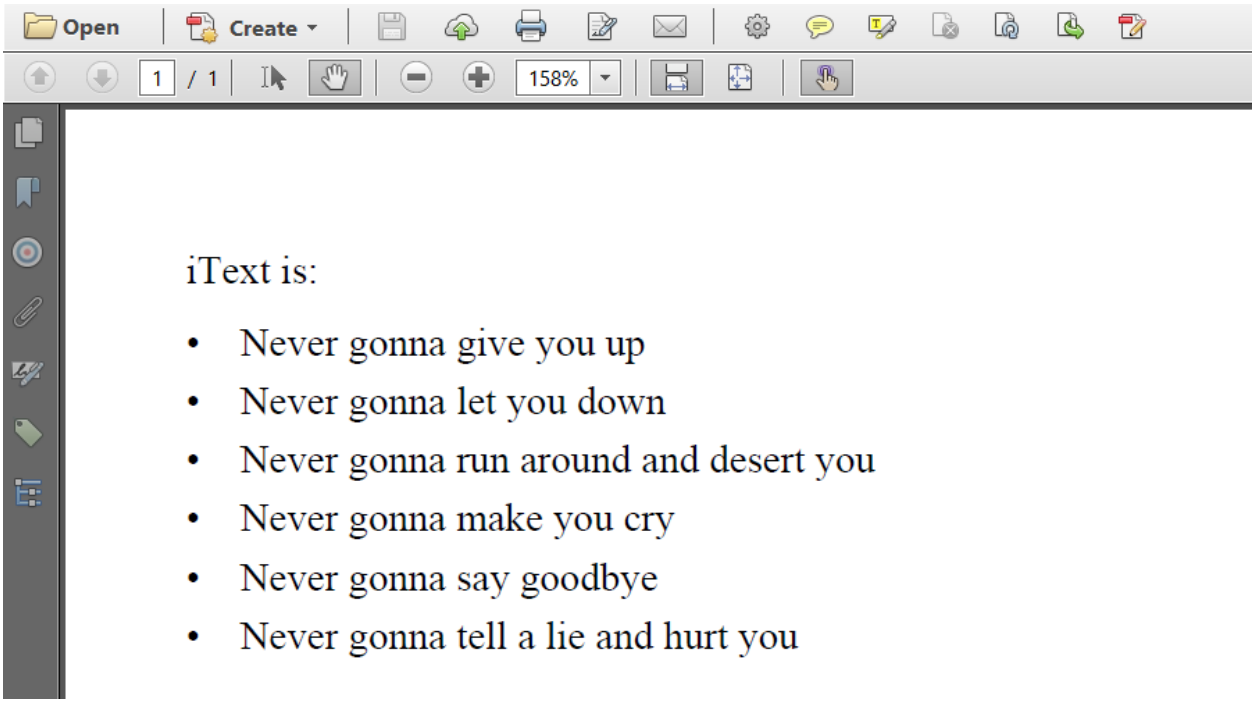


Figure 1.2: List example

The RickAstley[7] example shows how this is done:

[7] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1724-c01e02_rickastley.java

```
1  OutputStream fos = new FileOutputStream(dest);
2  PdfWriter writer = new PdfWriter(fos);
3  PdfDocument pdf = new PdfDocument(writer);
4  Document document = new Document(pdf);
5  // Create a PdfFont
6  PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
7  // Add a Paragraph
8  document.add(new Paragraph("iText is:").setFont(font));
9  // Create a List
10 List list = new List()
11     .setSymbolIndent(12)
12     .setListSymbol("\u2022")
13     .setFont(font);
14 // Add ListItem objects
15 list.add(new ListItem("Never gonna give you up"))
16     .add(new ListItem("Never gonna let you down"))
17     .add(new ListItem("Never gonna run around and desert you"))
18     .add(new ListItem("Never gonna make you cry"))
19     .add(new ListItem("Never gonna say goodbye"))
20     .add(new ListItem("Never gonna tell a lie and hurt you"));
21 // Add the list
22 document.add(list);
23 document.close();
```

Lines 1 to 4 and line 23 are identical to the Hello World example, but now we add more than just a `Paragraph`. iText always uses Helvetica as the default font for text content. If you want to change this, you have to create a `PdfFont` instance. You can do this by obtaining a font from the `PdfFontFactory` (line 6). We use this `font` object to change the font of a `Paragraph` (line 8) and a `List` (line 10). This `List` is a bullet list (line 12) and the list items are indented by 12 user units (line 11). We add six `ListItem` objects (line 15-20) and add the `list` to the `document`.

This is fun, isn't it? Let's introduce some images. Figure 1.3 shows what happens if we add an `Image` of a fox and a dog to a `Paragraph`.

Figure 1.3: Image example

If we remove the boiler-plate code from the QuickBrownFox[8] example, the following lines remain:

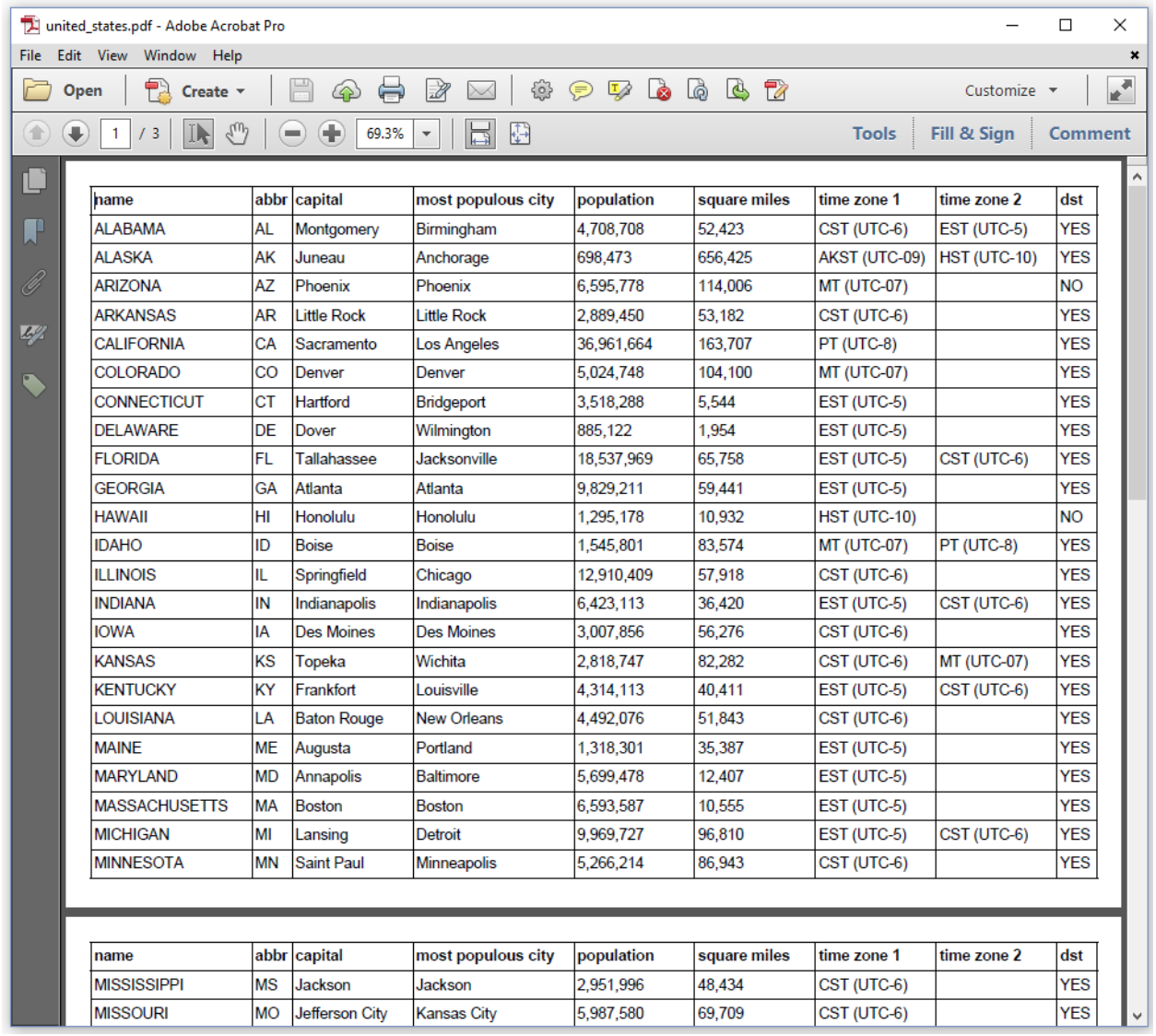
```
Image fox = new Image(ImageFactory.getImage(FOX));
Image dog = new Image(ImageFactory.getImage(DOG));
Paragraph p = new Paragraph("The quick brown ")
            .add(fox)
            .add(" jumps over the lazy ")
            .add(dog);
document.add(p);
```

We pass a path to an image to an `ImageFactory` that will return an object that can be used to create an iText `Image` object. The task of the `ImageFactory` is to detect the type of image that is passed (jpg, png, gif. bmp,...) and to process it so that it can be used in a PDF. In this case, we are adding the images so that they are part of a `Paragraph`. They replace the words "fox" and "dog".

Publishing a database

Many developers use iText to publish the results of a database query to a PDF document. Suppose that we have a database containing all the states of the United States of America and that we want to create a PDF that lists these states and some information about each one in a table as shown in Figure 1.4.

[8] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1725-c01e03_quickbrownfox.java



name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES
DELAWARE	DE	Dover	Wilmington	885,122	1,954	EST (UTC-5)		YES
FLORIDA	FL	Tallahassee	Jacksonville	18,537,969	65,758	EST (UTC-5)	CST (UTC-6)	YES
GEORGIA	GA	Atlanta	Atlanta	9,829,211	59,441	EST (UTC-5)		YES
HAWAII	HI	Honolulu	Honolulu	1,295,178	10,932	HST (UTC-10)		NO
IDAHO	ID	Boise	Boise	1,545,801	83,574	MT (UTC-07)	PT (UTC-8)	YES
ILLINOIS	IL	Springfield	Chicago	12,910,409	57,918	CST (UTC-6)		YES
INDIANA	IN	Indianapolis	Indianapolis	6,423,113	36,420	EST (UTC-5)	CST (UTC-6)	YES
IOWA	IA	Des Moines	Des Moines	3,007,856	56,276	CST (UTC-6)		YES
KANSAS	KS	Topeka	Wichita	2,818,747	82,282	CST (UTC-6)	MT (UTC-07)	YES
KENTUCKY	KY	Frankfort	Louisville	4,314,113	40,411	EST (UTC-5)	CST (UTC-6)	YES
LOUISIANA	LA	Baton Rouge	New Orleans	4,492,076	51,843	CST (UTC-6)		YES
MAINE	ME	Augusta	Portland	1,318,301	35,387	EST (UTC-5)		YES
MARYLAND	MD	Annapolis	Baltimore	5,699,478	12,407	EST (UTC-5)		YES
MASSACHUSETTS	MA	Boston	Boston	6,593,587	10,555	EST (UTC-5)		YES
MICHIGAN	MI	Lansing	Detroit	9,969,727	96,810	EST (UTC-5)	CST (UTC-6)	YES
MINNESOTA	MN	Saint Paul	Minneapolis	5,266,214	86,943	CST (UTC-6)		YES

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
MISSISSIPPI	MS	Jackson	Jackson	2,951,996	48,434	CST (UTC-6)		YES
MISSOURI	MO	Jefferson City	Kansas City	5,987,580	69,709	CST (UTC-6)		YES

Figure 1.4: Table example

Using a real database would probably increase the complexity of these simple examples, so we'll use a CSV file instead: united_states.csv[9] (see figure 1.5).

[9] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/data/united_states.csv

united_states.csv

```
name;abbr;capital;most populous city;population;square miles;time zone 1;time zone 2;dst
ALABAMA;AL;Montgomery;Birmingham;4,708,708;52,423;CST (UTC-6);EST (UTC-5);YES
ALASKA;AK;Juneau;Anchorage;698,473;656,425;AKST (UTC-09) ;HST (UTC-10) ;YES
ARIZONA;AZ;Phoenix;Phoenix;6,595,778;114,006;MT (UTC-07); ;NO
ARKANSAS;AR;Little Rock;Little Rock;2,889,450;53,182;CST (UTC-6); ;YES
CALIFORNIA;CA;Sacramento;Los Angeles;36,961,664;163,707;PT (UTC-8); ;YES
COLORADO;CO;Denver;Denver;5,024,748;104,100;MT (UTC-07); ;YES
CONNECTICUT;CT;Hartford;Bridgeport;3,518,288;5,544;EST (UTC-5); ;YES
DELAWARE;DE;Dover;Wilmington;885,122;1,954;EST (UTC-5); ;YES
FLORIDA;FL;Tallahassee;Jacksonville;18,537,969;65,758;EST (UTC-5);CST (UTC-6);YES
GEORGIA;GA;Atlanta;Atlanta;9,829,211;59,441;EST (UTC-5); ;YES
HAWAII;HI;Honolulu;Honolulu;1,295,178;10,932;HST (UTC-10) ; ;NO
IDAHO;ID;Boise;Boise;1,545,801;83,574;MT (UTC-07);PT (UTC-8);YES
ILLINOIS;IL;Springfield;Chicago;12,910,409;57,918;CST (UTC-6); ;YES
INDIANA;IN;Indianapolis;Indianapolis;6,423,113;36,420;EST (UTC-5);CST (UTC-6);YES
IOWA;IA;Des Moines;Des Moines;3,007,856;56,276;CST (UTC-6); ;YES
KANSAS;KS;Topeka;Wichita;2,818,747;82,282;CST (UTC-6);MT (UTC-07);YES
KENTUCKY;KY;Frankfort;Louisville;4,314,113;40,411;EST (UTC-5);CST (UTC-6);YES
LOUISIANA;LA;Baton Rouge;New Orleans;4,492,076;51,843;CST (UTC-6); ;YES
MAINE;ME;Augusta;Portland;1,318,301;35,387;EST (UTC-5); ;YES
MARYLAND;MD;Annapolis;Baltimore;5,699,478;12,407;EST (UTC-5); ;YES
MASSACHUSETTS;MA;Boston;Boston;6,593,587;10,555;EST (UTC-5); ;YES
```

Figure 1.5: United States CSV file

If you take a closer look at the boilerplate code in the UnitedStates[10] example, you'll discover that we've made a small change to the line that creates the `Document` (line 4). We've added an extra parameter that defines the size of the pages in the `Document`. The default page size is A4 and by default that page is used in portrait. In this example, we also use A4, but we rotate the page (`PageSize.A4.rotate()`) so that it is used in landscape as shown in Figure 1.4. We've also changed the margins (line 5). By default iText uses a margin of 36 user units (half an inch). We change all margins to 20 user units (this term will be explained further down in the text).

```
1  OutputStream fos = new FileOutputStream(dest);
2  PdfWriter writer = new PdfWriter(fos);
3  PdfDocument pdf = new PdfDocument(writer);
4  Document document = new Document(pdf, PageSize.A4.rotate());
5  document.setMargins(20, 20, 20, 20);
6  PdfFont font = PdfFontFactory.createFont(FontConstants.HELVETICA);
7  PdfFont bold = PdfFontFactory.createFont(FontConstants.HELVETICA_BOLD);
8  Table table = new Table(new float[]{4, 1, 3, 4, 3, 3, 3, 3, 1});
9  table.setWidthPercent(100);
10 BufferedReader br = new BufferedReader(new FileReader(DATA));
11 String line = br.readLine();
12 process(table, line, bold, true);
13 while ((line = br.readLine()) != null) {
14     process(table, line, font, false);
15 }
16 br.close();
17 document.add(table);
18 document.close();
```

[10] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1726-c01e04_unitedstates.java

In this example, we read this CSV file line by line, and we put all the data that is present in the CSV file in a `Table` object.

We start by creating two `PdfFont` objects of the same family: Helvetica regular (line 6) and Helvetica bold (line 7). We create a `Table` object for which we define nine columns by defining a `float` array with nine elements (line 8). Each `float` defines the relative width of a column. The first column is four times as wide as the second column; the third column is three times as wide as the second column; and so on. We also define the width of the table relative to the available width of the page (line 9). In this case, the table will use 100% of the width of the page, minus the page margins.

We then start to read the CSV file whose path is stored in the `DATA` constant (line 10). We read the first line to obtain the column headers (line 11) and we process that line (line 12). We wrote a `process()` method that adds the `line` to the `table` using a specific `font` and defining whether or not `line` contains the contents of a header row.

```
public void process(Table table, String line, PdfFont font, boolean isHeader) {
    StringTokenizer tokenizer = new StringTokenizer(line, ";");
    while (tokenizer.hasMoreTokens()) {
        if (isHeader) {
            table.addHeaderCell(
                new Cell().add(
                    new Paragraph(tokenizer.nextToken()).setFont(font)));
        } else {
            table.addCell(
                new Cell().add(
                    new Paragraph(tokenizer.nextToken()).setFont(font)));
        }
    }
}
```

We use a `StringTokenizer` to loop over all the fields that are stored in each line of our CSV file. We create a `Paragraph` in a specific `font`. We add that `Paragraph` to a new `Cell` object. Depending on whether or not we're dealing with a header row, we add this `Cell` to the `table` as a header cell or as an ordinary cell.

After we've processed the header row (line 12), we loop over the rest of the lines (line 13) and we process the rest of the rows (line 14). As you can tell from Figure 1.4, the table doesn't fit on a single page. There's no need to worry about that: iText will create as many new pages as necessary until the complete table was rendered. iText will also repeat the header row because we added the cells of that row using the `addHeaderCell()` method insead of using the `addCell()` method.

Once we've finished reading the data (line 15), we add the `table` to the document (line 17) and we close it (line 18). We have successfully published our CSV file as a PDF.

That was easy. With only a limited number of lines of code, we have already created quite a nice table in PDF.

Summary

With just a few examples, we have already seen a glimpse of the power of iText. We discovered that it's very easy to create a document programmatically. In this first chapter, we've discussed high-level objects such as `Paragraph`, `List`, `Image`, `Table` and `Cell`, which are iText's basic building blocks.

However, it's sometimes necessary to create a PDF with a lower-level syntax. iText makes this possible through its low-level API. We'll take a look at some examples that use these low-level methods in chapter 2.

Chapter 2: Adding low-level content

When we talk about *low-level content* in iText documentation, we always refer to PDF syntax that is written to a PDF content stream. PDF defines a series of operators such as `m` for which we created the `moveTo()` method in iText, `l` for which we created the `lineTo()` method, and `S` for which we created the `stroke()` method. By combining these operands in a PDF – or by combining these methods in iText – you can draw paths and shapes.

Let's take a look at a small example:

```
-406 0 m
406 0 l
S
```

This is PDF syntax that says: move to position (X = -406 ; Y = 0), then construct a path to position (X = 406 ; Y = 0); finally stroke that line - in this context, "stroking" means drawing. If we want to create this snippet of PDF syntax with iText, it goes like this:

```
1  canvas.moveTo(-406, 0)
2              .lineTo(406, 0)
3              .stroke();
```

That looks easy, doesn't it? But what is that `canvas` object we're using? Let's take a look at a couple examples to find out.

Drawing lines on a canvas

Suppose that we would like to create a PDF that looks like Figure 2.1.

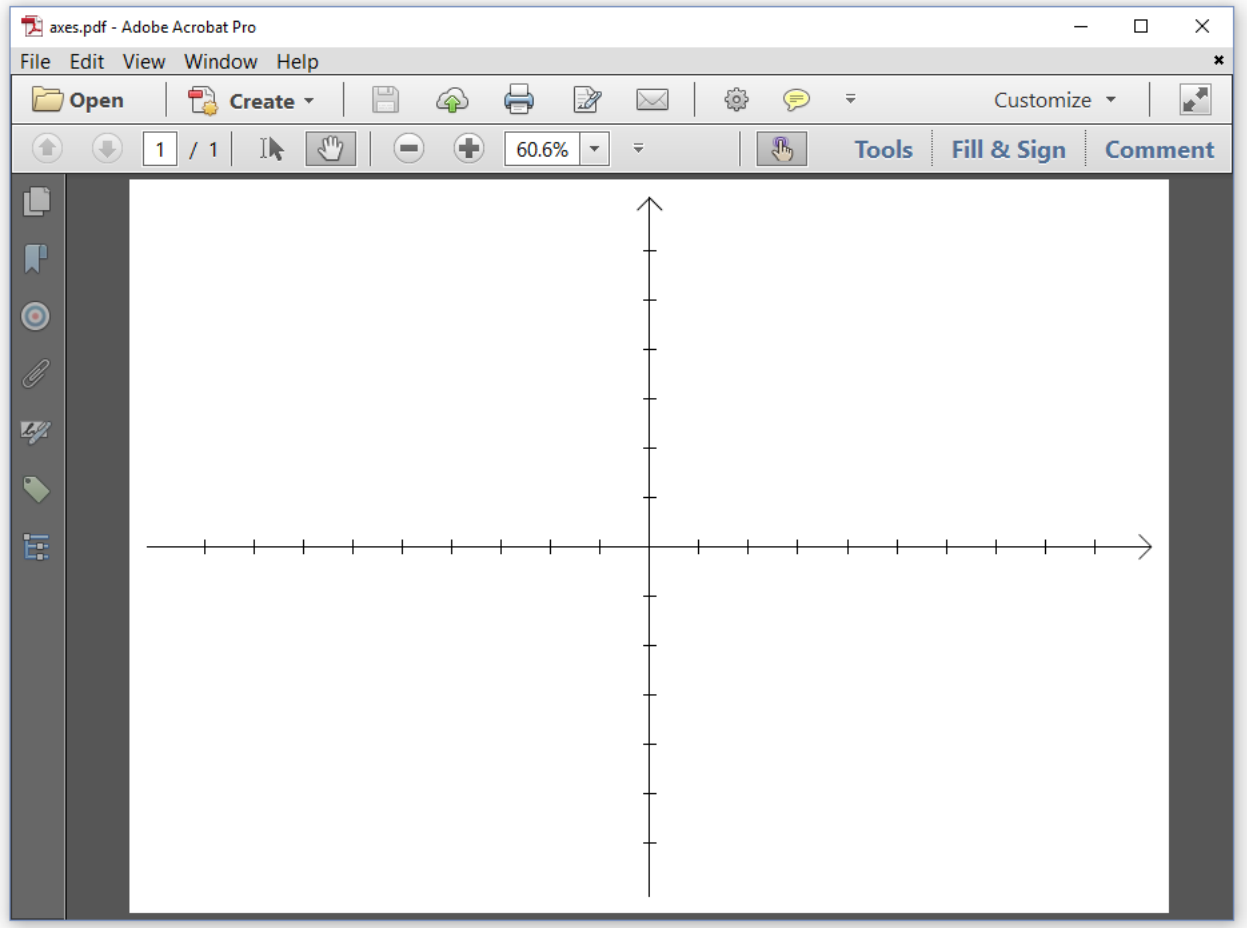


Figure 2.1: drawing an X and Y axis

This PDF showing an X and Y axis was created with the Axes[11] example. Let's examine this example step by step.

```
1 OutputStream fos = new FileOutputStream(dest);
2 PdfWriter writer = new PdfWriter(fos);
3 PdfDocument pdf = new PdfDocument(writer);
4 PageSize ps = PageSize.A4.rotate();
5 PdfPage page = pdf.addNewPage(ps);
6 PdfCanvas canvas = new PdfCanvas(page);
7 // Draw the axes
8 pdf.close();
```

The first thing that jumps out is that we no longer use a `Document` object. Just like in the previous chapter, we create an `OutputStream` (line 1), a `PdfWriter` (line 2) and a `PdfDocument` (line 3) but instead of creating a `Document` with a default or specific page size, we create a `PdfPage` (line 5) with

[11] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-2#1734-c02e01_axes.java

a specific `PageSize` (line 4). In this case, we use an A4 page with landscape orientation. Once we have a `PdfPage` instance, we use it to create a `PdfCanvas` (line 6). We'll use this `canvas` objects to create a sequence of PDF operators and operands. As soon as we've finished drawing and painting whatever paths and shapes we want to add to the page, we close the `PdfDocument` (line 8).

In the previous chapter, we closed the `Document` object with `document.close();` This implicitly closed the `PdfDocument` object. Now that there is no `Document` object, we have to close the `PdfDocument` object.

In PDF, all measurements are done in user units. By default one user unit corresponds with one point. This means that there are 72 user units in one inch. In PDF, the X axis points to the right and the Y axis points upwards. If you use the `PageSize` object to create the page size, the origin of the coordinate system is located in the lower-left corner of the page. All the coordinates that we use as operands for operators such as `m` or `l` use this coordinate system. We can change the coordinate system by changing the *current transformation matrix*.

The coordinate system and the transformation matrix

If you've followed a class in analytical geometry, you know that you can move objects around in space by applying a transformation matrix. In PDF, we don't move the objects, but we move the coordinate system and we draw the objects in the new coordinate system. Suppose that we want to move the coordinate system in such a way that the origin of the coordinate system is positioned in the exact middle of the page. In that case, we'd need to use the `concatMatrix()` method:

```
canvas.concatMatrix(1, 0, 0, 1, ps.getWidth() / 2, ps.getHeight() / 2);
```

The parameters of the `concatMatrix()` method are elements of a transformation matrix. This matrix consists of three columns and three rows:

```
a   b   0
c   d   0
e   f   1
```

The values of the elements in the third column are always fixed (`0`, `0`, and `1`), because we're working in a two dimensional space. The values `a`, `b`, `c`, and `d` can be used to scale, rotate, and skew the coordinate system. There is no reason why we are confined to a coordinate system where the axes are orthogonal or where the progress in the X direction needs to be identical to the progress in the Y direction. But let's keep things simple and use `1`, `0`, `0`, and `1` as values for `a`, `b`, `c`, and `d`. The elements `e` and `f` define the translation. We take the page size `ps` and we divide its width and height by two to get the values for `e` and `f`.

The graphics state

The current transformation matrix is part of the graphics state of the page. Other values that are defined in the graphics state are the line width, the stroke color (for lines), the fill color (for shapes), and so on. In another tutorial, we'll go in more depth, describing each value of the graphics state in great detail. For now it's sufficient to know that the default line width is 1 user unit and that the default stroke color is black. Let's draw those axes we saw in Figure 2.1:

```
1  //Draw X axis
2  canvas.moveTo(-(ps.getWidth() / 2 - 15), 0)
3          .lineTo(ps.getWidth() / 2 - 15, 0)
4          .stroke();
5  //Draw X axis arrow
6  canvas.setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.ROUND)
7          .moveTo(ps.getWidth() / 2 - 25, -10)
8          .lineTo(ps.getWidth() / 2 - 15, 0)
9          .lineTo(ps.getWidth() / 2 - 25, 10).stroke()
10         .setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.MITER);
11 //Draw Y axis
12 canvas.moveTo(0, -(ps.getHeight() / 2 - 15))
13         .lineTo(0, ps.getHeight() / 2 - 15)
14         .stroke();
15 //Draw Y axis arrow
16 canvas.saveState()
17         .setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.ROUND)
18         .moveTo(-10, ps.getHeight() / 2 - 25)
19         .lineTo(0, ps.getHeight() / 2 - 15)
20         .lineTo(10, ps.getHeight() / 2 - 25).stroke()
21         .restoreState();
22 //Draw X serif
23 for (int i = -((int) ps.getWidth() / 2 - 61);
24     i < ((int) ps.getWidth() / 2 - 60); i += 40) {
25     canvas.moveTo(i, 5).lineTo(i, -5);
26 }
27 //Draw Y serif
28 for (int j = -((int) ps.getHeight() / 2 - 57);
29     j < ((int) ps.getHeight() / 2 - 56); j += 40) {
30     canvas.moveTo(5, j).lineTo(-5, j);
31 }
32 canvas.stroke();
```

This code snippet consists of different parts:

- Lines 2-4 and 12-14 shouldn't have any secrets for you anymore. We move to a coordinate, we construct a line to another coordinate, and we stroke the line.
- Lines 6-10 draw two lines that are connected to each other. There are possible ways to draw that connection: *miter* (the lines join in a sharp point), *bevel* (the corner is beveled) and *round* (the corner is rounded). We want the corner to be rounded, so we change the default line join value (which is `MITER`) to `ROUND`. We construct the path of the arrow with one `moveTo()` and two `lineTwo()` invocations, and we change the line join value back to the default. Although the graphics state is now back to its original value, this isn't the best way to return to a previous graphics state.
- Lines 16-21 show a better practice we should use whenever we change the graphics state. First we save the current graphics state with the `saveState()` method, then we change the state and draw whatever lines or shapes we want to draw, finally, we use the `restoreState()` method to return to the original graphics state. All the changes that we applied after `saveState()` will be undone. This is especially interesting if you change multiple values (line width, color,…) or when it's difficult to calculate the reverse change (returning to the original coordinate system).
- In lines 23-31, we construct small serifs to be drawn on both axes every 40 user units. Observe that we don't stroke them immediately. Only when we've constructed the complete path, we call the `stroke()` method.

There's usually more than one way to draw lines and shapes to the canvas. It would lead us too far to explain the advantages and disadvantages of different approaches with respect to the speed of production of the PDF file, the impact on the file size, and the speed of rendering the document in a PDF viewer. That's something that needs to be further discussed in another tutorial.

There are also specific rules that need to be taken into account. For instance: sequences of `saveState()` and `restoreState()` need to be balanced. Every `saveState()` needs a `restoreState()`; it's forbidden to have a `restoreState()` that wasn't preceded by a `saveState()`.

For now let's adapt the first example of this chapter by changing line widths, introducing a dash pattern, and applying different stroke colors so that we get a PDF as shown in Figure 2.2.

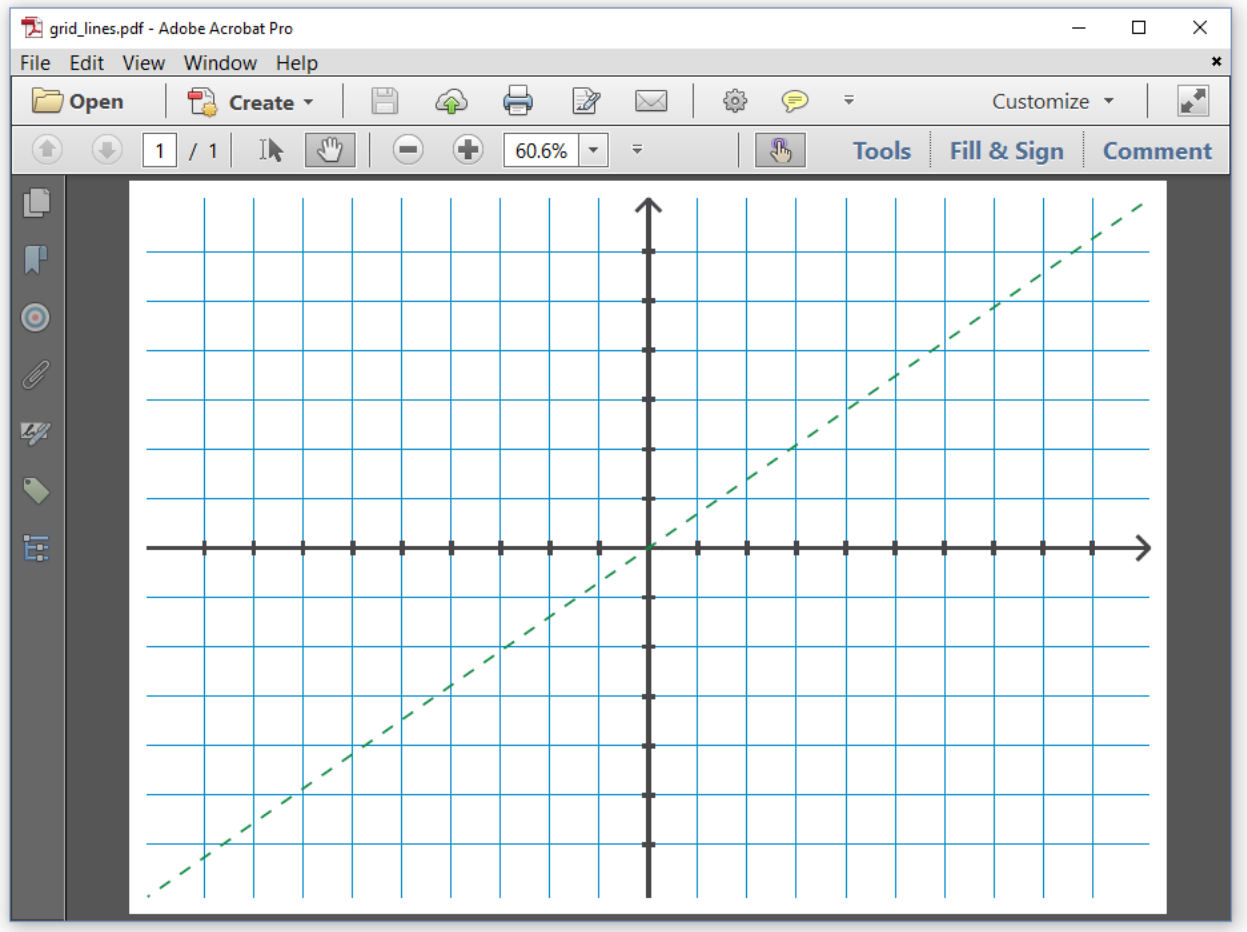


Figure 2.2: drawing a grid

In the GridLines[12] example, we first define a series of `Color` objects:

```
1  Color grayColor = new DeviceCmyk(0.f, 0.f, 0.f, 0.875f);
2  Color greenColor = new DeviceCmyk(1.f, 0.f, 1.f, 0.176f);
3  Color blueColor = new DeviceCmyk(1.f, 0.156f, 0.f, 0.118f);
```

The PDF specification (ISO-32000) defines many different color spaces, each of which has been implemented in a separate class in iText. The most commonly used color spaces are `DeviceGray` (a color defined by a single *intensity* parameter), `DeviceRgb` (defined by three parameters: red, green, and blue) and `DeviceCmyk` (defined by four parameters: cyan, magenta, yellow and black). In our example, we use three CMYK colors.

Be aware that we're not working with the `java.awt.Color` class. We're working with iText's `Color` class that can be found in the `com.itextpdf.kernel.color` package.

[12] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-2#1735-c02e02_gridlines.java

We want to create a grid that consists of thin blue lines:

```
1  canvas.setLineWidth(0.5f).setStrokeColor(blueColor);
2  for (int i = -((int) ps.getHeight() / 2 - 57);
3      i < ((int) ps.getHeight() / 2 - 56); i += 40) {
4      canvas.moveTo(-(ps.getWidth() / 2 - 15), i)
5              .lineTo(ps.getWidth() / 2 - 15, i);
6  }
7  for (int j = -((int) ps.getWidth() / 2 - 61);
8      j < ((int) ps.getWidth() / 2 - 60); j += 40) {
9      canvas.moveTo(j, -(ps.getHeight() / 2 - 15))
10             .lineTo(j, ps.getHeight() / 2 - 15);
11 }
12 canvas.stroke();
```

In line 1, we set the line width to half a user unit and the color to blue. In lines 2-10, we construct the paths of the grid lines, and we stroke them in line 11.

We reuse the code to draw the axes from the previous example, but we let them precede by a line that changes the line width and stroke color.

```
canvas.setLineWidth(3).setStrokeColor(grayColor);
```

After we've drawn the axes, we draw a dashed green line that is 2 user units wide:

```
1  canvas.setLineWidth(2).setStrokeColor(greenColor)
2          .setLineDash(10, 10, 8)
3          .moveTo(-(ps.getWidth() / 2 - 15), -(ps.getHeight() / 2 - 15))
4          .lineTo(ps.getWidth() / 2 - 15, ps.getHeight() / 2 - 15).stroke();
```

There are many possible variations to define a line dash, but in this case, we are defining the line dash using three parameters. The length of the dash is 10 user units; the length of the gap is 10 user units; the phase is 8 user units —the phase defines the distance in the dash pattern to start the dash.

Feel free to experiment with some of the other methods that are available in the `PdfCanvas` class. You can construct curves with the `curveTo()` method, rectangles with the `rectangle()` method, and so on. Instead of stroking paths with the `stroke()` method using the stroke color, you can also fill paths with the `fill()` method using the fill color. The `PdfCanvas` class offers much more than a Java version of the PDF operators. It also introduces a number of convenience classes to construct specific paths for which there are no operators available in PDF, such as ellipses or circles.

In our next example, we'll look at a subset of the graphics state that will allow us to add text at absolute positions.

The text state

In Figure 2.3, we see the opening titles of Episode V of Star Wars: The Empire Strikes Back.

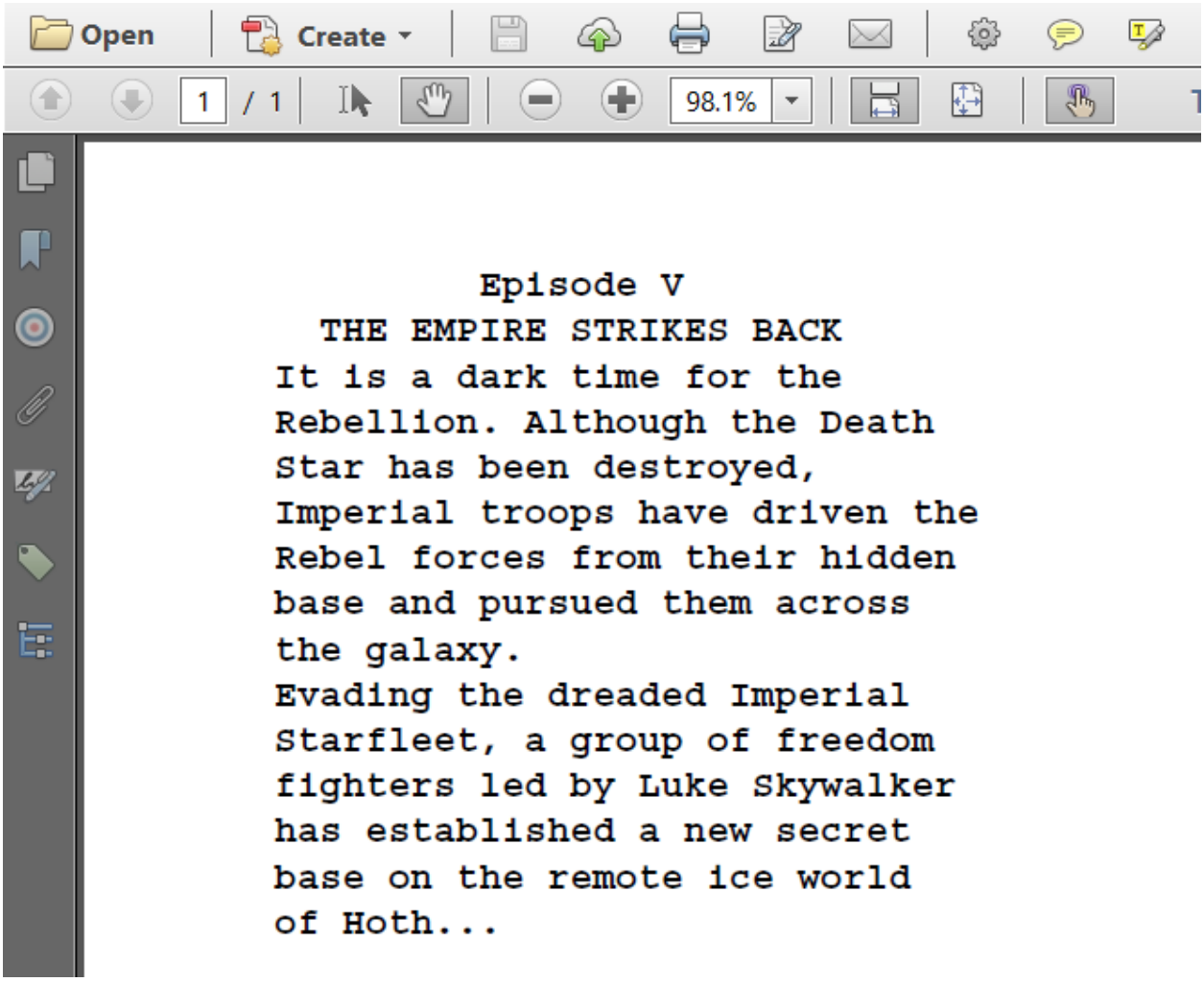


Figure 2.3: adding text at absolute positions

The best way to create such a PDF, would be to use a sequence of `Paragraph` objects with different alignments –center for the title; left aligned for the body text), and to add these paragraphs to a `Document` object. Using the high-level approach will distribute the text over several lines, introducing line breaks automatically if the content doesn't fit the width of the page, and page breaks if the remaining content doesn't fit the height of the page.

All of this doesn't happen when we add text using low-level methods. We need to break up the content into small chunks of text ourselves as is done in the StarWars[13] example:

[13] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-2#1736-c02e03_starwars.java

```
List<String> text = new ArrayList();
text.add("         Episode V         ");
text.add("  THE EMPIRE STRIKES BACK  ");
text.add("It is a dark time for the");
text.add("Rebellion. Although the Death");
text.add("Star has been destroyed,");
text.add("Imperial troops have driven the");
text.add("Rebel forces from their hidden");
text.add("base and pursued them across");
text.add("the galaxy.");
text.add("Evading the dreaded Imperial");
text.add("Starfleet, a group of freedom");
text.add("fighters led by Luke Skywalker");
text.add("has established a new secret");
text.add("base on the remote ice world");
text.add("of Hoth...");
```

For reasons of convenience, we change the coordinate system so that its origin lies in the top-left corner instead of the bottom-left corner. We then create a text object with the `beginText()` method, and we change the text state:

```
canvas.concatMatrix(1, 0, 0, 1, 0, ps.getHeight());
canvas.beginText()
    .setFontAndSize(PdfFontFactory.createFont(FontConstants.COURIER_BOLD), 14)
    .setLeading(14 * 1.2f)
    .moveText(70, -40);
```

We create a `PdfFont` to show the text in Courier Bold and we change the text state so that all text that is drawn will use this font with font size 14. We also define a leading of 1.2 times this font size. The leading is the distance between the baselines of two subsequent lines of text. Finally, we change the text matrix so that the cursor moves 70 user units to the right and 40 user units down.

Next, we loop over the different `String` values in our `text` list, show each `String` on a new line –moving the cursor down 16.2 user units (this is the leading)—, and we close the text object with the `endText()` method.

```
for (String s : text) {
    //Add text and move to the next line
    canvas.newlineShowText(s);
}
canvas.endText();
```

It's important not to show any text outside of a text object –which is delimited by the `beginText()`/`endText()` methods. It's also forbidden to nest `beginText()`/`endText()` sequences.

What if we pimped this example and changed it in such a way that it produces the PDF shown in figure 2.4?

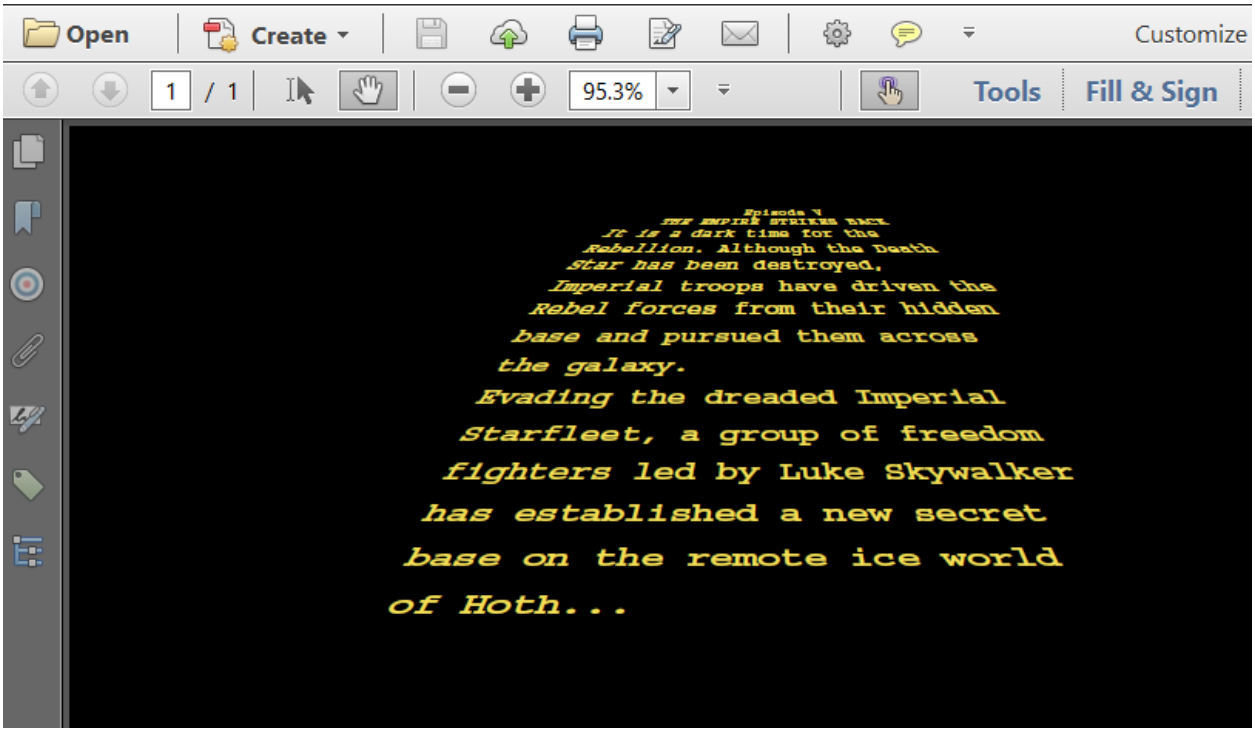


Figure 2.4: adding skewed and colored text at absolute positions

Changing the color of the background is the easy part in the StarWarsCrawl[14] example:

```
canvas.rectangle(0, 0, ps.getWidth(), ps.getHeight())
        .setColor(Color.BLACK, true)
        .fill();
```

We create a rectangle of which the lower-left corner has the coordinate X = 0, Y = 0, and of which the width and the height correspond with the width and the height of the page size. We set the fill color to black. We could have used `setFillColor(Color.BLACK)`, but we used the more generic `setColor()` method instead. The boolean indicates if we want to change the stroke color (`false`) or the fill color (`true`). Finally, we fill that path of the rectangle using the fill color as paint.

Now comes the less trivial part of the code: how do we add the text?

[14] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-2#1737-c02e04_starwarscrawl.java

```
1  canvas.concatMatrix(1, 0, 0, 1, 0, ps.getHeight());
2  Color yellowColor = new DeviceCmyk(0.f, 0.0537f, 0.769f, 0.051f);
3  float lineHeight = 5;
4  float yOffset = -40;
5  canvas.beginText()
6      .setFontAndSize(PdfFontFactory.createFont(FontConstants.COURIER_BOLD), 1)
7      .setColor(yellowColor, true);
8  for (int j = 0; j < text.size(); j++) {
9      String line = text.get(j);
10     float xOffset = ps.getWidth() / 2 - 45 - 8 * j;
11     float fontSizeCoeff = 6 + j;
12     float lineSpacing = (lineHeight + j) * j / 1.5f;
13     int stringWidth = line.length();
14     for (int i = 0; i < stringWidth; i++) {
15         float angle = (maxStringWidth / 2 - i) / 2f;
16         float charXOffset = (4 + (float) j / 2) * i;
17         canvas.setTextMatrix(fontSizeCoeff, 0,
18                 angle, fontSizeCoeff / 1.5f,
19                 xOffset + charXOffset, yOffset - lineSpacing)
20             .showText(String.valueOf(line.charAt(i)));
21     }
22 }
23 canvas.endText();
```

Once more, we change the origin of the coordinate system to the top of the page (line 1). We define a CMYK color for the text (line 2). We initialize a value for the line height (line 3) and the offset in the Y-direction (line 4). We begin writing a text object. We'll use Courier Bold as font and define a font size of 1 user unit (line 6). The font size is only 1, but we'll scale the text to a readable size by changing the text matrix. We don't define a leading; we won't need a leading because we won't use `newlineShowText()`. Instead we'll calculate the starting position of each individual character, and draw the text character by character. We also introduce a fill color (line 7).

Every glyph in a font is defined as a path. By default, the paths of the glyphs that make up a text are filled. That's why we set the fill color to change the color of the text.

We start looping over the `text` (line 8) and we read each line into a `String` (line 9). We'll need plenty of Math to define the different elements of the text matrix that will be used to position each glyph. We define an `xOffset` for every line (line 10). Our font size was defined as 1 user unit, but we'll multiply it with a `fontSizeCoeff` that will depend on the index of the line in the `text` array (line 11). We'll also define where the line will start relative to the `yOffset` (12).

We calculate the number of characters in each line (line 13) and we loop over all the characters

(line 14). We define an angle depending on the position of the character in the line (line 15). The `charOffset` depends on both the index of the line and the position of the character (line 16).

Now we're ready to set the text matrix (line 17-19). Parameter `a` and `d` define the scaling factors. We'll use them to change the font size. With parameter `c`, we introduce a skew factor. Finally, we calculate the coordinate of the character to determine the parameter `e` and `f`. Now that the exact position of the character is determined, we show the character using the `showText()` method (line 20). This method doesn't introduce any new lines. Once we've finished looping over all the characters in all the lines, we close the text object with the `endText()` method (line 23).

If you think this example was rather complex, you are absolutely right. I used it just to show that iText allows you to create content in whatever way you want. If it's possible in PDF, it's possible with iText. But rest assured, the upcoming examples will be much easier to understand.

Summary

In this chapter, we've been experimenting with PDF operators and operands and the corresponding iText methods. We've learned about a concept called graphics state that keeps track of properties such as the current transformation matrix, line width, color, and so on. Text state is a subset of the graphics state that covers all the properties that are related to text, such as the text matrix, the font and size of the text, and many other properties we haven't discussed yet. We'll get into much more detail in another tutorial.

One might wonder why a developer would need access to the low-level API knowing that there's so much high-level functionality in iText. That question will be answered in the next chapter.

Chapter 3: Using renderers and event handlers

In the first chapter of this tutorial, we created a `Document` with a certain page size and certain margins (defined explicitly or implicitly) and when we added basic building blocks such as `Paragraphs` and `Lists` to that `document` object. iText made sure that the content was nicely organized on the page. We also created a `Table` object to publish the contents of a CSV file and the result already looked nice. But what if all of this isn't sufficient? What if we want more control over how the content is laid out on the page? What if you're not happy with the rectangular borders that are drawn by the `Table` class? What if you want to add content that appears on a specific location on every page, no matter how many pages are created?

Should you draw all the content at absolute positions as was explained in the second chapter to meet these specific requirements? By playing with the Star Wards examples, we've experienced that this could lead to code that is quite complex (and code that is hard to maintain). Surely there must be way to combine the high-level approach using the basic building blocks with a more low-level approach that allows us to have more influence on the layout. That's what this third chapter is about.

Introducing a document renderer

Suppose that we want to add text and images to a document, but we don't want the text to span the complete width of the page. Instead, we want to organize the content in three columns as shown in Figure 3.1.

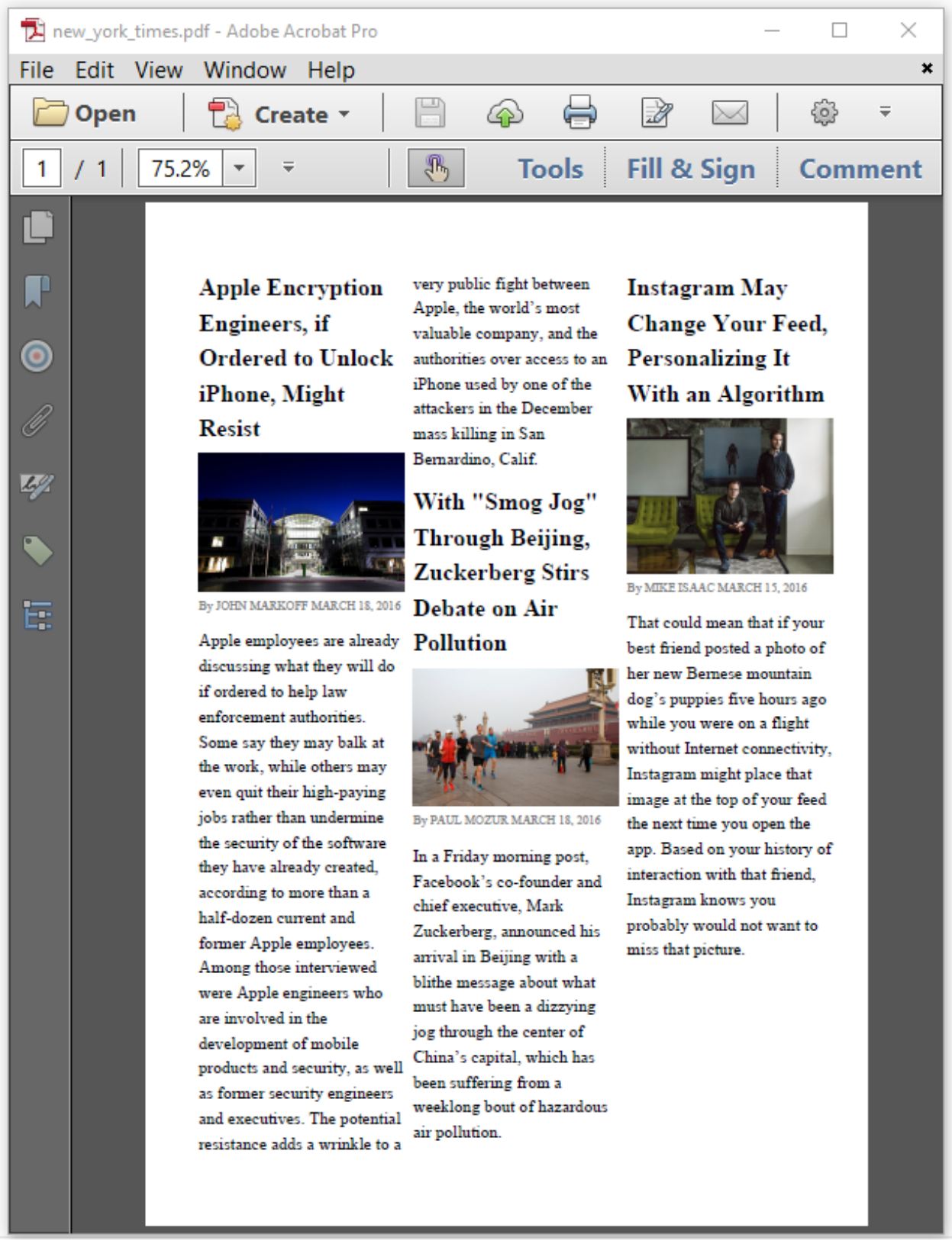


Figure 3.1: Text and images organized in columns

The NewYorkTimes[15] example shows how this is done.

```
1  OutputStream fos = new FileOutputStream(dest);
2  PdfWriter writer = new PdfWriter(fos);
3  PdfDocument pdf = new PdfDocument(writer);
4  PageSize ps = PageSize.A5;
5  Document document = new Document(pdf, ps);
6  //Set column parameters
7  float offSet = 36;
8  float columnWidth = (ps.getWidth() - offSet * 2 + 10) / 3;
9  float columnHeight = ps.getHeight() - offSet * 2;
10 //Define column areas
11 Rectangle[] columns = {
12     new Rectangle(offSet - 5, offSet, columnWidth, columnHeight),
13     new Rectangle(offSet + columnWidth, offSet, columnWidth, columnHeight),
14     new Rectangle(
15         offSet + columnWidth * 2 + 5, offSet, columnWidth, columnHeight)};
16 document.setRenderer(new ColumnDocumentRenderer(document, columns));
17 // adding content
18 Image inst = new Image(ImageFactory.getImage(INST_IMG)).setWidth(columnWidth);
19 String articleInstagram = new String(
20     Files.readAllBytes(Paths.get(INST_TXT)), StandardCharsets.UTF_8);
21 NewYorkTimes.addArticle(document,
22     "Instagram May Change Your Feed, Personalizing It With an Algorithm",
23     "By MIKE ISAAC MARCH 15, 2016", inst, articleInstagram);
24 doc.close();
```

The first five lines are pretty standard. We recognize them from chapter 1. In line 7, 8, and 9, we define a couple of parameters:

- an `offSet` that will be used to define the top and bottom margin, as well as for the right and left margin.
- the width of each column, `columnWidth`, that is calculated by dividing the available page width by three (because we want three columns). The available page width is the full page width minus the left and right margin, minus two times 5 user units that we will deduct from the margin so that we have a small gutter between the columns.
- the height of each column, `columnHeight`, that is calculated by subtracting the top and the bottom margin from the full page height.

We use these values to define three `Rectangle` objects:

[15] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-3#1742-c03e01_newyorktimes.java

- a `Rectangle` of which the coordinate of the lower-left corner is (X = `offSet - 5`, Y = `offSet`), width `columnWidth` and height `columnHeight`,
- a `Rectangle` of which the coordinate of the lower-left corner is (X = `offSet + columnWidth`, Y = `offSet`), width `columnWidth` and height `columnHeight`, and
- a `Rectangle` of which the coordinate of the lower-left corner is (X = `offSet + columnWidth * 2 + 5`, Y = `offSet`, width `columnWidth` and height `columnHeight`.

We put these three `Rectangle` objects in an array named `columns`, and we use it to create a `ColumnDocumentRenderer`. Once we declare this `ColumnDocumentRenderer` as the `DocumentRenderer` for our `Document` instance, all the content that we add to `document` will be laid out in columns that correspond with the `Rectangles` we've defined.

In line 18, we create an `Image` object and we scale the image so that it fits the width of a column. In line 19 and 20, we read a text file into a `String`. We use these objects as parameters for a custom `addArticle()` method.

```
1  public static void addArticle(
2      Document doc, String title, String author, Image img, String text)
3      throws IOException {
4      Paragraph p1 = new Paragraph(title)
5              .setFont(timesNewRomanBold)
6              .setFontSize(14);
7      doc.add(p1);
8      doc.add(img);
9      Paragraph p2 = new Paragraph()
10             .setFont(timesNewRoman)
11             .setFontSize(7)
12             .setFontColor(Color.GRAY)
13             .add(author);
14     doc.add(p2);
15     Paragraph p3 = new Paragraph()
16             .setFont(timesNewRoman)
17             .setFontSize(10)
18             .add(text);
19     doc.add(p3);
20 }
```

No new concepts were introduced in this code snippet. The objects `timesNewRoman` and `timesNewRomanBold` were defined as `static PdfFont` member-variables of the `NewYorkTimes`[16] class. That's much easier than what we did in the previous chapter, isn't it? Let's continue with an example that is a tad more complex.

[16] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-3#1742-c03e01_newyorktimes.java

Applying a block renderer

When we published the contents of a CSV file listing information about the states of the USA, we created a series of `Cell` objects that we added to a `Table` object. We didn't define a background color, nor did we define what the borders should look like. The default values were used.

By default, a cell doesn't have a background color. Its borders consist of a black rectangle with a line width of 0.5 user units.

Now, we'll take another data source, premier_league.csv[17], and we'll put that data in a `Table` that we'll spice up a little; see Figure 3.2.

POS	CLUB	Played	Won	Drawn	Lost	Goals For	Goals against	Goal Difference	Points
1	Leicester City	30	18	9	3	53	31	22	63
2	Tottenham Hotspur	30	16	10	4	53	24	29	58
3	Arsenal	29	15	7	7	46	30	16	52
4	Manchester City	29	15	6	8	52	31	21	51
5	West Ham United	29	13	10	6	45	33	12	49
6	Manchester United	29	13	8	8	37	27	10	47
7	Southampton	30	12	8	10	38	30	8	44
8	Liverpool	28	12	8	8	43	37	6	44
9	Stoke City	30	12	7	11	32	36	-4	43
10	Chelsea	29	10	10	9	43	39	4	40
11	West Bromwich Albion	29	10	9	10	30	36	-6	39
12	Everton	28	9	11	8	51	39	12	38
13	Bournemouth	30	10	8	12	38	47	-9	38
14	Watford	29	10	7	12	29	30	-1	37
15	Crystal Palace	29	9	6	14	32	39	-7	33
16	Swansea City	30	8	9	13	30	40	-10	33
17	Sunderland	29	6	7	16	35	54	-19	25
18	Norwich City	30	6	7	17	31	54	-23	25
19	Newcastle United	29	6	6	17	28	54	-26	24
20	Aston Villa	30	3	7	20	22	57	-35	16

Figure 3.2: a table with colored cells and rounded borders

[17] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/data/premier_league.csv

We won't repeat the boiler-plate code because it's identical to what we had in the previous example, except for one line:

```
PageSize ps = new PageSize(842, 680);
```

Before, we always used a standard paper format such as `PageSize.A4`. In this case, we use a user-defined page size of 842 by 680 user units (17.7 x 9.4 in). The body of the PremierLeague[18] example looks pretty straight-forward too.

```
1  PdfFont font = PdfFontFactory.createFont(FontConstants.HELVETICA);
2  PdfFont bold = PdfFontFactory.createFont(FontConstants.HELVETICA_BOLD);
3  Table table = new Table(new float[]{1.5f, 7, 2, 2, 2, 2, 3, 4, 4, 2});
4  table.setWidthPercent(100)
5          .setTextAlignment(Property.TextAlignment.CENTER)
6          .setHorizontalAlignment(Property.HorizontalAlignment.CENTER);
7  BufferedReader br = new BufferedReader(new FileReader(DATA));
8  String line = br.readLine();
9  process(table, line, bold, true);
10 while ((line = br.readLine()) != null) {
11     process(table, line, font, false);
12 }
13 br.close();
14 document.add(table);
```

There are only some minor differences when compared with the UnitedStates[19] example. In this example, we set the text alignment of the content of the `Table` in such a way that it is centered. We also change the horizontal alignment of the table itself –note that this doesn't matter much as the table takes up 100% of the available width anyway. The `process()` method is more interesting.

```
1  public void process(Table table, String line, PdfFont font, boolean isHeader) {
2      StringTokenizer tokenizer = new StringTokenizer(line, ";");
3      int columnNumber = 0;
4      while (tokenizer.hasMoreTokens()) {
5          if (isHeader) {
6              Cell cell = new Cell().add(new Paragraph(tokenizer.nextToken()));
7              cell.setNextRenderer(new RoundedCornersCellRenderer(cell));
8              cell.setPadding(5).setBorder(null);
9              table.addHeaderCell(cell);
10         } else {
```

[18] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-3#1743-c03e02_premierleague.java
[19] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1726-c01e04_unitedstates.java

```
11              columnNumber++;
12              Cell cell = new Cell().add(new Paragraph(tokenizer.nextToken()));
13              cell.setFont(font).setBorder(new SolidBorder(Color.BLACK, 0.5f));
14              switch (columnNumber) {
15                  case 4:
16                      cell.setBackgroundColor(greenColor);
17                      break;
18                  case 5:
19                      cell.setBackgroundColor(yellowColor);
20                      break;
21                  case 6:
22                      cell.setBackgroundColor(redColor);
23                      break;
24                  default:
25                      cell.setBackgroundColor(blueColor);
26                      break;
27              }
28              table.addCell(cell);
29          }
30      }
31  }
```

Let's start with the ordinary cells. In lines 16, 19, 22, and 25, we change the background color based on the column number.

In line 13, we set the font of the content of the `Cell` and we overrule the default border using the `setBorder()` method. We define the border as a black solid border with a 0.5 user unit line width.

The `SolidBorder` class extends the `Border` class; it has siblings such as `DashedBorder`, `DottedBorder`, `DoubleBorder`, and so on. If iText doesn't provide you with the border of your choice, you can either extend the `Border` class –you can use the existing implementations for inspiration–, or you can create your own `CellRenderer` implementation.

We use a custom `RoundedCornersCellRenderer()` in line 7. In line 8, we define a padding for the cell content, and we set the border to `null`. If `setBorder(null)` wasn't there, two borders would be drawn: one by iText, one by the cell renderer that we're about to examine.

```
1  private class RoundedCornersCellRenderer extends CellRenderer {
2      public RoundedCornersCellRenderer(Cell modelElement) {
3          super(modelElement);
4      }
5
6      @Override
7      public void drawBorder(DrawContext drawContext) {
8          Rectangle rectangle = getOccupiedAreaBBox();
9          float llx = rectangle.getX() + 1;
10         float lly = rectangle.getY() + 1;
11         float urx = rectangle.getX() + getOccupiedAreaBBox().getWidth() - 1;
12         float ury = rectangle.getY() + getOccupiedAreaBBox().getHeight() - 1;
13         PdfCanvas canvas = drawContext.getCanvas();
14         float r = 4;
15         float b = 0.4477f;
16         canvas.moveTo(llx, lly).lineTo(urx, lly).lineTo(urx, ury - r)
17                 .curveTo(urx, ury - r * b, urx - r * b, ury, urx - r, ury)
18                 .lineTo(llx + r, ury)
19                 .curveTo(llx + r * b, ury, llx, ury - r * b, llx, ury - r)
20                 .lineTo(llx, lly).stroke();
21         super.drawBorder(drawContext);
22     }
23 }
```

The `CellRenderer` class is a special implementation of the `BlockRenderer` class.

The `BlockRenderer` class can be used on `BlockElements` such as `Paragraph` and `List`. These renderer classes allow you to create custom functionality by overriding the `draw()` method. For instance: you could create a custom background for a `Paragraph`. The `CellRenderer` also has a `drawBorder()` method.

We override the `drawBorder()` method to draw a rectangle that is rounded at the top (line 6-21). The `getOccupiedAreaBBox()` method returns a `Rectangle` object that we can use to find the *bounding box* of the `BlockElement` (line 8). We use the `getX()`, `getY()`, `getWidth()`, and `getHeight()` method to define the coordinates of the lower-left and upper-right corner of the `Cell` (line 9-12).

The `drawContext()` parameter gives us access to the `PdfCanvas` instance (line 13). We draw the border as a sequence of lines and curves (line 14-20). This example demonstrates how the high-level approach using a `Table` consisting of `Cells` nicely ties in with the low-level approach where we create PDF syntax almost manually to draw a border that meets our needs exactly.

The code that draws the curves requires some knowledge about Math, but it's not rocket science. Most of the common types of borders are covered by iText so that you don't really need to worry about all the Math that takes place on under the hood.

There's much more to say about `BlockRenderers`, but we'll save that for another tutorial. We'll finish this chapter with an example that demonstrates how we can automatically add backgrounds, headers (or footers), watermarks, and a page number to every page that is created.

Handling events

When we add a `Table` with many rows to a document, there's a high chance that this table will be distributed over different pages. In Figure 3.3, we see a list of UFO sightings stored in ufo.csv[20]. The background of every odd page is colored lime; the background of every even page is blue. There's a header saying "THE TRUTH IS OUT THERE" and a watermark saying "CONFIDENTIAL" under the actual page content. Centered at the bottom of every page, there's a page number.

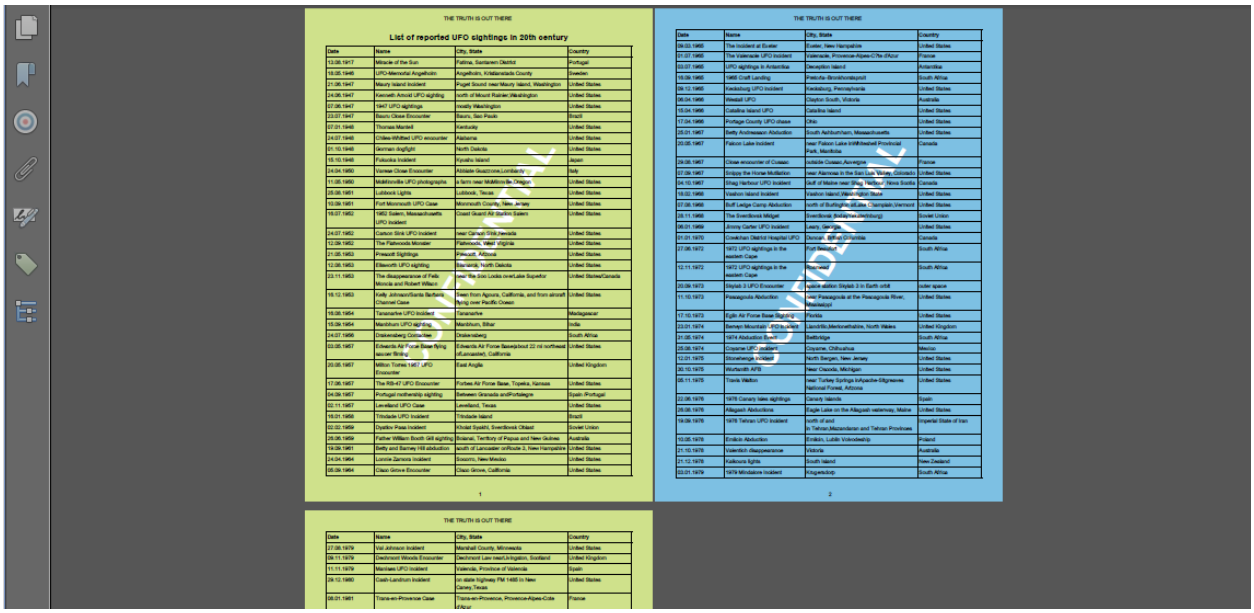


Figure 3.3: repeating background color and watermark

You've seen the code that creates the table in the UFO[21] example already a couple of times.

[20] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/data/ufo.csv

[21] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-3#1744-c03e03_ufo.java

```
1  OutputStream fos = new FileOutputStream(dest);
2  PdfWriter writer = new PdfWriter(fos);
3  PdfDocument pdf = new PdfDocument(writer);
4  pdf.addEventHandler(PdfDocumentEvent.END_PAGE, new MyEventHandler());
5  Document document = new Document(pdf);
6  Paragraph p = new Paragraph("List of reported UFO sightings in 20th century")
7          .setTextAlignment(Property.TextAlignment.CENTER)
8          .setFont(helveticaBold).setFontSize(14);
9  document.add(p);
10 Table table = new Table(new float[]{3, 5, 7, 4});
11 table.setWidthPercent(100);
12 BufferedReader br = new BufferedReader(new FileReader(DATA));
13 String line = br.readLine();
14 process(table, line, helveticaBold, true);
15 while ((line = br.readLine()) != null) {
16     process(table, line, helvetica, false);
17 }
18 br.close();
19 document.add(table);
20 document.close();
```

In the body of this snippet, we add a `Paragraph` that is centered by setting the text alignment to `Property.TextAlignment.CENTER`. We loop over a CSV file (`DATA`) and we process each line the same way we've already processed other lines taken from CSV files.

Line 4 is of special interest to us in the context of this example. We add an event handler `MyEventHandler` to the `PdfDocument`. Our `MyEventHandler` implementation implements `IEventHandler`, an interface with a single method: `handleEvent()`. This method will be triggered every time an event of type `PdfDocumentEvent.END_PAGE` occurs. That is: every time iText has finished adding content to a page, either because a new page is created, or because the last page has been reached and completed.

Let's examine our `IEventHandler` implementation.

```
1  protected class MyEventHandler implements IEventHandler {
2      public void handleEvent(Event event) {
3          PdfDocumentEvent docEvent = (PdfDocumentEvent) event;
4          PdfDocument pdfDoc = docEvent.getDocument();
5          PdfPage page = docEvent.getPage();
6          int pageNumber = pdfDoc.getPageNumber(page);
7          Rectangle pageSize = page.getPageSize();
8          PdfCanvas pdfCanvas = new PdfCanvas(
9              page.newContentStreamBefore(), page.getResources(), pdfDoc);
10
```

```
11          //Set background
12          Color limeColor = new DeviceCmyk(0.208f, 0, 0.584f, 0);
13          Color blueColor = new DeviceCmyk(0.445f, 0.0546f, 0, 0.0667f);
14          pdfCanvas.saveState()
15                  .setFillColor(pageNumber % 2 == 1 ? limeColor : blueColor)
16                  .rectangle(pageSize.getLeft(), pageSize.getBottom(),
17                      pageSize.getWidth(), pageSize.getHeight())
18                  .fill().restoreState();
19          //Add header and footer
20          pdfCanvas.beginText()
21                  .setFontAndSize(helvetica, 9)
22                  .moveText(pageSize.getWidth() / 2 - 60, pageSize.getTop() - 20)
23                  .showText("THE TRUTH IS OUT THERE")
24                  .moveText(60, -pageSize.getTop() + 30)
25                  .showText(String.valueOf(pageNumber))
26                  .endText();
27          //Add watermark
28          Canvas canvas = new Canvas(pdfCanvas, pdfDoc, page.getPageSize());
29          canvas.setProperty(Property.FONT_COLOR, Color.WHITE);
30          canvas.setProperty(Property.FONT_SIZE, 60);
31          canvas.setProperty(Property.FONT, helveticaBold);
32          canvas.showTextAligned(new Paragraph("CONFIDENTIAL"),
33              298, 421, pdfDoc.getPageNumber(page),
34              TextAlignment.CENTER, VerticalAlignment.MIDDLE, 45);
35
36          pdfCanvas.release();
37      }
38  }
```

In our implementation of the `handleEvent()` method, we obtain the `PdfDocument` instance from the `PdfDocumentEvent` that is passed as a parameter (line 3-4). The `event` also gives us access to the `PdfPage` (line 5). We need those objects, to get the page number (line 6), the page size (line 7), and a `PdfCanvas` instance (line 8-9).

Different paths and shapes that are drawn on a page can overlap. As a rule, whatever comes first in the content stream is drawn first. Content that is drawn afterwards can cover content that already exists. We want to add a background each time the content of a page has been completely rendered. Each `PdfPage` object keeps track of an array of content streams. You can use the `getContentStream()` method with an index as parameter to get each separate content stream. You can use `getFirstContentStream()` and `getLastContentStream()` to get the first and the last content stream. You can also create a new content stream with the `newContentStreamBefore()` and `newContentStreamAfter()` method.

In our `handleEvent()` method, we'll work with a `PdfCanvas` constructor that was created with the following parameters:

- `page.newContentStreamBefore()`: if we would draw an opaque rectangle *after* the page was rendered, that rectangle would cover all the existing content. We need access to a content stream that will be added *before* the content of a page, so that our background and our watermark don't cover the content in our table.
- `page.getResources()`: each content stream refers to external resources such as fonts and images. As we are going to add new content to a page, it's important that iText has access to the resources dictionary of that page.
- `pdfDoc`: We need access to the `PdfDocument` so that it can produce the new PDF objects that represent the content we're adding.

What are we adding to the `canvas` object?

- line 11-18: we define two colors `limeColor` and `blueColor`. We save the current graphics state, and then change the fill color to either of these colors, depending on the page number. We construct a rectangle and fill it. This will paint the complete page either in lime or blue. We restore the graphics state to return to the original fill color, because don't want the other content to be affected by the color change.
- line 20-26: we begin a text object. We set a font and a font size. We move to a centered position close to the top of the page to write `"THE TRUTH IS OUT THERE"`. We then move the cursor to the bottom of the page where we write the page number. We end the text object. This adds a header and a footer to our page.
- line 28-31: we create a new `Canvas` object, named `canvas`. `Canvas` is the high-level equivalent of `PdfCanvas`, just like `Document` is the high-level equivalent for `PdfDocument`. Instead of having to use PDF syntax to change the font, font size, font color and other properties, we can use the `setProperty()` method. The `setProperty()` method can also be used on the `Document` object, for instance to change the default font of the document. It's available for objects such as `Paragraph`, `List`, `Table` for the same purpose.
- line 32-34: we use the `showTextAligned()` method to add a `Paragraph` that will be centered at the position `X = 298. Y = 421` with an angle of 45 degrees.

Once we've added a background, a header and a footer, and a watermark, we release the `PdfCanvas` object.

In this example, we used two different approaches to add text at an absolute position. We used some low-level methods we encountered when we discussed *text state* in the previous chapter for the header and the footer. We could have used similar methods to add the watermark. However: we want to rotate the text and center it in the middle of the page, and that requires quite some Math. To avoid having to calculate the transformation matrix that would put the text at the desired coordinates, we used a convenience method. With `showTextAligned()`, iText does all the heavy lifting in your place.

Summary

In this chapter, we've found out why it's important to have some understanding of the low-level functionality discussed in the previous chapter. We can use this functionality in combination with basic building blocks to create custom functionality. We've created custom borders to `Cell` objects. We've added background colors to pages, and we've introduced headers and footers. When we added a watermark, we discovered that we don't really need to know all the ins and outs of PDF syntax. We were able to use a convenience method that took care of defining the transformation matrix to rotate and center text.

In the next example, we'll learn about a different type of content. We'll take a look at annotations, and we'll zoom in on one particular type of annotation that will allow us to create interactive forms.

Chapter 4: Making a PDF interactive

In the previous chapters, we've created PDF documents by adding content to a page. It didn't matter if we were adding high-level objects (e.g. a `Paragraph`) or low-level instructions (e.g. `lineTo()`, `moveTo()`, `stroke()`), iText converted everything to PDF syntax that was written to one or more content streams. In this chapter, we'll add content of a different nature. We'll add interactive features, known as annotations. Annotations aren't part of the content stream. They are usually added on top of the existing content. There are many different types of annotations, many of which allow user interaction.

Adding annotations

We'll start with a series of simple examples. Figure 4.1 shows a PDF with a `Paragraph` of text. On top of the text, we've added a green text annotation.

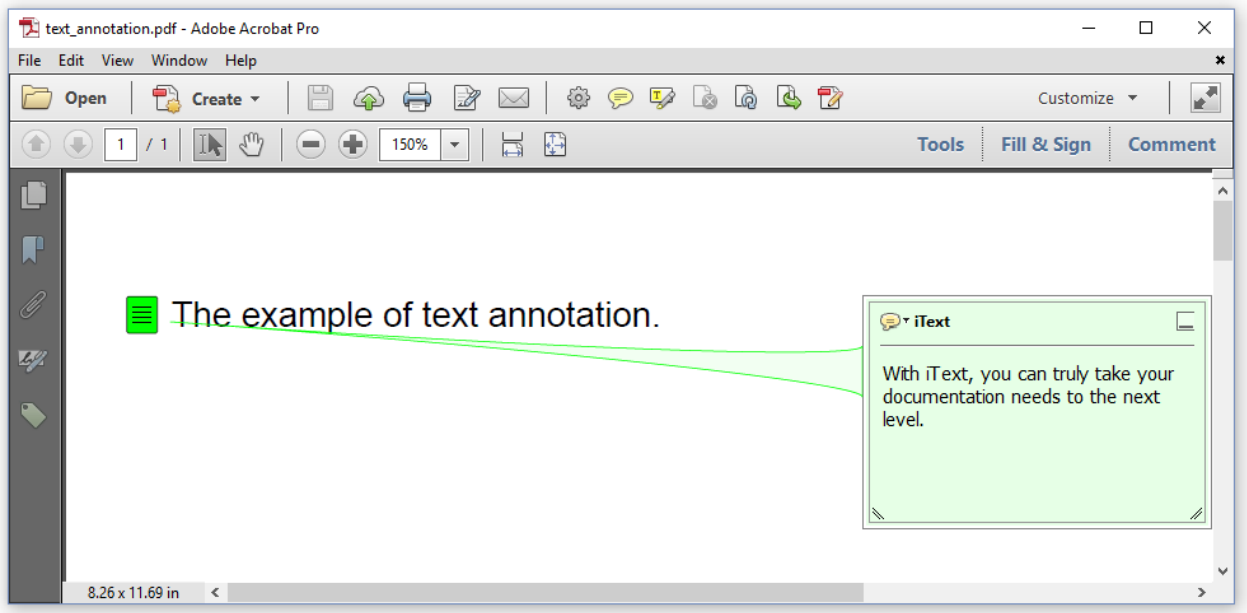


Figure 4.1: a text annotation

Most of the code of the TextAnnotation[22] example is identical to the Hello World example. The only difference is that we create and add an annotation:

[22] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1756-c04e01_01_textannotation.java

```
PdfAnnotation ann = new PdfTextAnnotation(new Rectangle(20, 800, 0, 0))
    .setColor(Color.GREEN)
    .setTitle(new PdfString("iText"))
    .setContents("With iText, "
        + "you can truly take your documentation needs to the next level.")
    .setOpen(true);
pdf.getFirstPage().addAnnotation(ann);
```

We define the location of the text annotation using a `Rectangle`. We set the color, title (a `PdfString`), contents (a `String`), and the open status of the annotation. We ask the `PdfDocument` for its first page and add the annotation.

In Figure 4.2, we created an annotation that is invisible, but that shows an URL if you hover over its location. You can open that URL by clicking the annotation. This is a link annotation.

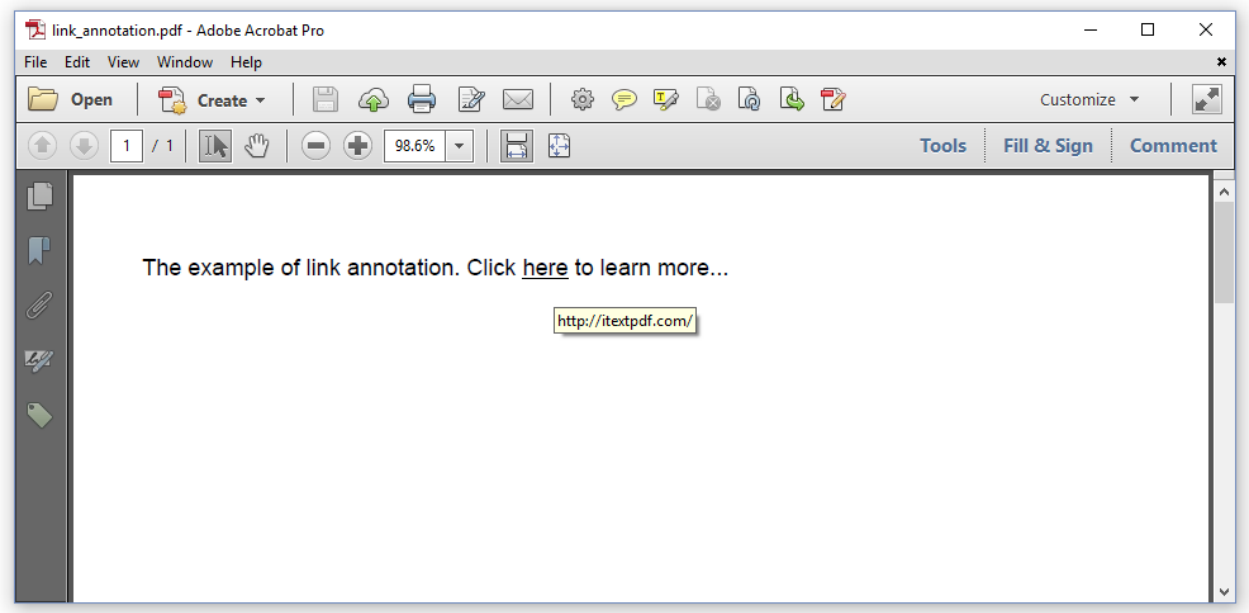


Figure 4.2: a link annotation

As the annotation is part of a sentence, it wouldn't be convenient if we had to calculate the position of the word "here". Fortunately, we can wrap the link annotation in a `Link` object and iText will calculate the `Rectangle` automatically. The LinkAnnotation[23] example shows how it's done.

[23] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1757-c04e01_02_linkannotation.java

```
1  PdfLinkAnnotation annotation = new PdfLinkAnnotation(new Rectangle(0, 0))
2          .setAction(PdfAction.createURI("http://itextpdf.com/"));
3  Link link = new Link("here", annotation);
4  Paragraph p = new Paragraph("The example of link annotation. Click ")
5          .add(link.setUnderline())
6          .add(" to learn more...");
7  document.add(p);
```

In line 2, we create a URI action that opens the iText web site. We use this action for the link annotation. We then create a `Link` object. This is a basic building block that accepts a link annotation as parameter. This link annotation won't be added to the content stream â€“because annotations aren't part of the content stream. Instead it will be added to the corresponding page at the corresponding coordinates. Making text clickable doesn't change the appearance of that text in the content stream. In our example, we underlined the word "here" so that we know where to click.

Every type of annotation requires its own type of parameters. Figure 4.3 shows a page with a line annotation.

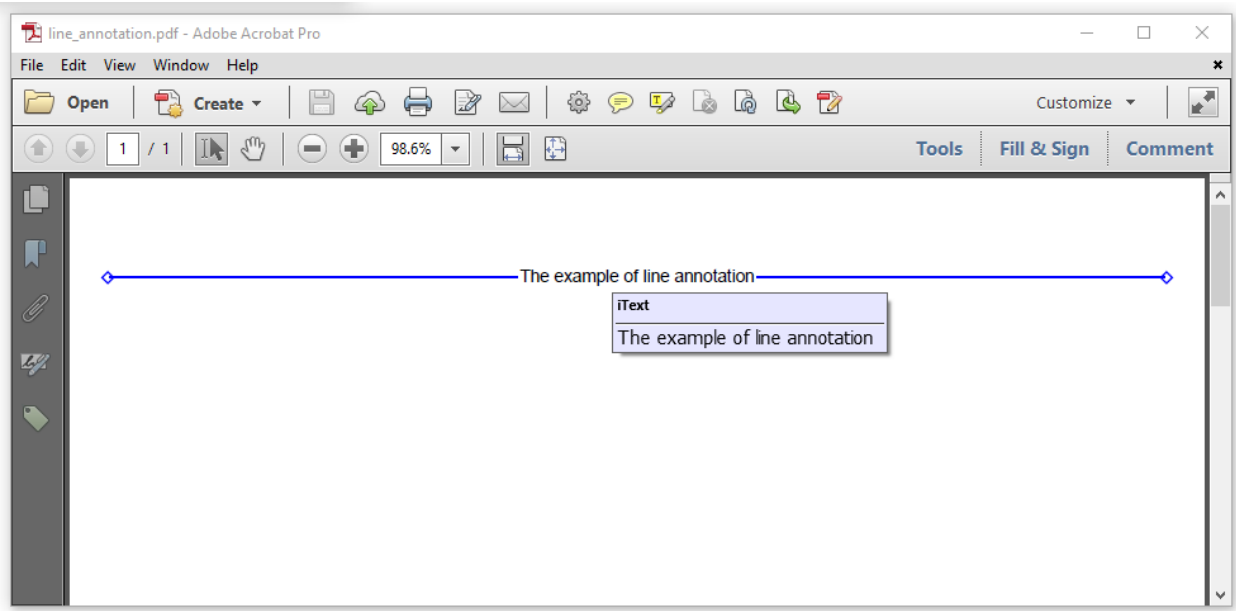


Figure 4.3: a line annotation

The LineAnnotation[24] shows what is needed to create this appearance.

[24] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1758-c04e01_03_lineannotation.java

```
OutputStream fos = new FileOutputStream(dest);
PdfWriter writer = new PdfWriter(fos);
PdfDocument pdf = new PdfDocument(writer);
PdfPage page = pdf.addNewPage();
PdfArray lineEndings = new PdfArray();
lineEndings.add(new PdfName("Diamond"));
lineEndings.add(new PdfName("Diamond"));
PdfAnnotation annotation = new PdfLineAnnotation(
    new Rectangle(0, 0),
    new float[]{20, 790, page.getPageSize().getWidth() - 20, 790})
        .setLineEndingStyles((lineEndings))
        .setContentsAsCaption(true)
        .setTitle(new PdfString("iText"))
        .setContents("The example of line annotation")
        .setColor(Color.BLUE);
page.addAnnotation(annotation);
pdf.close();
```

In this example, we add the annotation to a newly created page. There's no `Document` instance involved in this example.

ISO-32000-2 defines 28 different annotation types, two of which are deprecated in PDF 2.0. With iText, you can add all of these annotation types to a PDF document, but in the context of this tutorial, we'll only look at one more example before we move on to interactive forms. See figure 4.4.

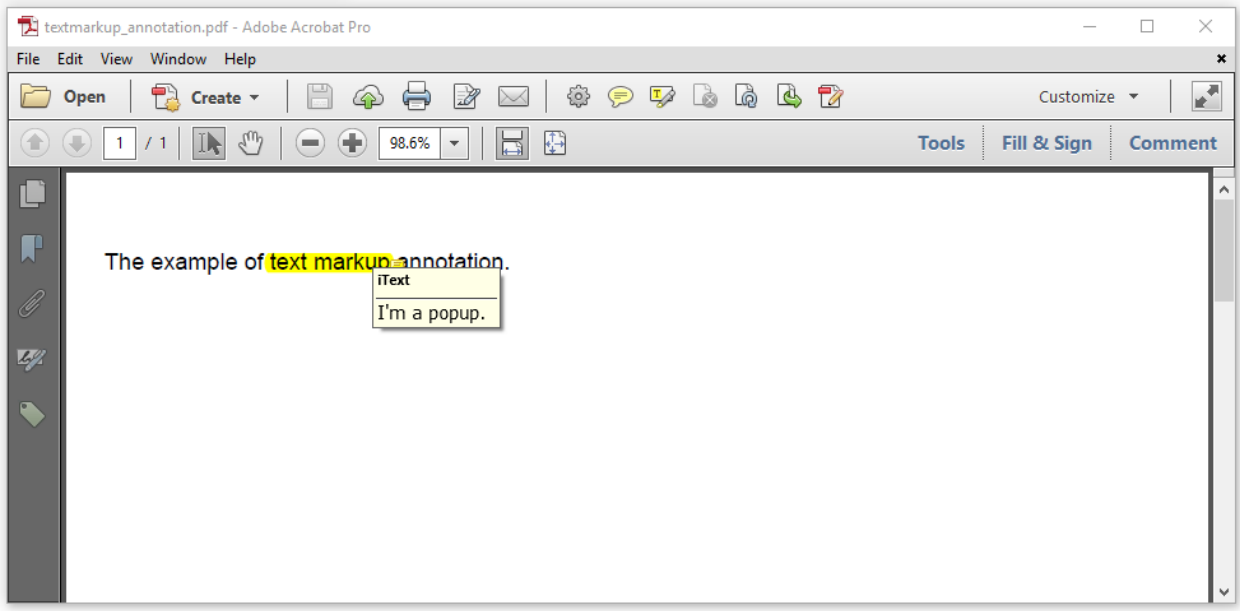


Figure 4.4: a markup annotation

Looking at the TextMarkupAnnotation[25] example, we see that we really need a separate tutorial to understand what all the nuts and bolts used in this code snippet are about.

```
PdfAnnotation ann = PdfTextMarkupAnnotation.createHighLight(
        new Rectangle(105, 790, 64, 10),
        new float[]{169, 790, 105, 790, 169, 800, 105, 800})
    .setColor(Color.YELLOW)
    .setTitle(new PdfString("Hello!"))
    .setContents(new PdfString("I'm a popup."))
    .setTitle(new PdfString("iText"))
    .setOpen(true)
    .setRectangle(new PdfArray(new float[]{100, 600, 200, 100}));
pdf.getFirstPage().addAnnotation(ann);
```

In the next section, we'll create an interactive form consisting of different form fields. Each form field in that form will correspond with a *widget annotation*, but those annotations will be created implicitly.

Creating an interactive form

In the next example, we're going to create an interactive form based on AcroForm technology. This technology was introduced in PDF 1.2 (1996) and allows you to populate a PDF document with form fields such as text fields, choices (combo box or list field), buttons (push buttons, check boxes and radio buttons), and signature fields.

It's tempting to compare a PDF form with a form in HTML, but that would be wrong. When text doesn't fit into the available text area of an HTML form, that field can be resized. The content of a list field can be updated on the fly based on a query to the server. In short, an HTML form can be very dynamic.

That isn't true for interactive forms based on AcroForm technology. Such a form can best be compared with a paper form where every field has its fixed place and its fixed size. The idea of using PDF forms for collecting user data in a web browser has been abandoned over the years. HTML forms are much more user friendly for online data collection.

That doesn't mean that AcroForm technology has become useless. Interactive PDF forms are very common in two specific use cases:

- *When the form is the equivalent of digital paper.* In some cases, there are strict formal requirements with respect to a form. It is important that the digital document is an exact

[25] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1759-c04e01_04_textmarkupannotation.java

replica of the corresponding form. Every form that is filled out needs to comply to the exact same formal requirements. If this is the case, then it's better to use PDF forms than HTML forms.

- *When the form isn't used for data collection, but as a template.* For example: you have a form that represents a voucher or an entry ticket for an event. On this form, you have different fields for the name of the person who bought the ticket, the date and the time of the event, the row and the seat number, and so on. When people buy a ticket, you don't need to regenerate the complete voucher, you can take the form and simply fill it out with the appropriate data.

In both use cases, the form will be created manually, for instance using Adobe software, LibreOffice, or any other tool with a graphical user interface.

You could also create such a form programmatically, but there are very few use cases that would justify using a software library to create a form or a template, instead of using a tool with a GUI. Nevertheless, we're going to give it a try.

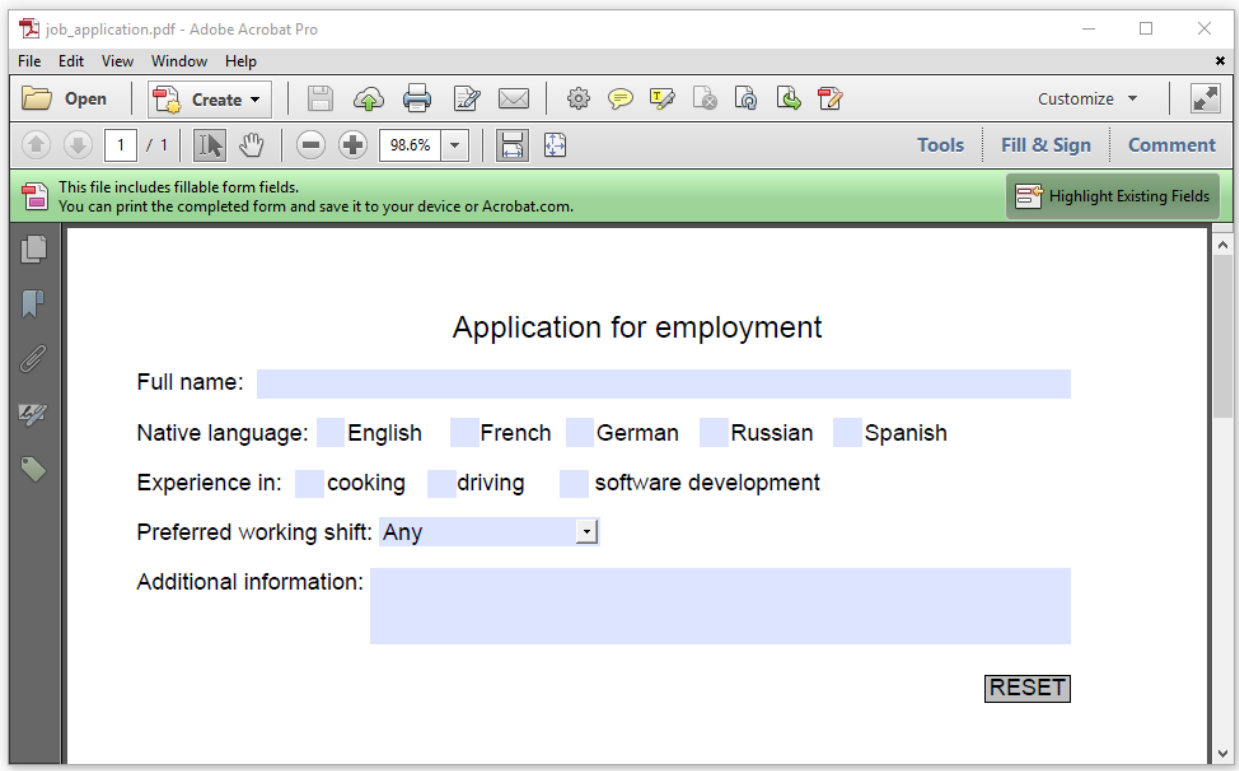


Figure 4.5: an interactive form

In Figure 4.5, we see text fields, radio buttons, check boxes, a combo box, a multi-line text field, and a push button. We see these fields because they are represented by a widget annotation. This widget annotation is created implicitly when we create a field. In the JobApplication[26] example, we create a

[26] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1760-c04e02_jobapplication.java

`PdfAcroForm` object, using the `PdfDocument` instance obtained from the `Document` object. The second parameter is a Boolean indicating if a new form needs to be created if there is no existing form. As we've just created the `Document`, there is no form present yet, so that parameter should be `true`:

```
PdfAcroForm form = PdfAcroForm.getAcroForm(doc.getPdfDocument(), true);
```

Now we can start adding fields. We'll use a `Rectangle` to define the dimension of each widget annotation and its position on the page.

Text field

We'll start with the text field that will be used for the full name.

```
1  PdfTextFormField nameField = PdfTextFormField.createText(
2      doc.getPdfDocument(), new Rectangle(99, 753, 425, 15), "name", "");
3  form.addField(nameField);
```

The `createText()` method needs a `PdfDocument` instance, a `Rectangle`, the name of the field, and a default value (in this case, the default value is an empty `String`). Note that the label of the field and the widget annotation are two different things. We've added "Full name:" using a `Paragraph`. That `Paragraph` is part of the content stream. The field itself doesn't belong in the content stream. It's represented using a widget annotation.

Radio buttons

We create a radio field for choosing a language. Note that there is one radio group named `language` with five unnamed button fields, one for each language that can be chosen:

```
1  PdfButtonFormField group = PdfFormField.createRadioGroup(
2      doc.getPdfDocument(), "language", "");
3  PdfFormField.createRadioButton(doc.getPdfDocument(),
4      new Rectangle(130, 728, 15, 15), group, "English");
5  PdfFormField.createRadioButton(doc.getPdfDocument(),
6      new Rectangle(200, 728, 15, 15), group, "French");
7  PdfFormField.createRadioButton(doc.getPdfDocument(),
8      new Rectangle(260, 728, 15, 15), group, "German");
9  PdfFormField.createRadioButton(doc.getPdfDocument(),
10     new Rectangle(330, 728, 15, 15), group, "Russian");
11 PdfFormField.createRadioButton(doc.getPdfDocument(),
12     new Rectangle(400, 728, 15, 15), group, "Spanish");
13 form.addField(group);
```

Only one language can be selected at a time. If multiple options could apply, we should have used check boxes.

Check boxes

In the next snippet, we'll introduce three check boxes, named `experience0`, `experience1`, `experience2`:

```
for (int i = 0; i < 3; i++) {
    PdfButtonFormField checkField = PdfFormField.createCheckBox(
        doc.getPdfDocument(), new Rectangle(119 + i * 69, 701, 15, 15),
        "experience".concat(String.valueOf(i+1)), "Off",
        PdfFormField.TYPE_CHECK);
    form.addField(checkField);
}
```

As you can see, we use the `createCheckBox()` method with the following parameters: the `PdfDocument` object, a `Rectangle`, the name of the field, the current value of the field, and the appearance of the check mark.

A check box has two possible values: the value of the *off* state must be `"Off"`; the value of the *on* state is usually `"Yes"` (it's the value iText uses by default), but some freedom is allowed here.

It's also possible to have people select one or more option from a list or a combo box. In PDF terminology, we call this a choice field.

Choice field

Choice fields can be configured in a way that people can select only one of the options, or several options. In our example, we create a combo box.

```
String[] options = {"Any", "6.30 am - 2.30 pm", "1.30 pm - 9.30 pm"};
PdfChoiceFormField choiceField = PdfFormField.createComboBox(
    doc.getPdfDocument(), new Rectangle(163, 676, 115, 15),
    "shift", "Any", options);
form.addField(choiceField);
```

Our choice field is named `"shift"` and it offers three `options` of which `"Any"` is selected by default.

Multi-line field

We also see a multi-line field in the form. As opposed to the regular text field, where you can only add text in a single line, text in this field will be wrapped if it doesn't fit on a single line.

```
PdfTextFormField infoField = PdfTextFormField.createMultilineText(
    doc.getPdfDocument(), new Rectangle(158, 625, 366, 40), "info", "");
form.addField(infoField);
```

We'll conclude our form with a push button.

Push button

In a real-world example we'd use a submit button that allows people to submit the data they've entered in the form to a server. Such PDF forms have become rare since HTML evolved to HTML 5 and related technologies, introducing much more user-friendly functionality to fill out form. We conclude the example by adding a reset button that will reset a selection of fields to their initial value when the button is clicked.

```
PdfButtonFormField button = PdfFormField.createPushButton(doc.getPdfDocument(),
        new Rectangle(479, 594, 45, 15), "reset", "RESET");
button.setAction(PdfAction.createResetForm(
    new String[] {"name", "language", "experience1", "experience2",
        "experience3", "shift", "info"}, 0));
form.addField(button);
```

If you want to create a PDF form using iText, you now have a fair idea of how it's done. In many cases, it's a much better idea to create a form manually, using a tool with a graphical user interface. You are then going to use iText to fill out this form automatically, for instance using data from a database.

Filling out a form

When we created our form, we could have defined default values, so that the form was filled out as shown in Figure 4.6.

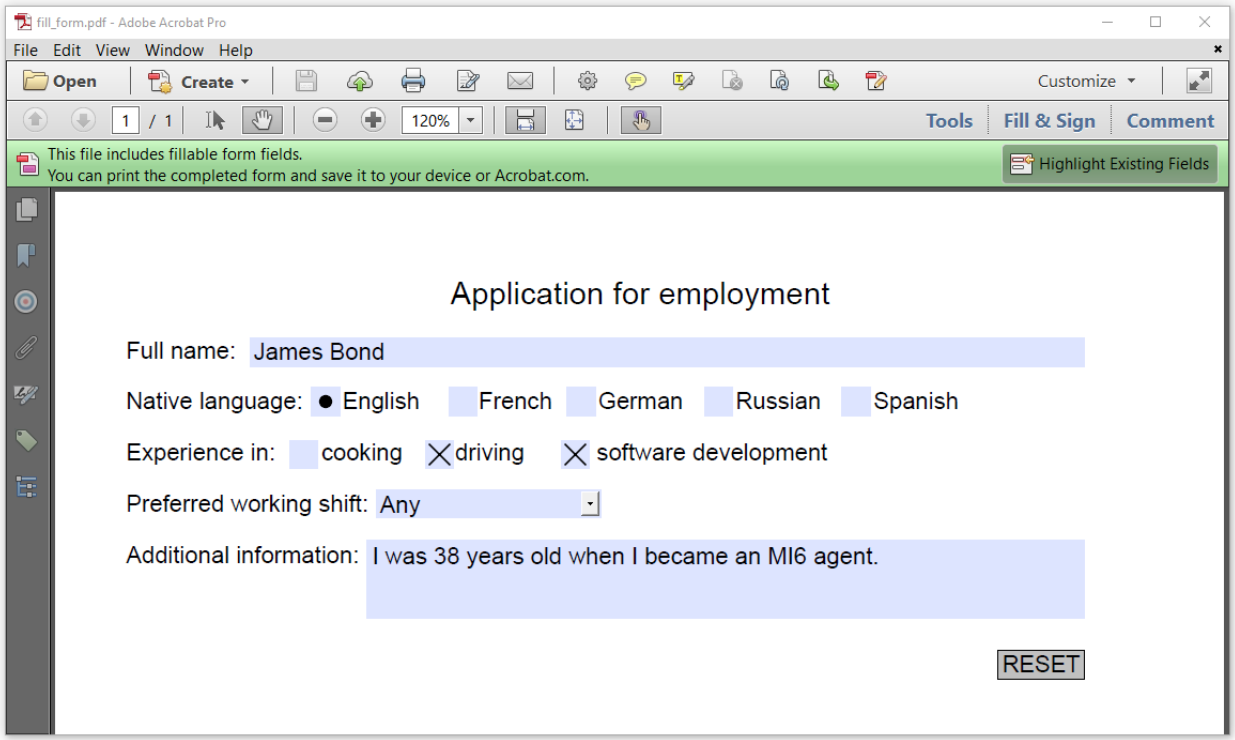


Figure 4.6: a filled-out interactive form

We can still add these values *after* we've created the form. The CreateAndFill[27] example shows us how.

```
Map<String, PdfFormField> fields = form.getFormFields();
fields.get("name").setValue("James Bond");
fields.get("language").setValue("English");
fields.get("experience1").setValue("Off");
fields.get("experience2").setValue("Yes");
fields.get("experience3").setValue("Yes");
fields.get("shift").setValue("Any");
fields.get("info").setValue("I was 38 years old when I became an MI6 agent.");
```

We asked the `PdfAcroForm` to which we've added all the form field for its fields, and we get a `Map` consisting of key-value pairs with the names and `PdfFormField` objects of each field. We can get the `PdfFormField` instances one by one, and set their value. Granted, this doesn't make much sense. It would probably have been smarter to set the correct value right away the moment you create each field. A more common use case is to pre-fill an existing form.

[27] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1761-c04e03_createandfill.java

Pre-filling an existing form

In the next example, we'll take an existing form, job_application.pdf[28], get a `PdfAcroForm` object from that form, and use the very same code to fill out that existing document. See the FillForm[29] example.

```
1  PdfReader reader = new PdfReader(src);
2  PdfWriter writer = new PdfWriter(dest);
3  PdfDocument pdf = new PdfDocument(reader, writer);
4  PdfAcroForm form = PdfAcroForm.getAcroForm(pdf, true);
5  Map<String, PdfFormField> fields = form.getFormFields();
6  fields.get("name").setValue("James Bond");
7  fields.get("language").setValue("English");
8  fields.get("experience1").setValue("Off");
9  fields.get("experience2").setValue("Yes");
10 fields.get("experience3").setValue("Yes");
11 fields.get("shift").setValue("Any");
12 fields.get("info").setValue("I was 38 years old when I became an MI6 agent.");
13 pdf.close();
```

We introduce a new object in line 1. `PdfReader` is a class that allows iText to access a PDF file and read the different PDF objects stored in a PDF file. In this case, `src` holds the path to an existing form.

I/O is handled by two classes in iText.

- `PdfReader` is the *input* class;
- `PdfWriter` is the *output* class.

In line 2, we create a `PdfWriter` that will write a new version of the source file. Line 3 is different from what we did before. We now create a `PdfDocument` object using the `reader` and the `writer` object as parameters. We obtain a `PdfAcroForm` instance using the same `getAcroForm()` method as before. Lines 5 to 12 are identical to the lines we used to fill out the values of the fields we created from scratch. When we close the `PdfDocument` (line 13), we have a PDF that is identical to the one shown in Figure 5.6.

[28] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/cmpfiles/chapter04/cmp_job_application.pdf
[29] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1762-c04e04_fillform.java

The form is still interactive: people can still change values if they want to. iText has been used in many applications to pre-fill forms. For instance: when people log in into an online service, a lot of information (e.g. name, address, phone number) is already known about them on the server side. When they need to fill out a form online, it doesn't make much sense to present them a blank file where they have to fill out their name, address and phone number all over again. Plenty of time can be saved if these values are already present in the form. This can be achieved by pre-filling the form with iText. People can check if the information is correct and if it isn't (for instance because their phone number changed), they can still change the content of the field.

Sometimes you don't want an end user to change information on a PDF. For instance: if the form is a voucher with a specific date and time, you don't want the end user to change that date and time. In that case, you'll flatten the form.

Flattening a form

When we add a single line to the previous code snippet, we get a PDF that is no longer interactive. The bar with the message "This file includes fillable form fields" has disappeared in Figure 4.7. When you click the name "James Bond", you can no longer manually change it.

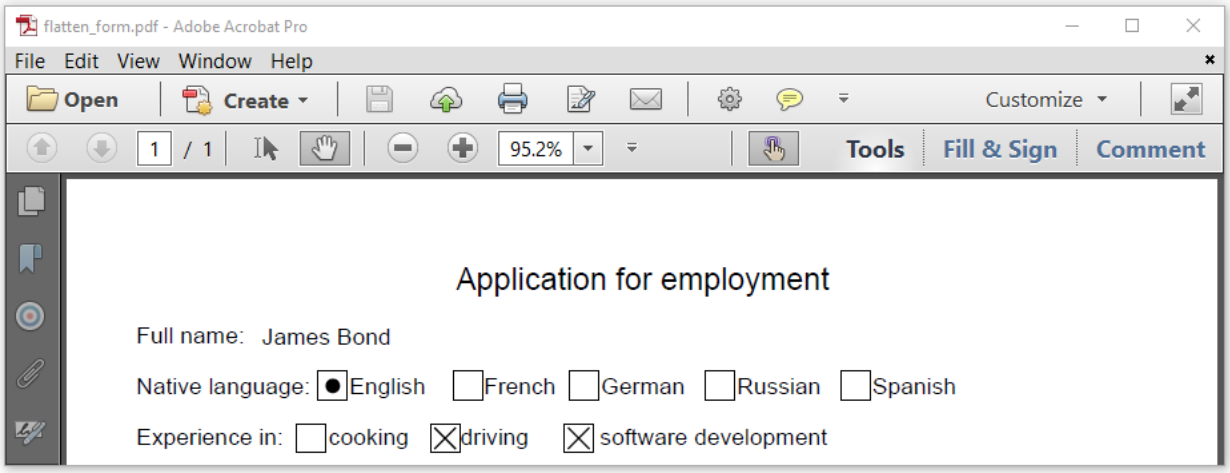


Figure 4.7: a flattened form

This extra line was added in the FlattenForm[30] example.

[30] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1763-c04e05_flattenform.java

```
1  PdfReader reader = new PdfReader(src);
2  PdfWriter writer = new PdfWriter(dest);
3  PdfDocument pdf = new PdfDocument(reader, writer);
4  PdfAcroForm form = PdfAcroForm.getAcroForm(pdf, true);
5  Map<String, PdfFormField> fields = form.getFormFields();
6  fields.get("name").setValue("James Bond");
7  fields.get("language").setValue("English");
8  fields.get("experience1").setValue("Off");
9  fields.get("experience2").setValue("Yes");
10 fields.get("experience3").setValue("Yes");
11 fields.get("shift").setValue("Any");
12 fields.get("info").setValue("I was 38 years old when I became an MI6 agent.");
13 form.flattenFields();
14 pdf.close();
```

After we've set all the values of the form fields, we add line 13: `form.flattenFields()` and all the fields will be removed; the corresponding widget annotations will be replaced by their content.

Summary

We started this chapter by looking as a handful of annotation types:

- a text annotation,
- a link annotation,
- a line annotation, and
- a text markup annotation.

We also mentioned widget annotations. This led us to the subject of interactive forms. We learned how to create a form, but more importantly how to fill out and flatten a form.

In the fill and flatten examples, we encountered a new class, `PdfReader`. In the next chapter, we'll take a look at some more examples that use this class.

Chapter 5: Manipulating an existing PDF document

In the examples for chapter 1 to 3, we've always created a new PDF document from scratch with iText. In the last couple of examples of chapter 4, we worked with an existing PDF document. We took an existing interactive PDF form and filled it out, either resulting in a pre-filled form, or resulting in a flattened document that was no longer interactive. In this example, we'll continue working with existing PDFs. We'll load an existing file using `PdfReader` and we'll use the `reader` object to create a new `PdfDocument`.

Adding annotations and content

In the previous chapter, we took an existing PDF form, job_application.pdf[31], and we filled out the fields. In this chapter, we'll take it a step further. We'll start by adding a text annotation, some text, and a new check box. This is shown in Figure 5.1.

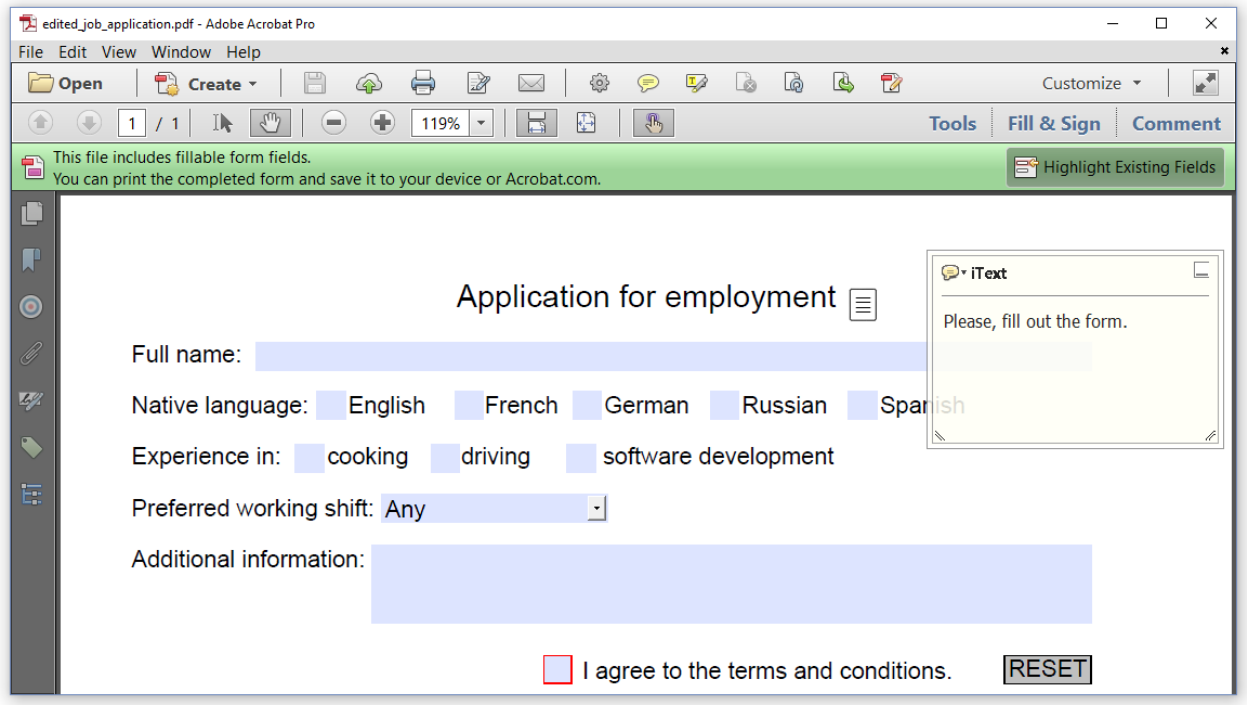


Figure 5.1: an updated form

[31] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/job_application.pdf

We'll repeat the code we've seen in the previous chapter in the AddAnnotationsAndContent[32] example.

```
PdfReader reader = new PdfReader(src);
PdfWriter writer = new PdfWriter(dest);
PdfDocument pdfDoc = new PdfDocument(reader, writer);
// add content
pdfDoc.close();
```

Where it says `// add content`, we'll add the annotation, the extra text, and the extra check box.

Just like in chapter 4, we add the annotation to a page obtained from the `PdfDocument` instance:

```
PdfTextAnnotation ann = new PdfTextAnnotation(new Rectangle(400, 795, 0, 0))
        .setTitle(new PdfString("iText"))
        .setContents("Please, fill out the form.")
        .setOpen(true);
pdfDoc.getFirstPage().addAnnotation(ann);
```

If we want to add content to a content stream, we need to create a `PdfCanvas` object. We can do this using a `PdfPage` object as a parameter for the `PdfCanvas` constructor:

```
PdfCanvas canvas = new PdfCanvas(pdfDoc.getFirstPage());
canvas.beginText().setFontAndSize(
        PdfFontFactory.createFont(FontConstants.HELVETICA), 12)
        .moveText(265, 597)
        .showText("I agree to the terms and conditions.")
        .endText();
```

The code to add the text is similar to what we did in chapter 2. Whether you're creating a document from scratch, or adding content to an existing document, has no impact on the instructions we use. The same goes for adding fields to a `PdfAcroForm` instance:

```
PdfAcroForm form = PdfAcroForm.getAcroForm(pdfDoc, true);
PdfButtonFormField checkField = PdfFormField.createCheckBox(
        pdfDoc, new Rectangle(245, 594, 15, 15),
        "agreement", "Off", PdfFormField.TYPE_CHECK);
checkField.setRequired(true);
form.addField(checkField);
```

Now that we've added an extra field, we might want to change the reset action:

[32] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-5#1773-c05e01_addannotationsandcontent.java

```
form.getField("reset").setAction(PdfAction.createResetForm(
    new String[]{"name", "language", "experience1", "experience2",
    "experience3", "shift", "info", "agreement"}, 0));
```

Let's see if we can also change some of the visual aspects of the form fields.

Changing the properties of form fields

In the FillAndModifyForm[33] example, we return to the FillForm[34] example from chapter 4, but instead of merely filling out the form, we also change the properties of the fields:

```
PdfAcroForm form = PdfAcroForm.getAcroForm(pdfDoc, true);
Map<String, PdfFormField> fields = form.getFormFields();
fields.get("name").setValue("James Bond").setBackgroundColor(Color.ORANGE);
fields.get("language").setValue("English");
fields.get("experience1").setValue("Yes");
fields.get("experience2").setValue("Yes");
fields.get("experience3").setValue("Yes");
List<PdfString> options = new ArrayList<PdfString>();
options.add(new PdfString("Any"));
options.add(new PdfString("8.30 am - 12.30 pm"));
options.add(new PdfString("12.30 pm - 4.30 pm"));
options.add(new PdfString("4.30 pm - 8.30 pm"));
options.add(new PdfString("8.30 pm - 12.30 am"));
options.add(new PdfString("12.30 am - 4.30 am"));
options.add(new PdfString("4.30 am - 8.30 am"));
PdfArray arr = new PdfArray(options);
fields.get("shift").setOptions(arr);
fields.get("shift").setValue("Any");
PdfFont courier = PdfFontFactory.createFont(FontConstants.COURIER);
fields.get("info")
    .setValue("I was 38 years old when I became a 007 agent.", courier, 7);
```

Please take a closer look at the following lines:

- line 3: we set the value of the `"name"` field to `"James Bond"`, but we also change the background color to `Color.ORANGE`.

[33] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-5#1774-c05e02_fillandmodifyform.java
[34] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-4#1762-c04e04_fillform.java

- line 8-17: we create a Java `List` containing more options than the form originally contained (line 8-15). We convert this `List` to a `PdfArray` (line 16) and we use this array to update the options of the `"shift"` field (line 17).
- line 19-21: we create a new `PdfFont` and we use this font and a new font size as extra parameters when we set the value of the `"info"` field.

Let's take a look at Figure 5.2 to see if our changes were applied.

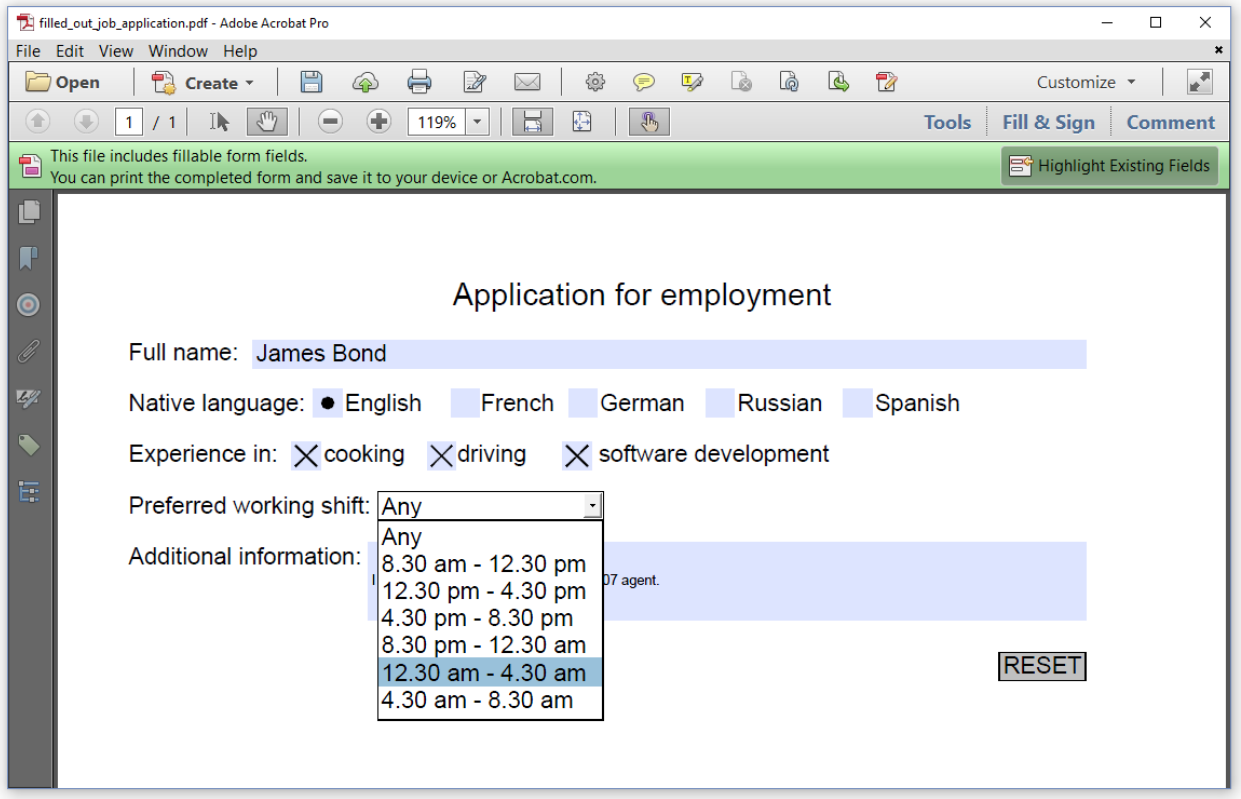


Figure 5.2: updated form with highlighted fields

We see that the `"shift"` field now has more options, but we don't see the background color of the `"name"` field. It's also not clear if the font of the `"info"` field has changed. What's wrong? Nothing is wrong, the fields are currently highlighted and the blue highlighting covers the background color. Let's click "Highlight Existing Fields" and see what happens.

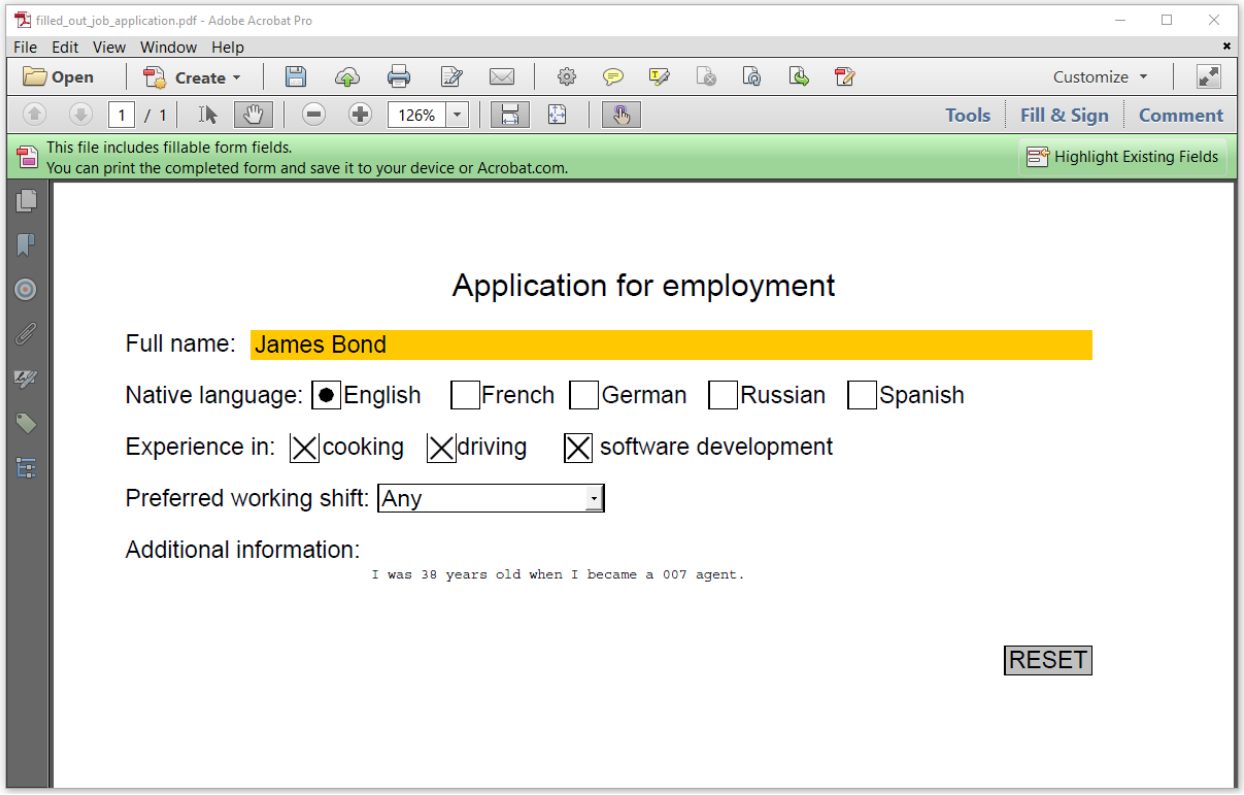


Figure 5.3: updated form, no highlighting

Now Figure 5.3 looks exactly the way we expected. We wouldn't have had this problem if we had added `form.flattenFields();` right before closing the `PdfDocument`, but in that case, we would no longer have a form either. We'll make some more forms examples in the next chapter, but for now, let's see what we can do with existing documents that don't contain a form.

Adding a header, footer, and watermark

Do you remember the report of the UFO sightings in the 20th century we created in chapter 3? We'll use a similar report for the next couple of examples: ufo.pdf[35], see Figure 5.4.

[35] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/ufo.pdf

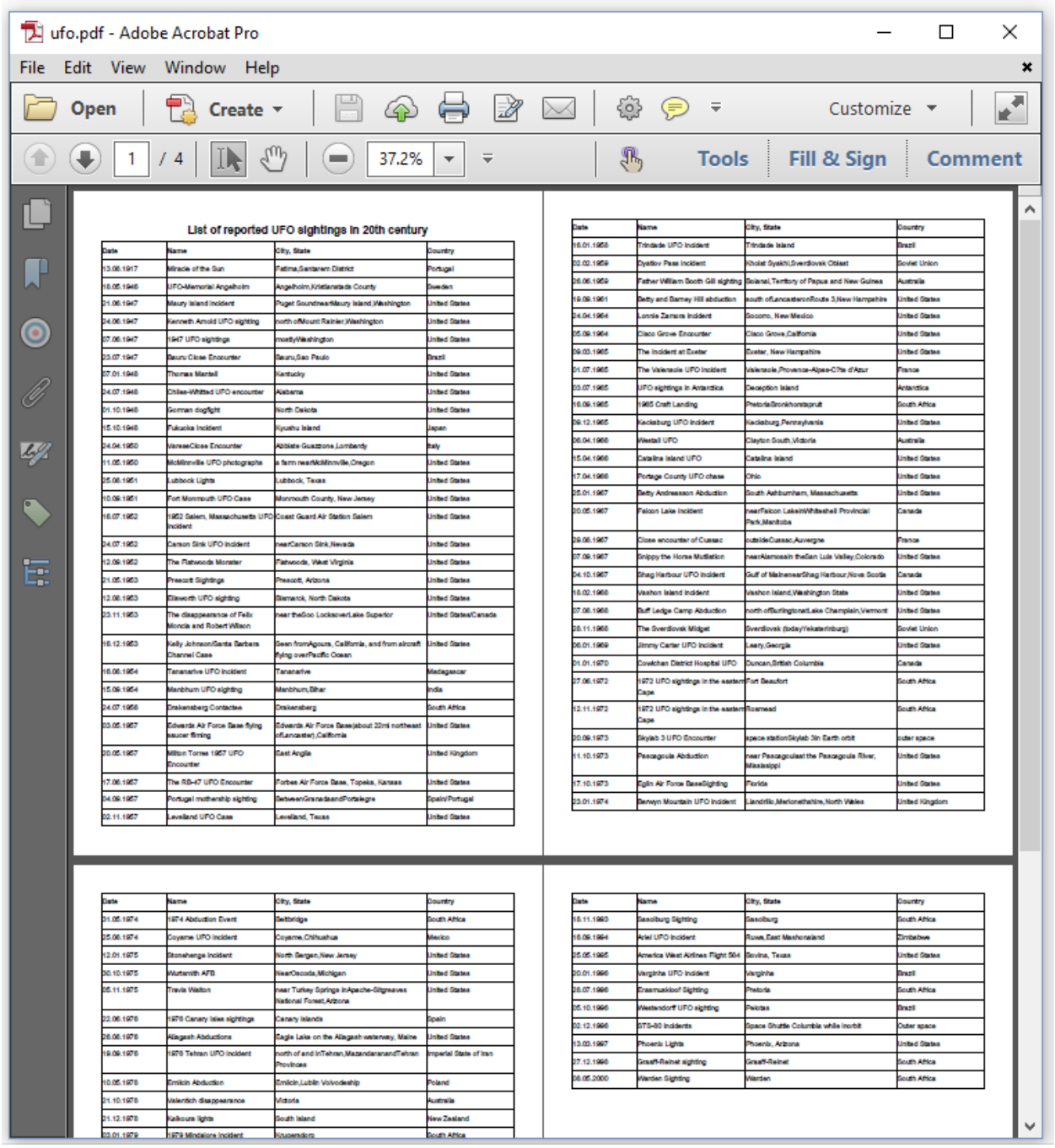


Figure 5.4: UFO sightings report

As you can see, it's not so fancy as the report we made in chapter 3. What if we'd like to add a header, a watermark and a footer saying "page X of Y" to this existing report? Figure 5.5 shows what such a report would look like.

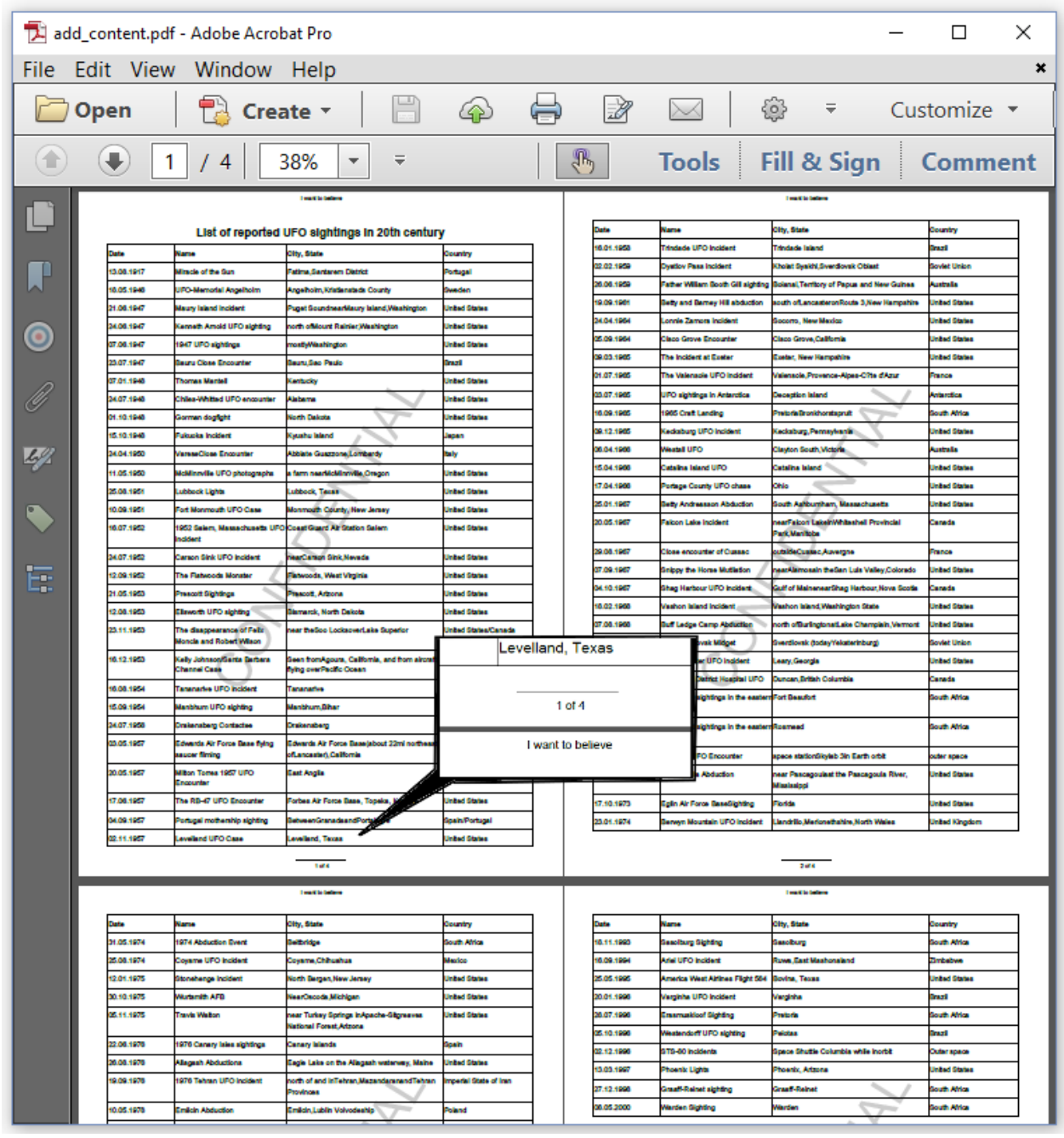


Figure 5.5: UFO sightings report with header, footer, and watermark

In Figure 5.5, we zoom in on an advantage that we didn't have when we added the page numbers in chapter 3. In chapter 3, we didn't know the total number of pages at the moment we were adding the footer, hence we only added the current page number. Now that we have an existing document, we can add "1 of 4", "2 of 4", and so on.

When creating a document from scratch, it's possible to create a placeholder for the total number of pages. Once all the pages are created, we can then add the total number of pages to that placeholder, but that's outside the scope of this introductory tutorial.

The AddContent[36] example shows how we can add content to every page in an existing document.

```
PdfReader reader = new PdfReader(src);
PdfWriter writer = new PdfWriter(dest);
PdfDocument pdfDoc = new PdfDocument(reader, writer);
Document document = new Document(pdfDoc);
Rectangle pageSize;
PdfCanvas canvas;
int n = pdfDoc.getNumberOfPages();
for (int i = 1; i <= n; i++) {
    PdfPage page = pdfDoc.getPage(i);
    pageSize = page.getPageSize();
    canvas = new PdfCanvas(page);
    // add new content
}
pdfDoc.close();
```

We use the `pdfDoc` object to create a `Document` instance. We'll use that `document` object to add some content. We also use the `pdfDoc` object to find the number of pages in the original PDF. We loop over all the pages, and we get the `PdfPage` object of each page. Let's take a look at the `// add new content` part we omitted.

```
//Draw header text
canvas.beginText().setFontAndSize(
        PdfFontFactory.createFont(FontConstants.HELVETICA), 7)
        .moveText(pageSize.getWidth() / 2 - 24, pageSize.getHeight() - 10)
        .showText("I want to believe")
        .endText();
//Draw footer line
canvas.setStrokeColor(Color.BLACK)
        .setLineWidth(.2f)
        .moveTo(pageSize.getWidth() / 2 - 30, 20)
        .lineTo(pageSize.getWidth() / 2 + 30, 20).stroke();
//Draw page number
canvas.beginText().setFontAndSize(
        PdfFontFactory.createFont(FontConstants.HELVETICA), 7)
```

[36] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-5#1775-c05e03_addcontent.java

```
15          .moveText(pageSize.getWidth() / 2 - 7, 10)
16          .showText(String.valueOf(i))
17          .showText(" of ")
18          .showText(String.valueOf(n))
19          .endText();
20  //Draw watermark
21  Paragraph p = new Paragraph("CONFIDENTIAL").setFontSize(60);
22  canvas.saveState();
23  PdfExtGState gs1 = new PdfExtGState().setFillOpacity(0.2f);
24  canvas.setExtGState(gs1);
25  document.showTextAligned(p,
26          pageSize.getWidth() / 2, pageSize.getHeight() / 2,
27          pdfDoc.getPageNumber(page),
28          Property.TextAlignment.CENTER, Property.VerticalAlignment.MIDDLE, 45);
29  canvas.restoreState();
```

We are adding four parts of content:

1. A header (line 2-6): we use low-level text functionality to add `"I want to believe"` at the top of the page.
2. A footer line (line 8-11): we use low-level graphics functionality to draw a line at the bottom of the page.
3. A footer with the page number (13-19): we use low-level text functionality to add the page number, followed by `" of "`, followed by the total number of pages at the bottom of the page.
4. A watermark (lin 21-28): we create a `Paragraph` with the text we want to add as a watermark. Then we change the opacity of the canvas. Finally we add the `Paragraph` to the document, centered in the middle of the page and with an angle of 45 degrees, using the `showTextAligned()` method.

We're doing something special when we add the watermark. We're changing the graphics state of the `canvas` object obtained from the page. Then we add text to the corresponding page in the `document`. Internally, iText will detect that we're already using the `PdfCanvas` instance of that page and the `showTextAligned()` method will write to that same `canvas`. This way, we can use a mix of low-level and convenience methods.

In the final example of this chapter, we'll change the page size and orientation of the pages of our UFO sightings report.

Changing the page size and orientation

If we take a look at Figure 5.6, we see our original report from Figure 5.4, but the pages are bigger and the second page has been turned up-side down.

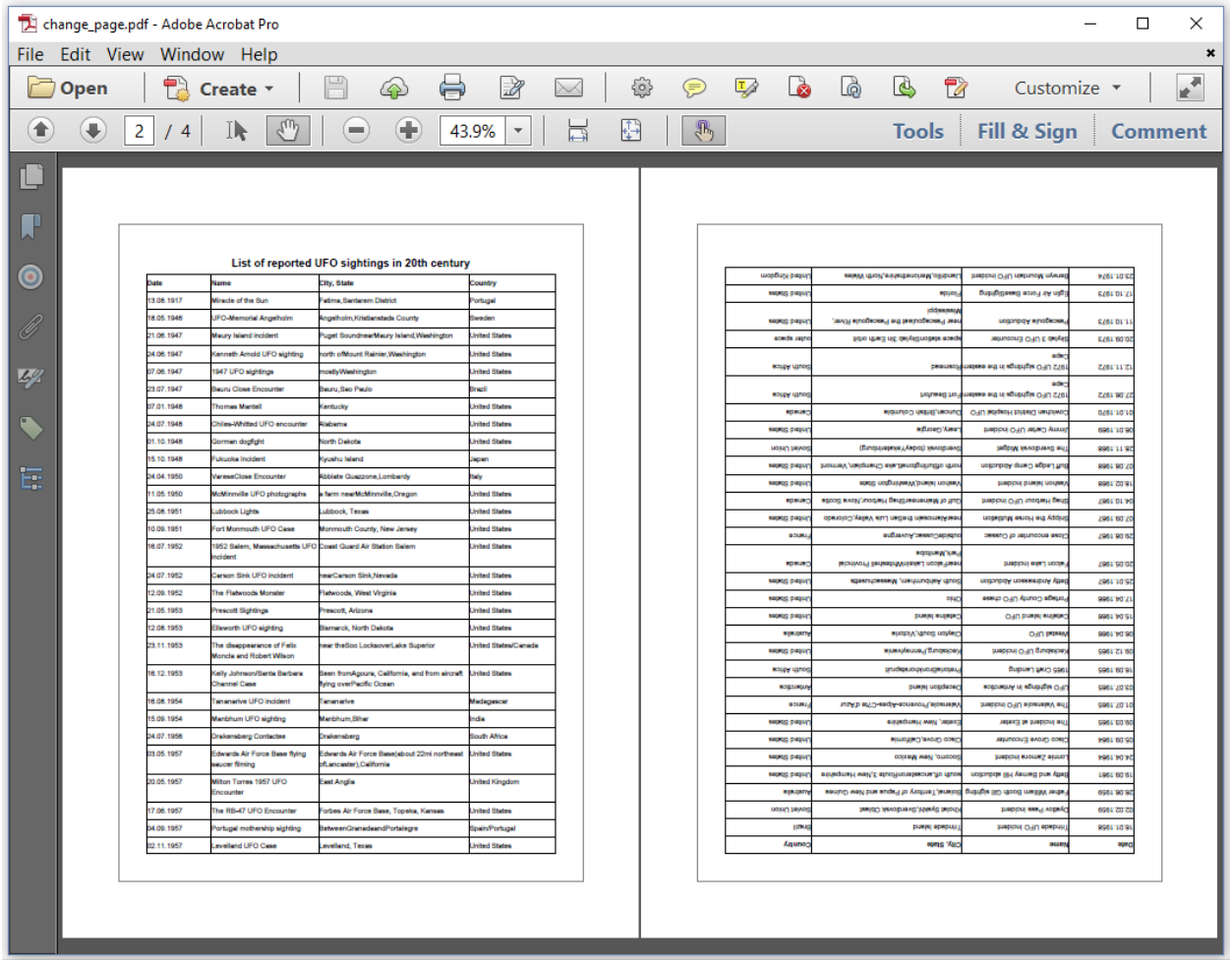


Figure 5.6: changed page size and orientation

The ChangePage[37] example shows how this was done.

```
PdfReader reader = new PdfReader(src);
PdfWriter writer = new PdfWriter(dest);
PdfDocument pdfDoc = new PdfDocument(reader, writer);
float margin = 72;
for (int i = 1; i <= pdfDoc.getNumberOfPages(); i++) {
    PdfPage page = pdfDoc.getPage(i);
    // change page size
    Rectangle mediaBox = page.getMediaBox();
    Rectangle newMediaBox = new Rectangle(
            mediaBox.getLeft() - margin, mediaBox.getBottom() - margin,
            mediaBox.getWidth() + margin * 2, mediaBox.getHeight() + margin * 2);
    page.setMediaBox(newMediaBox);
```

[37] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-5#1776-c05e04_changepage.java

```
13      // add border
14      PdfCanvas over = new PdfCanvas(page);
15      over.setStrokeColor(Color.GRAY);
16      over.rectangle(mediaBox.getLeft(), mediaBox.getBottom(),
17              mediaBox.getWidth(), mediaBox.getHeight());
18      over.stroke();
19      // change rotation of the even pages
20      if (i % 2 == 0) {
21          page.setRotation(180);
22      }
23  }
24  pdfDoc.close();
```

No need for a `Document` instance here, we work with the `PdfDocument` instance only. We loop over all the pages (line 5) and get the `PdfPage` instance of each page (line 6).

- A page can have different page boundaries, one of which isn't optional: the `/MediaBox`. We get the value of this page boundary as a `Rectangle` (line 8) and we create a new `Rectangle` that is an inch larger on each side (line 9-11). We use the `setMediaBox()` method to change the page size.
- We create a `PdfCanvas` object for the page (line 14), and we stroke a gray line using the dimensions of the original `mediaBox` (line 15-18).
- For every even page (line 20), we set the page rotation to 180 degrees.

Manipulating an existing PDF document requires some knowledge about PDF. For instance: you need to know the concept of the `/MediaBox`. We have tried to keep the examples simple, but that also means that we've cut some corners. For instance: in our last example, we didn't bother to check if a `/CropBox` was defined. If the original PDF had a `/CropBox`, enlarging the `/MediaBox` wouldn't have had any visual effect. We'll need a more in-depth tutorial to cover topics like these.

Summary

In the previous chapter, we learned about interactive PDF forms. In this chapter, we continued working with these forms. We added an annotation, some text, and an extra field to an existing form. We also changed some properties while filling out a form.

We then moved on to PDFs without any interactivity. First, we added a header, a footer, and a watermark. Then, we played with the size and the orientation of the pages of an existing document.

In the next chapter, we'll scale and tile existing documents, and we'll discover how to assemble multiple documents into a single PDF.

Chapter 6: Reusing existing PDF documents

In this chapter, we'll do some more document manipulation, but there will be a subtle difference in approach. In the examples of the previous chapter, we created *one* `PdfDocument` instance that linked a `PdfReader` to a `PdfWriter`. We manipulated *a single document.*

In this chapter, we'll always create at least two `PdfDocument` instances: one or more for the source document(s), and one for the destination document.

Scaling, tiling, and N-upping

Let's start with some examples that scale and tile a document.

Scaling PDF pages

Suppose that we have a PDF file with a single page, measuring 16.54 by 11.69 in. See Figure 6.1.

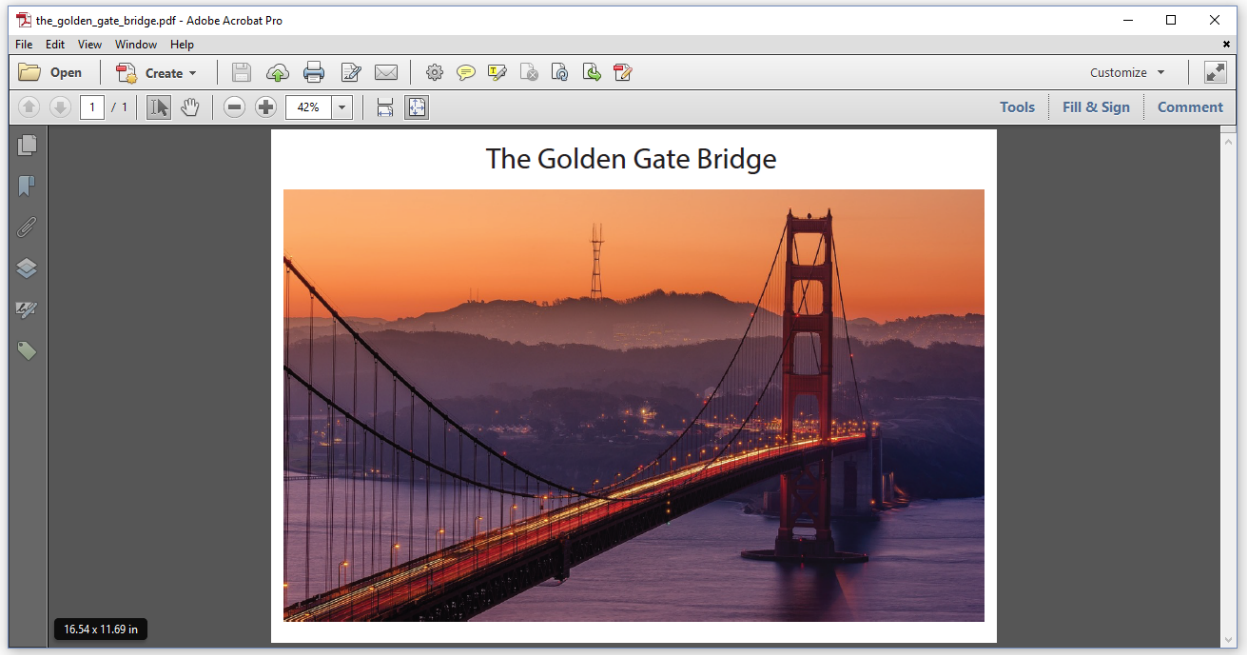


Figure 6.1: Golden Gate Bridge, original size 16.54 x 11.69 in

Now we want to create a PDF file with three pages. In page one, the original page is scaled down to 11.69 x 8.26 in as shown in Figure 6.2. On page 2, the original page size is preserved. On page 3, the original page is scaled up to 23.39 x 16.53 in as shown in Figure 6.3.

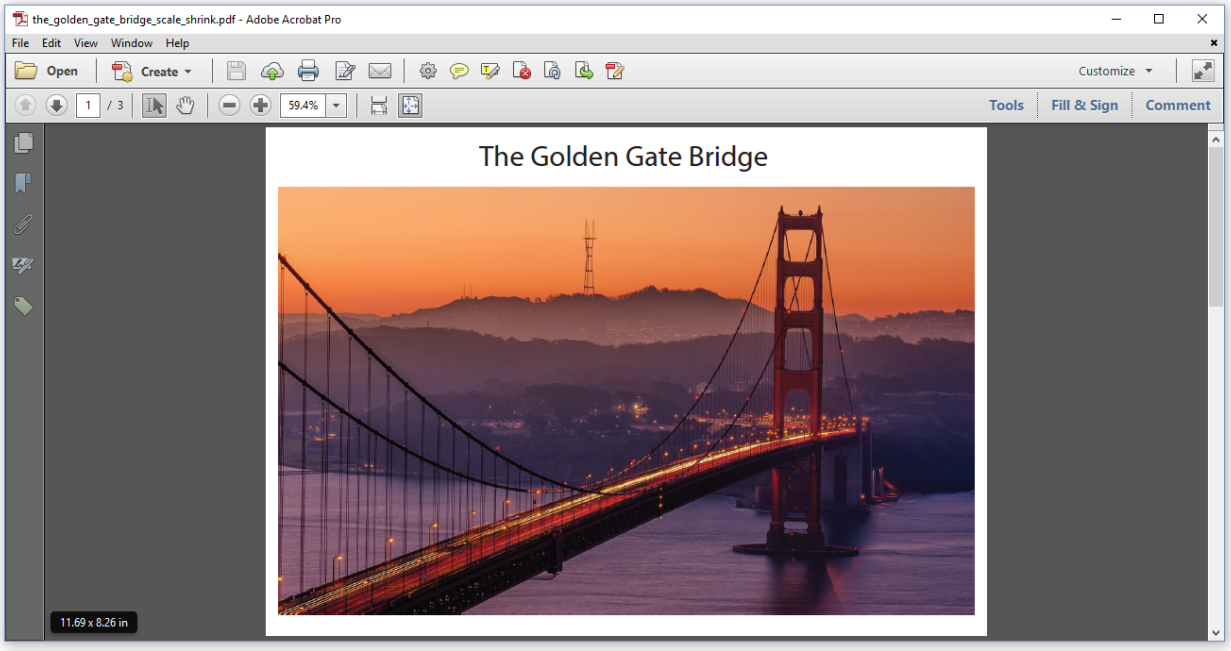


Figure 6.2: Golden Gate Bridge, scaled down to 11.69 x 8.26 in

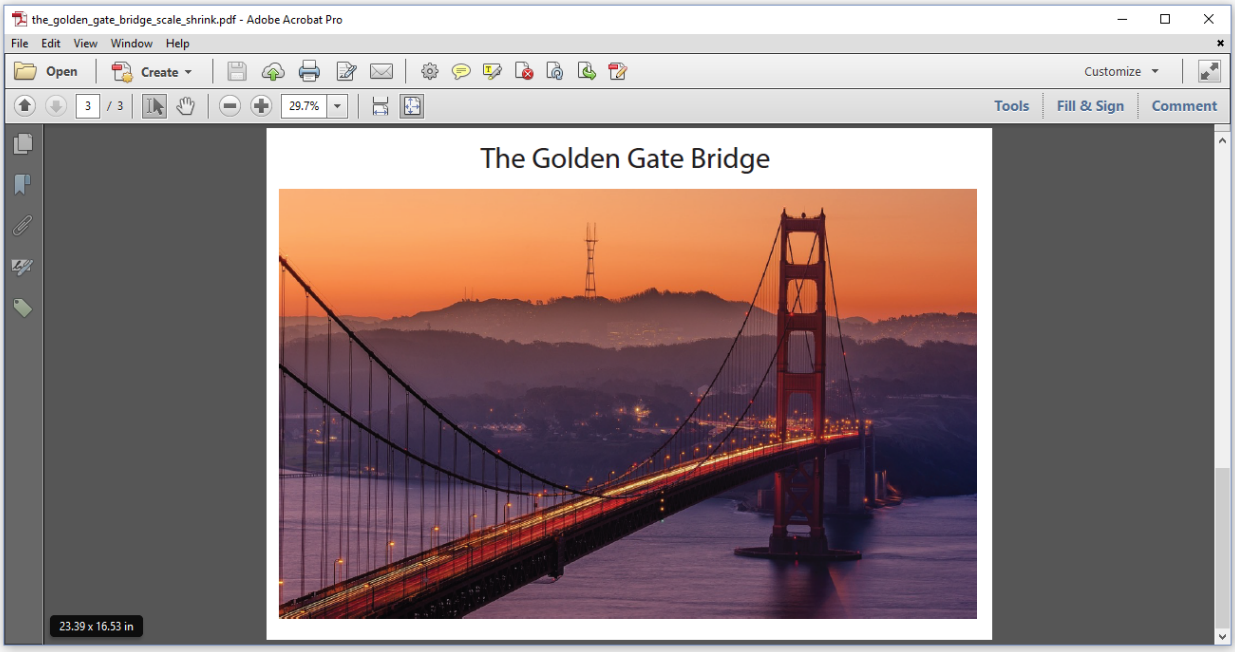


Figure 6.3: Golden Gate Bridge, scaled up to 23.39 x 16.53 in

The TheGoldenGateBridge_Scale_Shrink[38] example shows how it's done.

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  PdfDocument origPdf = new PdfDocument(new PdfReader(src));
3  PdfPage origPage = origPdf.getPage(1);
4  Rectangle orig = origPage.getPageSizeWithRotation();
5
6  //Add A4 page
7  PdfPage page = pdf.addNewPage(PageSize.A4.rotate());
8  //Shrink original page content using transformation matrix
9  PdfCanvas canvas = new PdfCanvas(page);
10 AffineTransform transformationMatrix = AffineTransform.getScaleInstance(
11     page.getPageSize().getWidth() / orig.getWidth(),
12     page.getPageSize().getHeight() / orig.getHeight());
13 canvas.concatMatrix(transformationMatrix);
14 PdfFormXObject pageCopy = origPage.copyAsFormXObject(pdf);
15 canvas.addXObject(pageCopy, 0, 0);
16
17 //Add page with original size
18 pdf.addPage(origPage.copyTo(pdf));
19
20 //Add A2 page
21 page = pdf.addNewPage(PageSize.A2.rotate());
22 //Scale original page content using transformation matrix
23 canvas = new PdfCanvas(page);
24 transformationMatrix = AffineTransform.getScaleInstance(
25     page.getPageSize().getWidth() / orig.getWidth(),
26     page.getPageSize().getHeight() / orig.getHeight());
27 canvas.concatMatrix(transformationMatrix);
28 canvas.addXObject(pageCopy, 0, 0);
29
30 pdf.close();
31 origPdf.close();
```

In this code snippet, we create a `PdfDocument` instance that will create a new PDF document (line 1); and we create a `PdfDocument` instance that will read an existing PDF document (line 2). We get a `PdfPage` instance for the first page of the existing PDF (line 3), and we get its dimensions (line 4). We then add three pages to the new PDF document:

1. We add an A4 page using landscape orientation (line 7) and we create a `PdfCanvas` object for that page. Instead of calculating the `a`, `b`, `c`, `d`, `e`, and `f` value for a transformation

[38] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1782-c06e01_thegoldengatebridge_scale_shrink.java

matrix that will scale the coordinate system, we use an `AffineTransform` instance using the `getScaleInstance()` method (line 9-12). We apply that transformation (line 13), we create a Form XObject containing the original page (line 14) and we add that XObject to the new page (line 15).

2. Adding the original page in its original dimensions is much easier. We just create a new page by copying the `origPage` to the new `PdfDocument` instance, and we add it to the `pdf` using the `addPage()` method (line 18).
3. Scaling up and shrinking is done in the exact same way. This time, we add a new A2 page using landscape orientation (line 21) and we use the exact same code we had before to scale the coordinate system (line 23-27). We reuse the `pageCopy` object and add it to the `canvas` (line 29).

We close the `pdf` to finalize the new document (line 30) and we close the `origPdf` to release the resources of the original document.

We can use the same functionality to tile a PDF page.

Tiling PDF pages

Tiling a PDF page means that you distribute the content of one page over different pages. For instance: if you have a PDF with a single page of size A3, you can create a PDF with four pages of a different size –or even the same size–, each showing one quarter of the original A3 page. This is what we've done in Figure 6.4.

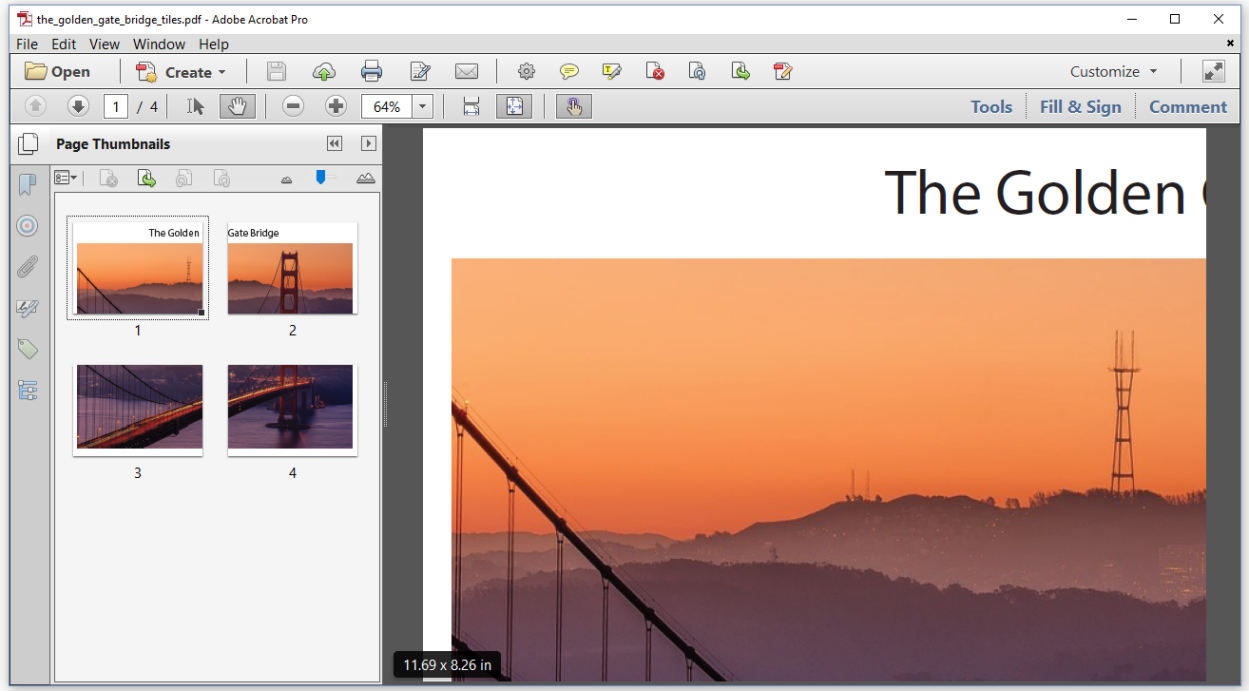


Figure 6.4: Golden Gate Bridge, tiled pages

Let's take a look at the TheGoldenGateBridge_Tiles[39] example.

```
PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
PdfDocument sourcePdf = new PdfDocument(new PdfReader(src));
PdfPage origPage = sourcePdf.getPage(1);
PdfFormXObject pageCopy = origPage.copyAsFormXObject(pdf);
Rectangle orig = origPage.getPageSize();
//Tile size
Rectangle tileSize = PageSize.A4.rotate();
AffineTransform transformationMatrix = AffineTransform.getScaleInstance(
    tileSize.getWidth() / orig.getWidth() * 2f,
    tileSize.getHeight() / orig.getHeight() * 2f);
//The first tile
PdfPage page = pdf.addNewPage(PageSize.A4.rotate());
PdfCanvas canvas = new PdfCanvas(page);
canvas.concatMatrix(transformationMatrix);
canvas.addXObject(pageCopy, 0, -orig.getHeight() / 2f);
//The second tile
page = pdf.addNewPage(PageSize.A4.rotate());
canvas = new PdfCanvas(page);
canvas.concatMatrix(transformationMatrix);
```

[39] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1783-c06e02_thegoldengatebridge_tiles.java

```
20  canvas.addXObject(pageCopy, -orig.getWidth() / 2f, -orig.getHeight() / 2f);
21  //The third tile
22  page = pdf.addNewPage(PageSize.A4.rotate());
23  canvas = new PdfCanvas(page);
24  canvas.concatMatrix(transformationMatrix);
25  canvas.addXObject(pageCopy, 0, 0);
26  //The fourth tile
27  page = pdf.addNewPage(PageSize.A4.rotate());
28  canvas = new PdfCanvas(page);
29  canvas.concatMatrix(transformationMatrix);
30  canvas.addXObject(pageCopy, -orig.getWidth() / 2f, 0);
31  // closing the documents
32  pdf.close();
33  sourcePdf.close();
```

We've seen lines 1-5 before; we already used them in the previous example. In line 7, we define a tile size, and we create a `transformationMatrix` to scale the coordinate system depending on the original size and the tile size. Then we add the tiles, one by one: line 12-15, line 17-20, line 22-25, and line 27-30 are identical, except for one detail: the offset used in the `addXObject()` method.

Let's use the PDF with the Golden Gate Bridge for one more example. Let's do the opposite of tiling: let's N-up a PDF.

N-upping a PDF

Figure 6.5 shows what we mean by N-upping. In the next example, we're going to put N pages on one single page.

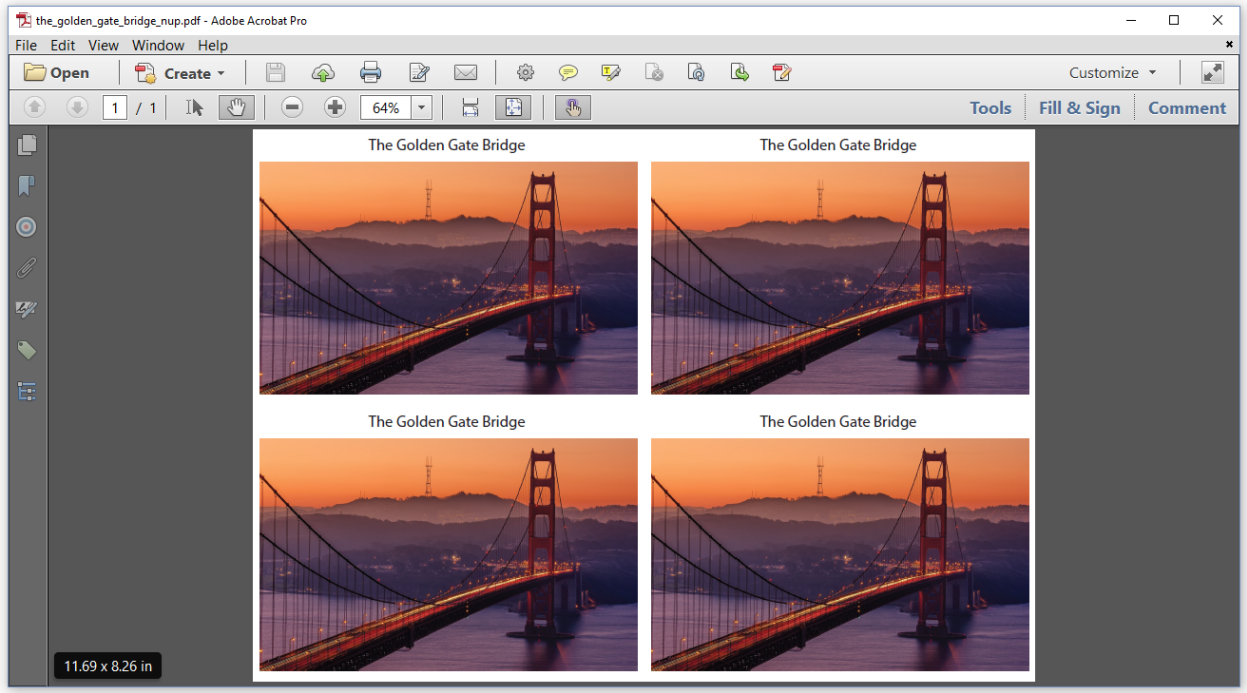


Figure 6.5: Golden Gate Bridge, four pages on one

In the TheGoldenGateBridge_N_up[40] example, N is equal to 4. We will put 4 pages on one single page.

```
PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
PdfDocument sourcePdf = new PdfDocument(new PdfReader(SRC));
//Original page
PdfPage origPage = sourcePdf.getPage(1);
Rectangle orig = origPage.getPageSize();
PdfFormXObject pageCopy = origPage.copyAsFormXObject(pdf);
//N-up page
PageSize nUpPageSize = PageSize.A4.rotate();
PdfPage page = pdf.addNewPage(nUpPageSize);
PdfCanvas canvas = new PdfCanvas(page);
//Scale page
AffineTransform transformationMatrix = AffineTransform.getScaleInstance(
    nUpPageSize.getWidth() / orig.getWidth() / 2f,
    nUpPageSize.getHeight() / orig.getHeight() / 2f);
canvas.concatMatrix(transformationMatrix);
//Add pages to N-up page
canvas.addXObject(pageCopy, 0, orig.getHeight());
canvas.addXObject(pageCopy, orig.getWidth(), orig.getHeight());
```

[40] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1784-c06e03_thegoldengatebridge_n_up.java

```
19  canvas.addXObject(pageCopy, 0, 0);
20  canvas.addXObject(pageCopy, orig.getWidth(), 0);
21  // close the documents
22  pdf.close();
23  sourcePdf.close();
```

So far, we've only reused a single page from a single PDF in this chapter. In the next series of examples, we'll assemble different PDF files into one.

Assembling documents

Let's go from San Francisco to Los Angeles, and take a look at Figure 6.6 where we'll find three documents about the Oscars.

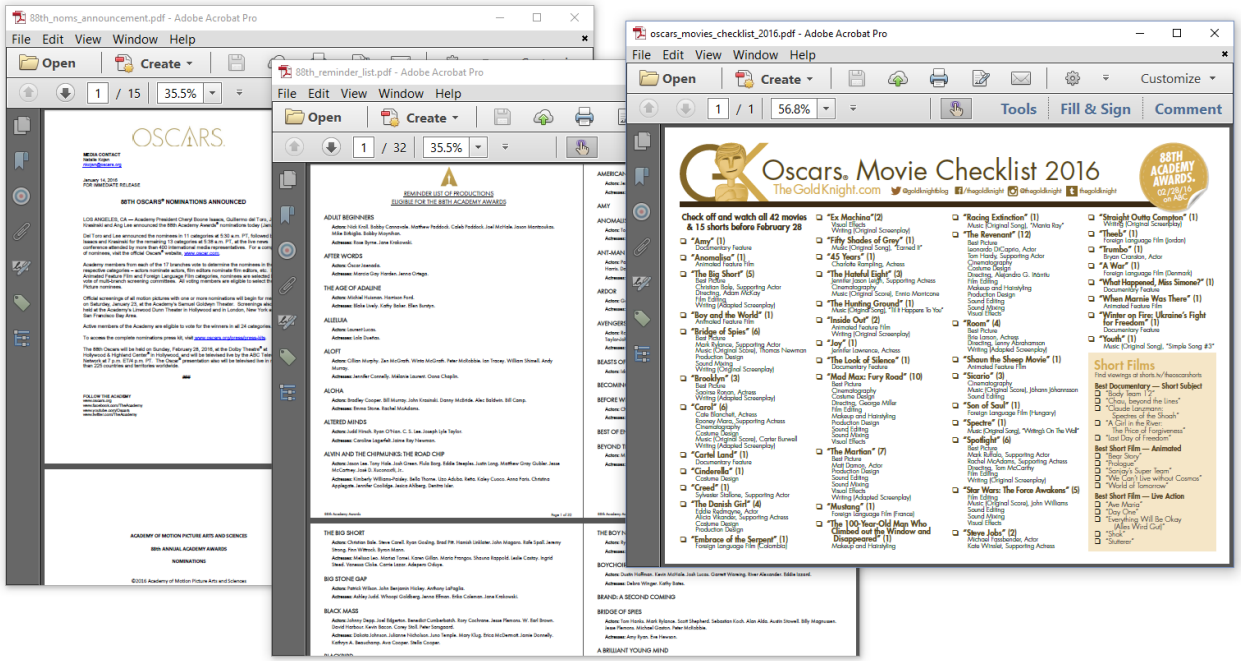


Figure 6.6: The Oscars, source documents

The documents are:

- 88th_reminder_list.pdf[41]: a 32-page document, entitled "Reminder List of Productions Eligible for the 88th Academy Awards",
- 88th_noms_announcement.pdf[42]: a 15-page document, entitled "Oscars"

[41] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/88th_reminder_list.pdf
[42] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/88th_noms_announcement.pdf

- oscars_movies_checklist_2016.pdf[43]: a 1-page document, entitled "Oscars Movie Checklist 2016"

In the next couple of examples, we'll merge these documents.

Merging documents with PdfMerger

Figure 6.7 shows a PDF that was created by merging the first 32-page document with the second 15-page document, resulting in a 47-page document.

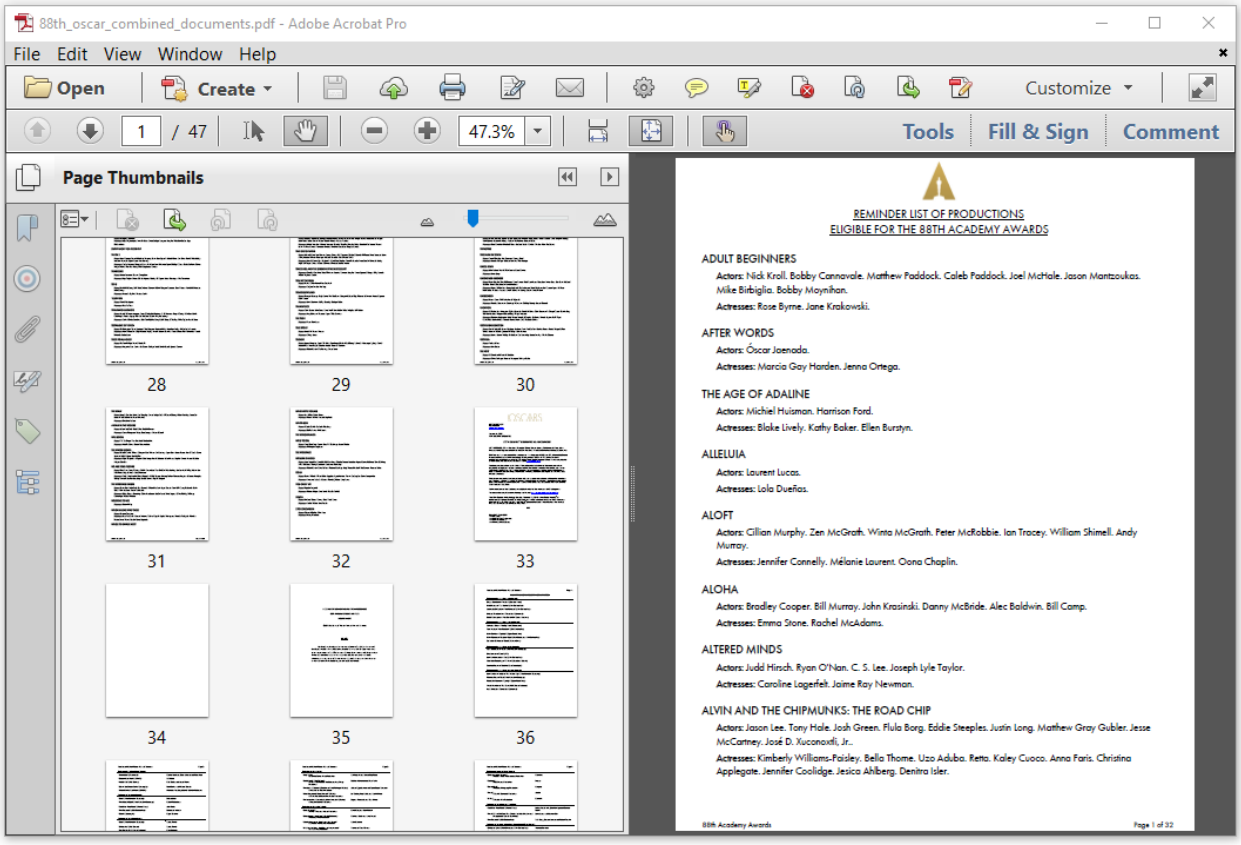


Figure 6.7: Merging two documents

The code of the 88th_Oscar_Combine[44] example is almost self-explaining.

[43] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/oscars_movies_checklist_2016.pdf
[44] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1785-c06e04_88th_oscar_combine.java

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  PdfMerger merger = new PdfMerger(pdf);
3  //Add pages from the first document
4  PdfDocument firstSourcePdf = new PdfDocument(new PdfReader(SRC1));
5  merger.addPages(firstSourcePdf, 1, firstSourcePdf.getNumberOfPages());
6   //Add pages from the second pdf document
7  PdfDocument secondSourcePdf = new PdfDocument(new PdfReader(SRC2));
8  merger.addPages(secondSourcePdf, 1, secondSourcePdf.getNumberOfPages());
9  // merge and close
10 merger.merge();
11 firstSourcePdf.close();
12 secondSourcePdf.close();
13 pdf.close();
```

We create a `PdfDocument` to create a new PDF (line 1). The `PdfMerger` class is new. It's a class that will make it easier for us to reuse pages from existing documents (line 2). Just like before, we create a `PdfDocument` for the source file (line 4, line 7); we then add all the pages to the `merger` instance (line 5, line 8). Once we're done adding pages, we `merge()` (line 10) and `close()` (line 11-13).

We don't need to add *all* the pages if we don't want to. We can easily add only a limited selection of pages. See for instance the 88th_Oscar_CombineXofY[45] example.

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  PdfMerger merger = new PdfMerger(pdf);
3  PdfDocument firstSourcePdf = new PdfDocument(new PdfReader(SRC1));
4  merger.addPages(firstSourcePdf, Arrays.asList(1, 5, 7, 1));
5  PdfDocument secondSourcePdf = new PdfDocument(new PdfReader(SRC2));
6  merger.addPages(secondSourcePdf, Arrays.asList(1, 15));
7  merger.merge();
8  firstSourcePdf.close();
9  secondSourcePdf.close();
10 pdf.close();
```

Now the resulting document only has six pages. Pages 1, 5, 7, 1 from the first document (the first page is repeated), and pages 1 and 15 from the second document. `PdfMerger` is a convenience class that makes merging documents a no-brainer. In some cases however, you'll want to add pages one by one.

Adding pages to a PdfDocument

Figure 6.8 shows the result of the merging of specific pages based on a Table of Contents (TOC) that we'll create on the fly. This TOC contains link annotations that allow you to jump to a specific page if you click an entry of the TOC.

[45] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1786-c06e05_88th_oscar_combinexofy.java

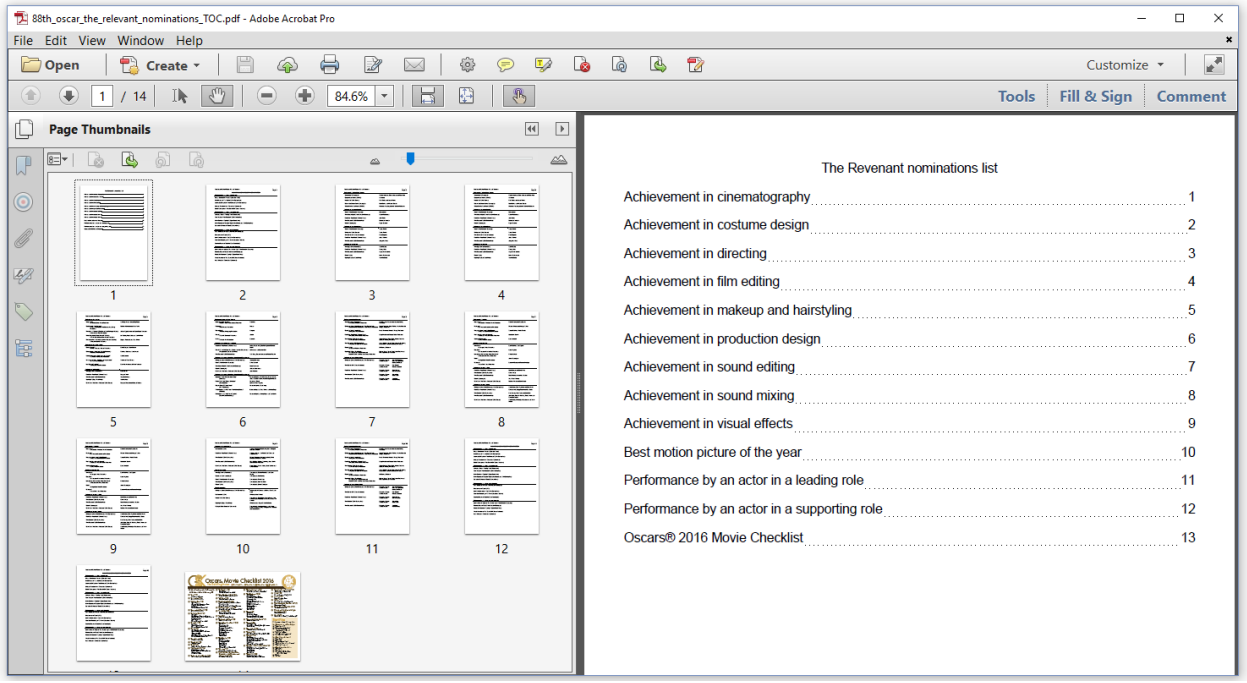


Figure 6.8: Merging documents based on a TOC

The 88th_Oscar_Combine_AddTOC[46] example is more complex than the two previous examples. Let's examine it step by step.

Suppose that we have a `TreeMap` of all the categories the move "The Revenant" was nominated for, where the key is the nomination and the value is the page number of the document where the nomination is mentioned.

```
public static final Map<String, Integer> TheRevenantNominations =
    new TreeMap<String, Integer>();
static {
    TheRevenantNominations.put("Performance by an actor in a leading role", 4);
    TheRevenantNominations.put(
        "Performance by an actor in a supporting role", 4);
    TheRevenantNominations.put("Achievement in cinematography", 4);
    TheRevenantNominations.put("Achievement in costume design", 5);
    TheRevenantNominations.put("Achievement in directing", 5);
    TheRevenantNominations.put("Achievement in film editing", 6);
    TheRevenantNominations.put("Achievement in makeup and hairstyling", 7);
    TheRevenantNominations.put("Best motion picture of the year", 8);
    TheRevenantNominations.put("Achievement in production design", 8);
    TheRevenantNominations.put("Achievement in sound editing", 9);
    TheRevenantNominations.put("Achievement in sound mixing", 9);
```

[46] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1787-c06e06_88th_oscar_combine_addtoc.java

```
    TheRevenantNominations.put("Achievement in visual effects", 10);
}
```

The first lines of the code that creates the PDF are pretty simple.

```
PdfDocument pdfDoc = new PdfDocument(new PdfWriter(dest));
Document document = new Document(pdfDoc);
document.add(new Paragraph(new Text("The Revenant nominations list"))
    .setTextAlignment(Property.TextAlignment.CENTER));
```

But we need to take a really close look once we start to loop over the entries in the `TreeMap`.

```
PdfDocument firstSourcePdf = new PdfDocument(new PdfReader(SRC1));
for (Map.Entry<String, Integer> entry : TheRevenantNominations.entrySet()) {
    //Copy page
    PdfPage page  = firstSourcePdf.getPage(entry.getValue()).copyTo(pdfDoc);
    pdfDoc.addPage(page);
    //Overwrite page number
    Text text = new Text(String.format(
        "Page %d", pdfDoc.getNumberOfPages() - 1));
    text.setBackgroundColor(Color.WHITE);
    document.add(new Paragraph(text).setFixedPosition(
            pdfDoc.getNumberOfPages(), 549, 742, 100));
    //Add destination
    String destinationKey = "p" + (pdfDoc.getNumberOfPages() - 1);
    PdfArray destinationArray = new PdfArray();
    destinationArray.add(page.getPdfObject());
    destinationArray.add(PdfName.XYZ);
    destinationArray.add(new PdfNumber(0));
    destinationArray.add(new PdfNumber(page.getMediaBox().getHeight()));
    destinationArray.add(new PdfNumber(1));
    pdfDoc.addNameDestination(destinationKey, destinationArray);
    //Add TOC line with bookmark
    Paragraph p = new Paragraph();
    p.addTabStops(
        new TabStop(540, Property.TabAlignment.RIGHT, new DottedLine()));
    p.add(entry.getKey());
    p.add(new Tab());
    p.add(String.valueOf(pdfDoc.getNumberOfPages() - 1));
    p.setProperty(Property.ACTION, PdfAction.createGoTo(destinationKey));
    document.add(p);
}
firstSourcePdf.close();
```

Here we go:

- Line 1: we create a `PdfDocument` with the source file containing all the info about all the nominations.
- Line 2: we loop over an alphabetic list of the nominations for "The Revenant".
- Line 3-4: we get the page that corresponds with the nomination, and we add a copy to the `PdfDocument`.
- Line 7-8: we create an iText `Text` element containing the page number. We subtract 1 from that page number, because the first page in our document is the unnumbered page containing the TOC.
- Line 9: we set the background color to `Color.WHITE`. This will cause an opaque white rectangle to be drawn with the same size of the `Text`. We do this to cover the original page number.
- Line 10-11: we add this `text` at a fixed position on the the current page in the `PdfDocument`. The fixed position is: X = 549, Y = 742, and the width of the text is 100 user units.
- Line 13: we create a key we'll use to name the destination.
- Line 14-19: we create a `PdfArray` containing information about the destination. We'll refer to the page we've just added (line 15), we'll define the destination using an X,Y coordinate and a zoom factor (line 16), we add the values of X (line 17), Y (line 18), and the zoom factor (line 19).
- Line 20: we add the named destination to the `PdfDocument`.
- Line 22: we create an empty `Paragraph`.
- Line 23-24: we add a tab stop at position X = 540, we define that the tab needs to be right aligned, and the space preceding the tab needs to be a `DottedLine`.
- Line 25: we add the nomination to the `Paragraph`.
- Line 26: we introduce a `Tab`.
- Line 27: we add the page number minus 1 (because the page with the TOC is page 0).
- Line 28: we add an action that will be triggered when someone clicks on the `Paragraph`.
- Line 29: we add the `Paragraph` to the `document`.
- Line 31: we close the source document.

We've been introducing a lot of new functionality that really requires a more in-depth tutorial, but we're looking at this example for one main reason: to show that there's a significant difference between the `PdfDocument` object, to which a new page is added with every pass through the loop, and the `Document` object, to which we keep adding `Paragraph` objects on the first page.

Let's go through some of these steps one more time to add the checklist.

```
//Add the last page
PdfDocument secondSourcePdf = new PdfDocument(new PdfReader(SRC2));
PdfPage page  = secondSourcePdf.getPage(1).copyTo(pdfDoc);
pdfDoc.addPage(page);
//Add destination
PdfArray destinationArray = new PdfArray();
destinationArray.add(page.getPdfObject());
destinationArray.add(PdfName.XYZ);
destinationArray.add(new PdfNumber(0));
destinationArray.add(new PdfNumber(page.getMediaBox().getHeight()));
destinationArray.add(new PdfNumber(1));
pdfDoc.addNameDestination("checklist", destinationArray);
//Add TOC line with bookmark
Paragraph p = new Paragraph();
p.addTabStops(new TabStop(540, Property.TabAlignment.RIGHT, new DottedLine()));
p.add("Oscars\u00ae 2016 Movie Checklist");
p.add(new Tab());
p.add(String.valueOf(pdfDoc.getNumberOfPages() - 1));
p.setProperty(Property.ACTION, PdfAction.createGoTo("checklist"));
document.add(p);
secondSourcePdf.close();
// close the document
document.close();
```

This code snippet adds the check list with the overview of all the nominations. An extra line saying "Oscars® 2016 Movie Checklist" is added to the TOC.

This example introduces a couple of new concepts for educational purposes. It shouldn't be used in a real-world application, because it contains a major flaw. We make the assumption that the TOC will consist of only one page. Suppose that we added more lines to the `document` object, then you would see a strange phenomenon: the text that doesn't fit on the first page, would be added on the second page. This second page wouldn't be a new page, it would be the first page that we added in the loop. In other words: the content of the first imported page would be overwritten. This is a problem that can be fixed, but it's outside the scope of this short introductory tutorial.

We'll finish this chapter with some examples in which we merge forms.

Merging forms

Merging forms is special. In HTML, it's possible to have more than one form in a single HTML file. That's not the case for PDF. In a PDF file, there can be only one form. If you want to merge

two forms and you want to preserve the forms, you need to use a special method and a special `IPdfPageExtraCopier` implementation.

Figure 6.9 shows the combination of two different forms, subscribe.pdf[47] and state.pdf[48]

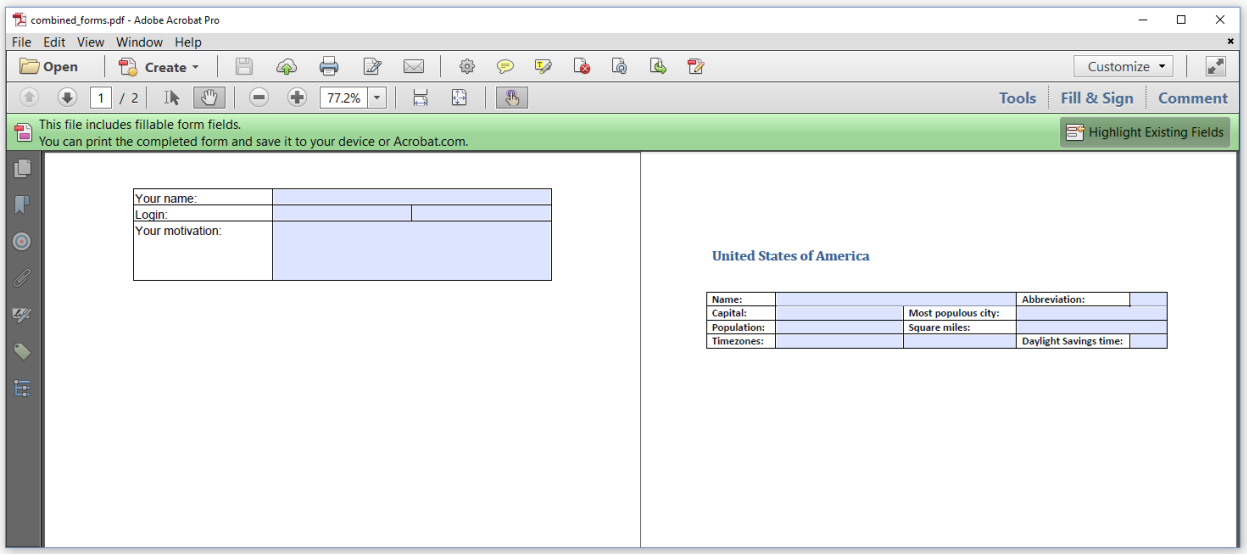


Figure 6.9: merging two different forms

The Combine_Forms[49] example is different from what we had before.

```
PdfDocument destPdfDocument = new PdfDocument(new PdfWriter(dest));
PdfDocument[] sources = new PdfDocument[] {
        new PdfDocument(new PdfReader(SRC1)),
        new PdfDocument(new PdfReader(SRC2))
};
for (PdfDocument sourcePdfDocument : sources) {
    sourcePdfDocument.copyPagesTo(
            1, sourcePdfDocument.getNumberOfPages(),
            destPdfDocument, new PdfPageFormCopier());
    sourcePdfDocument.close();
}
destPdfDocument.close();
```

In this code snippet, we use the `copyPageTo()` method. The first two parameters define the *from*/*to* range for the pages of the source document. The third parameter defines the destination document. The fourth parameter indicates that we are copying forms and that the two different forms in the two different documents should be merged into a single form. `PdfPageFormCopier` is an implementation

[47] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/subscribe.pdf
[48] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/state.pdf
[49] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1788-c06e07_combine_forms.java

of the `IPdfPageExtraCopier` interface that makes sure that the two different forms are merged into one single form.

Merging two forms isn't always trivial, because the name of each field needs to be unique. Suppose that we would merge the same form twice. Then we would have two widget annotations for each field. A field with a specific name, for instance `"name"`, can be visualized using different widget annotations, but it can only have one value. Suppose that you would have a widget annotation for the field `"name"` on page one, and a widget annotation for the same field on page two, then changing the value shown in the widget annotation on one page would automatically also change the value shown in the widget annotations on the other page.

In the next example, we are going to fill out and merge the same form, state.pdf[50], as many times as there are entries in the CSV file united_states.csv[51]; see Figure 6.10.

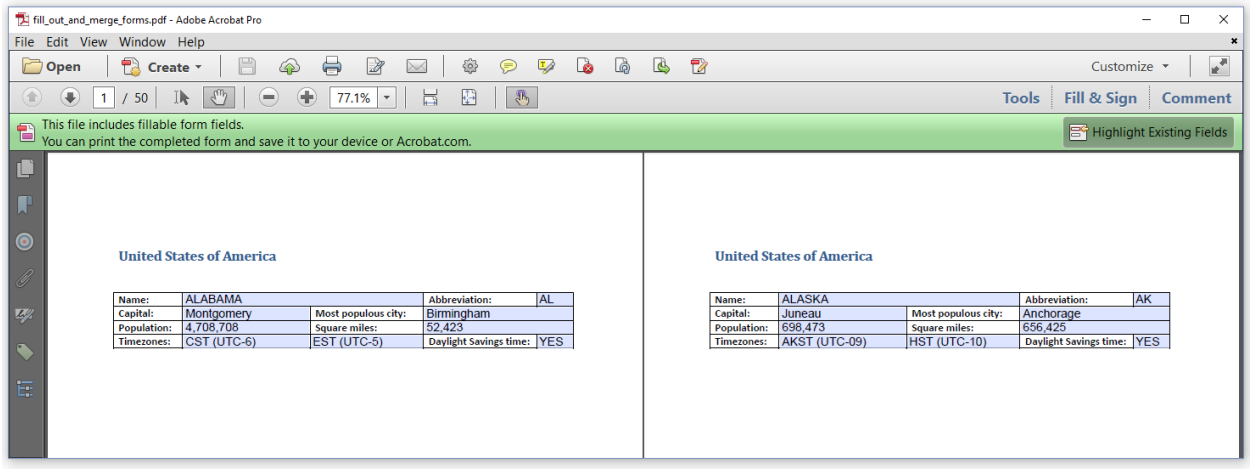


Figure 6.10: Merging identical forms

If we'd keep the names of the fields the way they are in the original form, changing the value of the state "ALABAMA" into "CALIFORNIA", would also change the name "ALASKA" on the second page, and the name of all the other states on the other pages. We made sure that this doesn't happen by renaming all the fields before merging the forms.

Let's take a look at the FillOutAndMergeForms[52] example.

[50] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/pdf/state.pdf

[51] http://gitlab.itextsupport.com/itext7/samples/raw/develop/publications/jumpstart/src/main/resources/data/united_states.csv

[52] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1789-c06e08_filloutandmergeforms.java

```
1  PdfDocument pdfDocument = new PdfDocument(new PdfWriter(dest));
2  BufferedReader bufferedReader = new BufferedReader(new FileReader(DATA));
3  String line;
4  boolean headerLine = true;
5  int i = 1;
6  while ((line = bufferedReader.readLine()) != null) {
7      if (headerLine) {
8          headerLine = false;
9          continue;
10     }
11     ByteArrayOutputStream baos = new ByteArrayOutputStream();
12     PdfDocument sourcePdfDocument = new PdfDocument(
13             new PdfReader(SRC), new PdfWriter(baos));
14     //Rename fields
15     i++;
16     PdfAcroForm form = PdfAcroForm.getAcroForm(sourcePdfDocument, true);
17     form.renameField("name", "name_" + i);
18     form.renameField("abbr", "abbr_" + i);
19     // ... (removed repetitive lines)
20     form.renameField("dst", "dst_" + i);
21     //Fill out fields
22     StringTokenizer tokenizer = new StringTokenizer(line, ";");
23     Map<String, PdfFormField> fields = form.getFormFields();
24     fields.get("name_" + i).setValue(tokenizer.nextToken());
25     fields.get("abbr_" + i).setValue(tokenizer.nextToken());
26     // ... (removed repetitive lines)
27     fields.get("dst_" + i).setValue(tokenizer.nextToken());
28     // close the source document and use it to create a new PdfDocument
29     sourcePdfDocument.close();
30     sourcePdfDocument = new PdfDocument(
31         new PdfReader(new ByteArrayInputStream(baos.toByteArray())));
32     //Copy pages
33     sourcePdfDocument.copyPagesTo(
34         1, sourcePdfDocument.getNumberOfPages(),
35         pdfDocument, new PdfPageFormCopier());
36     sourcePdfDocument.close();
37 }
38 bufferedReader.close();
39 pdfDocument.close();
```

Let's start by looking at the code inside the `while` loop. We're looping over the different states of the USA stored in a CSV file (line 6). We skip the first line that contains the information for the column headers (line 7-10). The next couple of lines are interesting. So far, we've always been writing PDF

files to disk. In this example, we are creating PDF files in memory using a `ByteArrayOutputStream` (line 11-13).

As mentioned before, we start by renaming all the fields. We get the `PdfAcroForm` instance (line 16) and we use the `renameField()` method to rename fields such as `"name"` to `"name_1"`, `"name_2"`, and so on. Note that we've skipped some lines for brevity in the code snippet. Once we've renamed all the fields, we set their value (line 23-27).

When we close the `sourcePdfDocument` (line 29), we have a complete PDF file in memory. We create a new `sourcePdfDocument` using a `ByteArrayInputStream` created with that file in memory (line 30-31). We can now copy the pages of that new `sourcePdfDocument` to our destination `pdfDocument`.

This is a rather artificial example, but it's a good example to explain some of the usual pitfalls when merging forms:

- Without the `PdfPageFormCopier`, the forms won't be correctly merged.
- One field can only have one value, no matter how many times that field is visualized using a widget annotation.

A more common use case, is to fill out *and flatten* the same form multiple times in memory, simultaneously merging all the resulting documents in one PDF.

Merging flattened forms

Figure 6.11 shows two PDF documents that were the result of the same procedure: we filled out a form in memory as many times as there are states in the USA. We flattened these filled out forms, and we merged them into one single document.

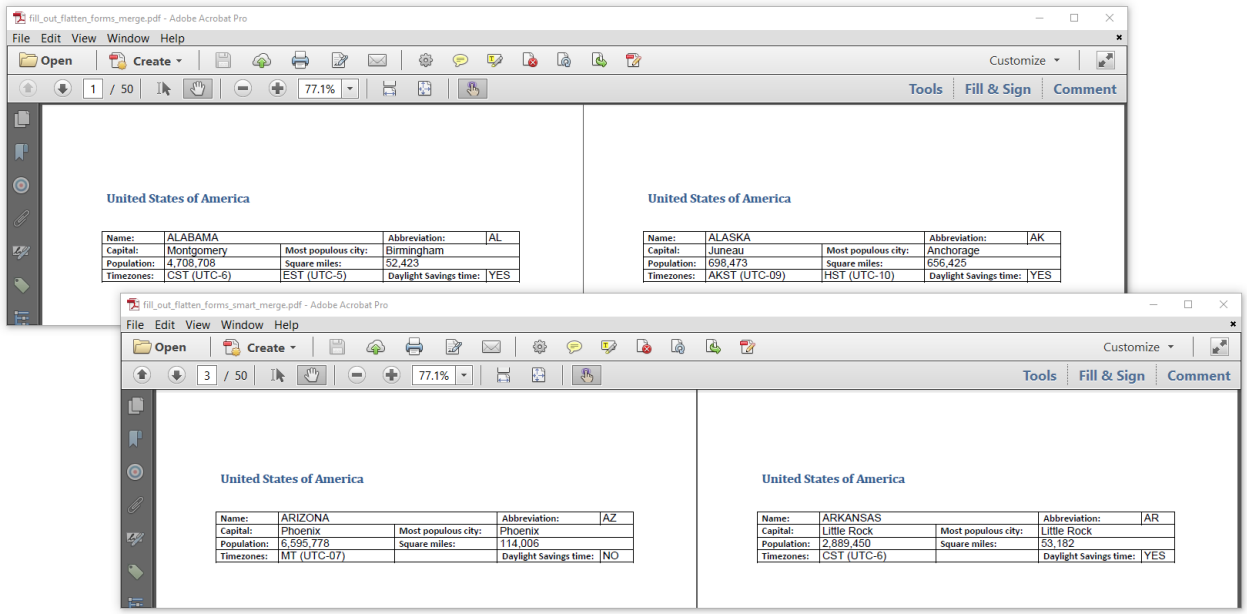


Figure 6.11: Filling, flattening and merging forms

From the outside, these documents look identical, but if we look at their file size in Figure 12, we see a huge difference.

Name	Date modified	Type	Size
fill_out_flatten_forms_merge.pdf	4/10/2016 17:31 PM	Adobe Acrobat D...	12,930 KB
fill_out_flatten_forms_smart_merge.pdf	4/10/2016 17:31 PM	Adobe Acrobat D...	365 KB

Figure 6.12: difference in file size depending on how documents are merged

What is causing this difference in file size? We need to take a look at the FillOutFlattenAndMergeForms[53] example to find out.

```
1  PdfDocument destPdfDocument =
2      new PdfDocument(new PdfWriter(dest1));
3  PdfDocument destPdfDocumentSmartMode =
4      new PdfDocument(new PdfWriter(dest2).setSmartMode(true));
5  BufferedReader bufferedReader = new BufferedReader(new FileReader(DATA));
6  String line;
7  boolean headerLine = true;
8  while ((line = bufferedReader.readLine()) != null) {
9      if (headerLine) {
10         headerLine = false;
11         continue;
```

[53] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1790-c06e09_filloutflattenandmergeforms.java

```
12      }
13      ByteArrayOutputStream baos = new ByteArrayOutputStream();
14      PdfDocument sourcePdfDocument =
15          new PdfDocument(new PdfReader(SRC), new PdfWriter(baos));
16      //Fill out fields
17      PdfAcroForm form = PdfAcroForm.getAcroForm(sourcePdfDocument, true);
18      StringTokenizer tokenizer = new StringTokenizer(line, ";");
19      Map<String, PdfFormField> fields = form.getFormFields();
20      fields.get("name").setValue(tokenizer.nextToken());
21      fields.get("abbr").setValue(tokenizer.nextToken());
22      // ... (removed repetitive lines)
23      fields.get("dst").setValue(tokenizer.nextToken());
24      //Flatten fields
25      form.flattenFields();
26      sourcePdfDocument.close();
27      sourcePdfDocument = new PdfDocument(
28          new PdfReader(new ByteArrayInputStream(baos.toByteArray())));
29      //Copy pages
30      sourcePdfDocument.copyPagesTo(
31          1, sourcePdfDocument.getNumberOfPages(), destPdfDocument, null);
32      sourcePdfDocument.copyPagesTo(
33          1, sourcePdfDocument.getNumberOfPages(), destPdfDocumentSmartMode, null);
34      sourcePdfDocument.close();
35  }
36  bufferedReader.close();
37  destPdfDocument.close();
38  destPdfDocumentSmartMode.close();
```

In this code snippet, we create two documents simultaneously:

- The `destPdfDocument` instance (line 1-2) is created the same way we've been creating `PdfDocument` instances all along.
- The `destPdfDocumentSmartMode` instance (line 3-4) is also created that way, but we've turned on the *smart mode*.

We loop over the lines of the CSV file like we did before (line 8), but since we're going to flatten the forms, we no longer have to rename the fields. The fields will be lost due to the flattening process anyway. We create a new PDF document in memory (line 12-15) and we fill out the fields (line 17-23). We flatten the fields (line 25) and close the document created in memory (line 26). We use the file created in memory to create a new source file. We add all the pages of this source file to the two `PdfDocument` instances, one working in *normal mode*, the other in *smart mode*. We no longer need

to use a `PdfPageFormCopier` instance, because the forms have been flattened; they are no longer forms.

What is the difference between these *normal* and *smart* mode?

- When we copy the pages of the filled out forms to the `PdfDocument` working in *normal mode*, the `PdfDocument` processes each document as if it's totally unrelated to the other documents that are being added. In this case, the resulting document will be bloated, because the documents *are* related: they all share the same template. That template is added to the PDF document as many times as there are states in the USA. In this case, the result is a file of about 12 MBytes.
- When we copy the pages of the filled out forms to the `PdfDocument` working in *smart mode*, the `PdfDocument` will take the time to compare the resources of each document. If two separate documents share the same resources (e.g. a template), then that resource is copied to the new file only once. In this case, the result can be limited to 365 KBytes.

Both the 12 MBytes and the 365 KBytes files look exactly the same when opened in a PDF viewer or when printed, but it goes without saying that the 365 KBytes files is to be preferred over the 12 MBytes file.

Summary

In this chapter, we've been scaling, tiling, N-upping one file with a different file as result. We've also assembled files in many different ways. We discovered that there are quite some pitfalls when merging interactive forms. Much more remains to be said about reusing content from existing PDF documents.

In the next chapter, we'll discuss PDF documents that comply to special PDF standards such as PDF/UA and PDF/A. We'll discover that merging PDF/A documents also requires some special attention.

Chapter 7: Creating PDF/UA and PDF/A documents

In chapter 1 to 4, we've created PDF documents using iText 7. In chapters 5 and 6, we've manipulated and reused existing PDF documents. All the PDFs we dealt with in those chapters were PDF documents that complied to ISO 32000, which is the core standard for PDF. ISO 32000 isn't the only ISO standard for PDF, there are many different sub-standards that were created for specific reasons. In this chapter, we'll highlight two:

- ISO 14289 is better known as PDF/UA. UA stands for Universal Accessibility. PDFs that comply with the PDF/UA standard can be consumed by anyone, including people who are blind or visually impaired.
- ISO 19005 is better known as PDF/A. A stands for Archiving. The goal of this standard is the long-term preservation of digital documents.

In this chapter, we'll learn more about PDF/A and PDF/UA by creating a series of PDF/A and PDF/UA files.

Creating accessible PDF documents

Before we start with a PDF/UA example, let's take a closer look at the problem we want to solve. In chapter 1, we created a document that included images. In the sentence "Quick brown fox jumps over the lazy dog", we replaced the words "fox" and "dog" by images representing a fox and a dog. When this file is read out loud, a machine doesn't know that the first image represents a fox and that the second image represents a dog, hence the file will be read as "Quick brown jumps over the lazy."

In an ordinary PDF, content is painted to a canvas. We might use high-level objects such as `List` and `Table`, but once the PDF is created, there is no structure left. A list is a sequence of lines and a text snippet in a list item doesn't know that it's part of a list. A table is just a bunch of lines and text added at absolute positions on a page. A text snippet in a table doesn't know it belongs to a cell in a specific column and a specific row.

Unless we make the PDF a tagged PDF, the document doesn't contain any semantic structure. When there's no semantic structure, the PDF isn't accessible. To be accessible, the document needs to be able to distinguish which part of a page is actual content, and which part is an artifact that isn't part

of the actual content (e.g. a header, a page number). A line of text needs to know if its a title, if it's part of a paragraph, and so on. We can add all of this information to the page, by creating a *structure tree* and by defining content as *marked content*. This sounds complex, but if you use iText 7's high-level objects, it's sufficient to introduce the method `setTagged()`. By defining a `PdfDocument` as a tagged document, the structure we introduce by using objects such as `List`, `Table`, `Paragraph`, will be reflected in the Tagged PDF.

This is only one requirement to make a PDF accessible. The QuickBrownFox_PDFUA[54] example will help us understand the other requirements.

```
1  PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
2  Document document = new Document(pdf);
3  //Setting some required parameters
4  pdf.setTagged();
5  pdf.getCatalog().setLang(new PdfString("en-US"));
6  pdf.getCatalog().setViewerPreferences(
7          new PdfViewerPreferences().setDisplayDocTitle(true));
8  PdfDocumentInfo info = pdf.getDocumentInfo();
9  info.setTitle("iText7 PDF/UA example");
10 //Create XMP meta data
11 pdf.createXmpMetadata();
12 //Fonts need to be embedded
13 PdfFont font = PdfFontFactory.createFont(FONT, PdfEncodings.WINANSI, true);
14 Paragraph p = new Paragraph();
15 p.setFont(font);
16 p.add(new Text("The quick brown "));
17 Image foxImage = new Image(ImageFactory.getImage(FOX));
18 //PDF/UA: Set alt text
19 foxImage.getAccessibilityProperties().setAlternateDescription("Fox");
20 p.add(foxImage);
21 p.add(" jumps over the lazy ");
22 Image dogImage = new Image(ImageFactory.getImage(DOG));
23 //PDF/UA: Set alt text
24 dogImage.getAccessibilityProperties().setAlternateDescription("Dog");
25 p.add(dogImage);
26 document.add(p);
27 document.close();
```

We create a `PdfDocument` and a `Document` as usual (line 1-2). We make sure that we comply to PDF by introducing some extra features:

1. We tell the `PdfDocument` that we're going to create Tagged PDF (line 4),

[54] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1807-c07e01_quickbrownfox_pdfua.java

2. We add a language specifier. In our case, the document knows that the main language used in this document is American English (line 5).
3. We change the viewer preferences so that the title of the document is always displayed in the top bar of the PDF viewer (line 6-7). Obviously, this implies that we add a title to the metadata of the document (line 8-9).
4. When we've set the title, we added metadata to a PDF object known as the *Info dictionary*. In PDF/UA, it is mandatory to have the same metadata stored in the PDF as XML. This XML may not be compressed. Processors that don't "understand" PDF must be able to detect this XMP metadata and process it. An XMP stream is created automatically based on the entries in the Info dictionary when using the method `createXmpMetadata()` (line 11).
5. All fonts need to be embedded (line 13). There are some other requirements relating to fonts, but it would lead us too far right now to discuss these in detail.
6. All the content needs to be tagged. When an image is encountered, we need to provide a description of that image using *alt text* (line 19 and line 24).

We have now created a PDF/UA document. When we look at the resulting page in Figure 7.1, we don't see much difference, but if we open the Tags panel, we see that the document has a specific structure.

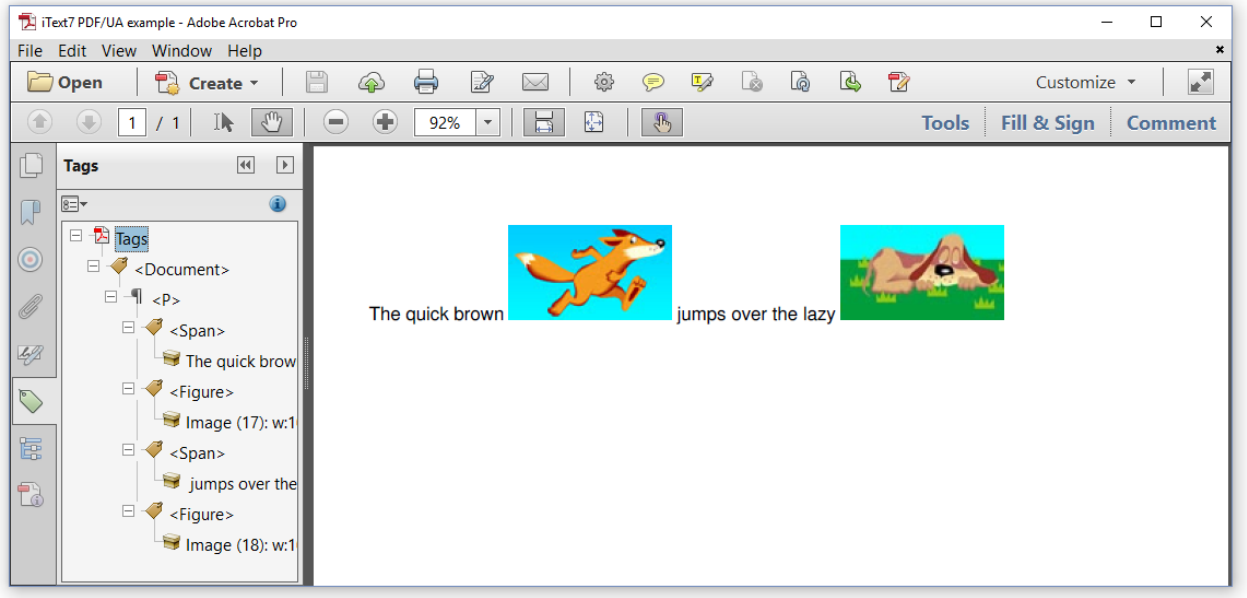


Figure 7.1: a PDF/UA document and its structure

We see that the `<Document>` consists of a `<P>`aragraph that is composed of four parts, two `<Span>`s and two `<Figures>`s. We'll create a more complex PDF/UA document later in this chapter, but let's take a look at what makes PDF/A special first.

Creating PDFs for long-term preservation, part 1

Part 1 of ISO 19005 was released in 2005. It was defined as a subset of version 1.4 of Adobe's PDF specification (which, at that time, wasn't an ISO standard yet). ISO 19005-1 introduced a series of obligations and restrictions:

- *The document needs to be self-contained:* all fonts need to be embedded; external movie, sound or other binary files are not allowed.
- *The document needs to contain metadata in the eXtensible Metadata Platform (XMP) format:* ISO 16684 (XMP) describes how to embed XML metadata into a binary file, so that software that doesn't know how to interpret the binary data format can still extract the file's metadata.
- *Functionality that isn't future-proof isn't allowed:* the PDF can't contain any JavaScript and may not be encrypted.

ISO 19005-1:2005 (PDF/A-1) defined two conformance levels:

- *Level B ("basic"):* ensures that the visual appearance of a document will be preserved for the long term.
- *Level A ("accessible"):* ensures that the visual appearance of a document will be preserved for the long term, but also introduces structural and semantic properties. The PDF needs to be a *Tagged PDF*.

The QuickBrownFox_PDFA_1b[55] example shows how we can create a "Quick brown fox" PDF that complies to PDF/A-1b.

```
1  //Initialize PDFA document with output intent
2  PdfADocument pdf = new PdfADocument(new PdfWriter(dest),
3      PdfAConformanceLevel.PDF_A_1B,
4      new PdfOutputIntent("Custom", "", "http://www.color.org",
5              "sRGB IEC61966-2.1", new FileInputStream(INTENT)));
6  Document document = new Document(pdf);
7  //Create XMP meta data
8  pdf.createXmpMetadata();
9  //Fonts need to be embedded
10 PdfFont font = PdfFontFactory.createFont(FONT, PdfEncodings.WINANSI, true);
11 Paragraph p = new Paragraph();
12 p.setFont(font);
13 p.add(new Text("The quick brown "));
14 Image foxImage = new Image(ImageFactory.getImage(FOX));
```

[55] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1809-c07e02_quickbrownfox_pdfa_1b.java

```
15  p.add(foxImage);
16  p.add(" jumps over the lazy ");
17  Image dogImage = new Image(ImageFactory.getImage(DOG));
18  p.add(dogImage);
19  document.add(p);
20  document.close();
```

The first thing that jumps to the eye, is that we are no longer using a `PdfDocument` instance. Instead, we create a `PdfADocument` instance. The `PdfADocument` constructor needs a `PdfWriter` as its first parameter, but also a conformance level (in this case `PdfAConformanceLevel.PDF_A_1B`) and a `PdfOutputIntent`. This output intent tells the document how to interpret the colors that will be used in the document. In line 8, we create the XMP metadata stream. In line 10, we make sure that the font we're using is embedded.

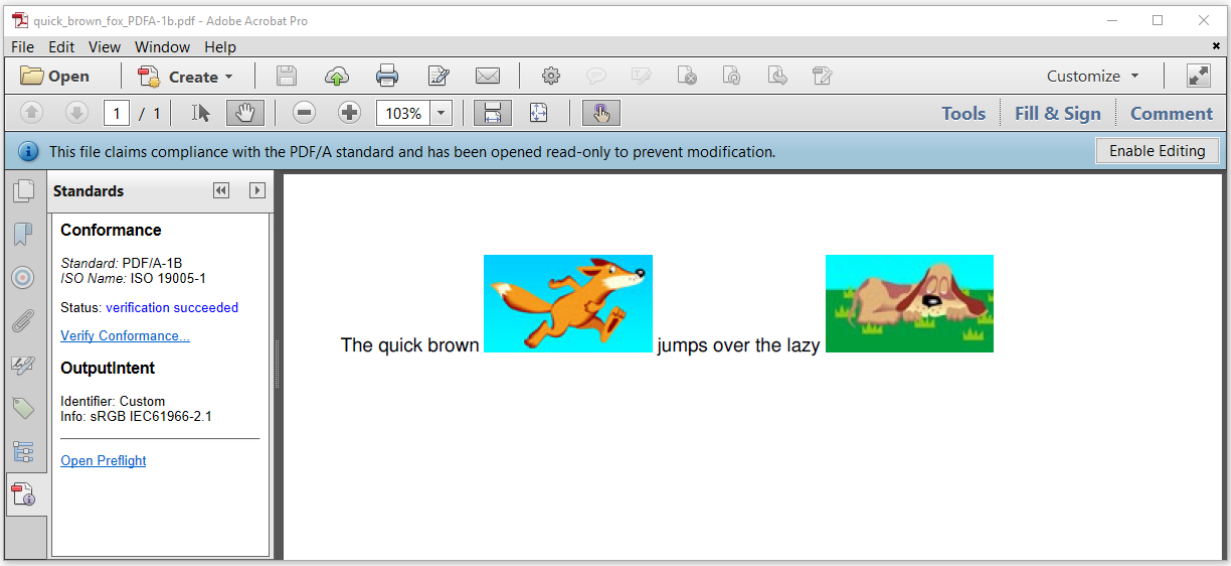


Figure 7.2: a PDF/A-1 level B document

Looking at the PDF shown in Figure 7.2, we see a blue ribbon with the text "This file claims compliance with the PDF/A standard and has been opened read-only to prevent modification." Allow me to explain two things about this sentence:

1. This doesn't mean that the PDF is, in effect, compliant with the PDF/A standard. It only claims it is. To be sure, you need to open the Standards panel in Adobe Acrobat. When you click on the "Verify Conformance" link, Acrobat will verify if the document is what it claims to be. In this case, we read "Status: verification succeeded"; we have successfully created a document complying with PDF/A-1B.
2. The document has been opened read-only, not because you are not allowed to modify it (PDF/A is not a way to protect a PDF against modification), but Adobe Acrobat presents it as read-only because any modification might change the PDF into a PDF that is no longer

compliant to the PDF/A standard. It's not trivial to update a PDF/A without breaking its PDF/A status.

Let's adapt our example, and create a PDF/A-1 level A document with the QuickBrownFox_PDFA_1a[56] example.

```
1  //Initialize PDFA document with output intent
2  PdfADocument pdf = new PdfADocument(new PdfWriter(dest),
3      PdfAConformanceLevel.PDF_A_1A,
4      new PdfOutputIntent("Custom", "", "http://www.color.org",
5              "sRGB IEC61966-2.1", new FileInputStream(INTENT)));
6  Document document = new Document(pdf);
7  //Setting some required parameters
8  pdf.setTagged();
9  //Create XMP meta data
10 pdf.createXmpMetadata();
11 //Fonts need to be embedded
12 PdfFont font = PdfFontFactory.createFont(FONT, PdfEncodings.WINANSI, true);
13 Paragraph p = new Paragraph();
14 p.setFont(font);
15 p.add(new Text("The quick brown "));
16 Image foxImage = new Image(ImageFactory.getImage(FOX));
17 //Set alt text
18 foxImage.getAccessibilityProperties().setAlternateDescription("Fox");
19 p.add(foxImage);
20 p.add(" jumps over the lazy ");
21 Image dogImage = new Image(ImageFactory.getImage(DOG));
22 //Set alt text
23 dogImage.getAccessibilityProperties().setAlternateDescription("Dog");
24 p.add(dogImage);
25 document.add(p);
26 document.close();
```

We've changed `PdfAConformanceLevel.PDF_A_1B` into `PdfAConformanceLevel.PDF_A_1A` in line 3. We've made the `PdfADocument` a Tagged PDF (line 8) and we've added some *alt text* for the images. Figure 7.3 is somewhat confusing.

[56] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1808-c07e02_quickbrownfox_pdfa_1a.java

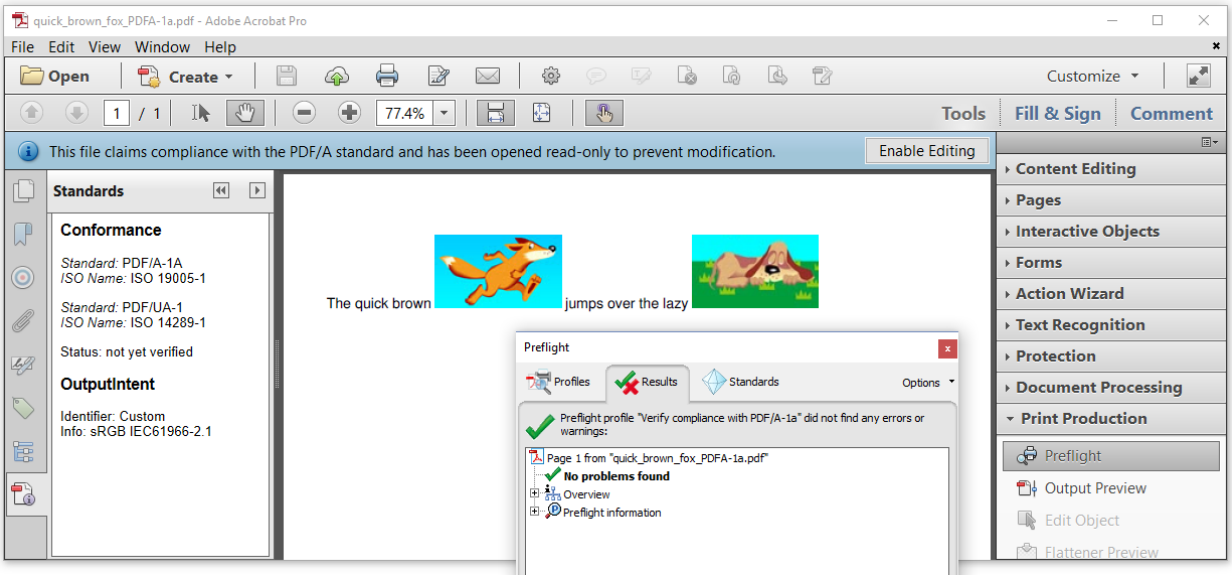


Figure 7.3: a PDF/A-1 level A document

When we look at the Standards panel, we see that the document thinks it conforms to PDF/A-1A *and* to PDF/UA-1. We don't have a "Verify Conformance" link, so we have to use Preflight. Preflight informs us that there were "No problems found" when executing the "Verify compliance with PDF/A-1a" profile. We can't verify the PDF/UA compliance because PDF/UA involves some requirements that can't be verified by a machine. For instance: a machine wouldn't notice if we switched the description of the image of the fox with the description of the image of the dog. That would make the document inaccessible as the document would spread false information to people depending on screen-readers. In any case, we know that our document doesn't comply to the PDF/UA standard because we omitted a number of essential elements (such as the language).

From the start, it was determined that approved parts of ISO 19005 could never become invalid. New, subsequent parts would only define new, useful features. That's what happened when part 2 and part 3 were created.

Creating PDFs for long-term preservation, part 2 and 3

ISO 19005-2:2011 (PDF/A-2) was introduced to have a PDF/A standard that was based on the ISO standard (ISO 32000-1) instead of on Adobe's PDF specification. PDF/A-2 also adds a handful of features that were introduced in PDF 1.5, 1.6 and 1.7:

- *Useful additions include:* support for JPEG2000, Collections, object-level XMP, and optional content.
- *Useful improvements include:* better support for transparency, comment types and annotations, and digital signatures.

PDF/A-2 also defines an extra level besides Level A and Level B:

- *Level U ("Unicode"):* ensures that the visual appearance of a document will be preserved for the long term, and that all text is stored in UNICODE.

ISO 19005-3:2012 (PDF/A-3) was an almost identical copy of PDF/A-2. There was only one difference with PDF/A-2: in PDF/A-3, attachments don't need to be PDF/A. You can attach any file to a PDF/A-3, for instance: an XLS file containing calculations of which the results are used in the document, the original Word document that was used to create the PDF document, and so on. The document itself needs to conform to all the obligations and restrictions of the PDF/A specification, but these obligations and restrictions do not apply to its attachments.

In the UnitedStates_PDFA_3a[57] example, we'll create a document that complies with PDF/UA as well as with PDF/A-3A. We choose PDF/A3, because we're going to add the CSV file that was used as the source for creating the PDF.

```
PdfADocument pdf = new PdfADocument(new PdfWriter(dest),
    PdfAConformanceLevel.PDF_A_3A,
    new PdfOutputIntent("Custom", "", "http://www.color.org",
            "sRGB IEC61966-2.1", new FileInputStream(INTENT)));
Document document = new Document(pdf, PageSize.A4.rotate());
//Setting some required parameters
pdf.setTagged();
pdf.getCatalog().setLang(new PdfString("en-US"));
pdf.getCatalog().setViewerPreferences(
        new PdfViewerPreferences().setDisplayDocTitle(true));
PdfDocumentInfo info = pdf.getDocumentInfo();
info.setTitle("iText7 PDF/A-3 example");
//Create XMP meta data
pdf.createXmpMetadata();
//Add attachment
PdfDictionary parameters = new PdfDictionary();
parameters.put(PdfName.ModDate, new PdfDate().getPdfObject());
PdfFileSpec fileSpec = PdfFileSpec.createEmbeddedFileSpec(
    pdf, Files.readAllBytes(Paths.get(DATA)), "united_states.csv",
    "united_states.csv", new PdfName("text/csv"), parameters,
    PdfName.Data, false);
fileSpec.put(new PdfName("AFRelationship"), new PdfName("Data"));
pdf.addFileAttachment("united_states.csv", fileSpec);
PdfArray array = new PdfArray();
array.add(fileSpec.getPdfObject().getIndirectReference());
pdf.getCatalog().put(new PdfName("AF"), array);
//Embed fonts
```

[57] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1810-c07e03_unitedstates_pdfa_3a.java

```
28  PdfFont font = PdfFontFactory.createFont(FONT, true);
29  PdfFont bold = PdfFontFactory.createFont(BOLD_FONT, true);
30  // Create content
31  Table table = new Table(new float[]{4, 1, 3, 4, 3, 3, 3, 3, 1});
32  table.setWidthPercent(100);
33  BufferedReader br = new BufferedReader(new FileReader(DATA));
34  String line = br.readLine();
35  process(table, line, bold, true);
36  while ((line = br.readLine()) != null) {
37      process(table, line, font, false);
38  }
39  br.close();
40  document.add(table);
41  //Close document
42  document.close();
```

Let's examine the different parts of this example.

- Line 1-5: We create a `PdfADocument` (`PdfAConformanceLevel.PDF_A_3A`) and a `Document`.
- Line 7: Making the PDF a Tagged PDF is a requirement for PDF/UA as well as for PDF/A-3A.
- Line 8-12: Setting the language, the document title and the viewer preference to display the title is a requirement for PDF/UA.
- Line 14: Adding XMP metadata is a requirement for PDF/UA as well as for PDF/A.
- Line 16-22: We add a file attachment using specific parameters that are required for PDF/A-3A.
- Line 28-29: We embed the fonts which is a requirement for PDF/UA as well as for PDF/A.
- Line 30-40: We've seen this code before in the UnitedStates[58] example in chapter 1 (including the `process()` method).
- Line 42: We close the document.

Figure 7.4 demonstrates how using the `Table` class with `Cell` objects added as header cells, and `Cell` objects added as normal cells, resulted in a structure tree that makes the PDF document accessible.

[58] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1726-c01e04_unitedstates.java

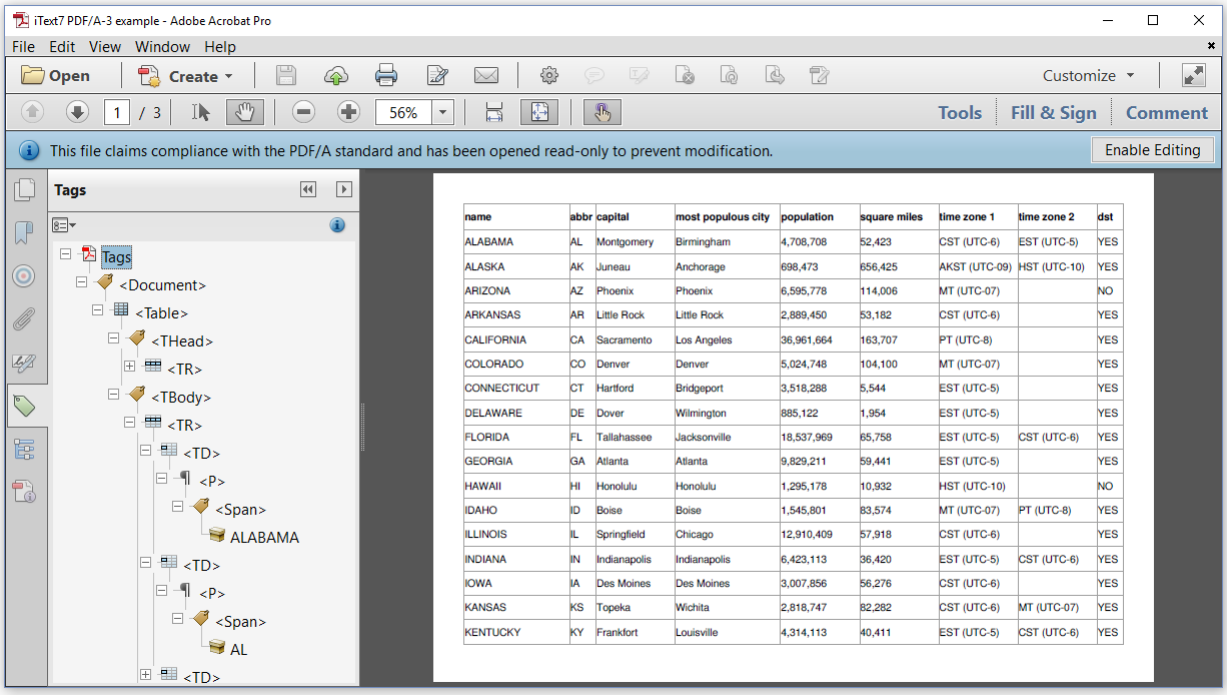


Figure 7.4: a PDF/A-3 level A document

When we open the Attachments panel as shown in Figure 7.5, we see our original united_states.csv file that we can easily extract from the PDF.

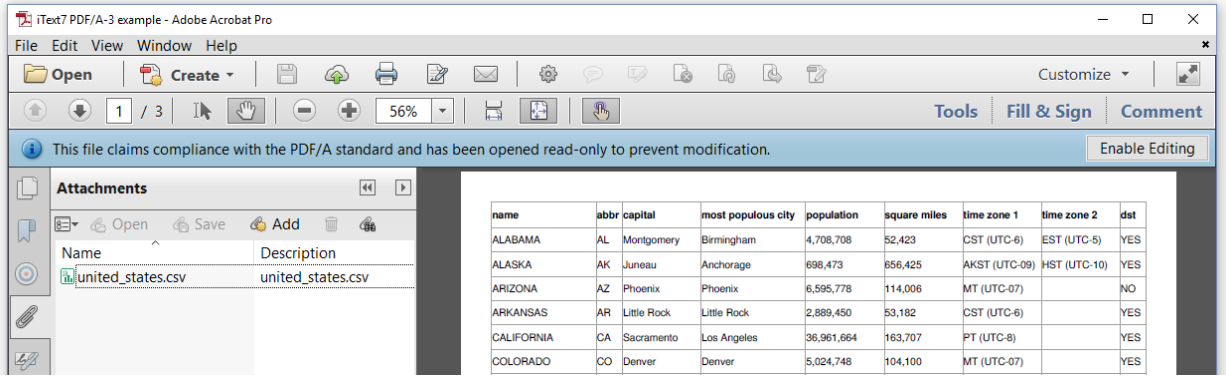


Figure 7.5: a PDF/A-3 level A document and its attachment

The examples in this chapter taught us that PDF/UA or PDF/A documents involve extra requirements when compared to ordinary PDFs. *"Can we use iText to convert an existing PDF to a PDF/UA or PDF/A document"* is a question that is posted frequently on mailing-lists or user forums. I hope that this chapter explains that iText can't do this automatically.

- If you have a document that has a picture of a fox and a dog, iText can't add any missing *alt text* for those images, because iText can't see that fox nor that dog. iText only sees pixels, it can't interpret the image.

- If you are using a font that isn't embedded, iText doesn't know what that font looks like. If you don't provide the corresponding font program, iText can never embed that font.

These are only two examples of many that explain why converting an ordinary PDF to PDF/A or PDF/UA isn't trivial. It's very easy to change the PDF so that it shows a blue bar saying that the document complies to PDF/A, but that doesn't many that claim is true.

We also need to pay attention when we merge existing PDF/A documents.

Merging PDF/A documents

When merging PDF/A documents, it's very important that every single document that you are adding to `PdfMerger` is already a PDF/A document. You can't mix PDF/A documents and ordinary PDF documents into one single PDF and hope the result will be a PDF/A document. The same is true for mixing a PDF/A level A document with a PDF/A level B document. One has a structure tree, the other hasn't; you can't expect the resulting PDF to be a PDF/A level A document.

Figure 7.6 shows how we merged the two PDF/A level A documents we created in the previous sections.

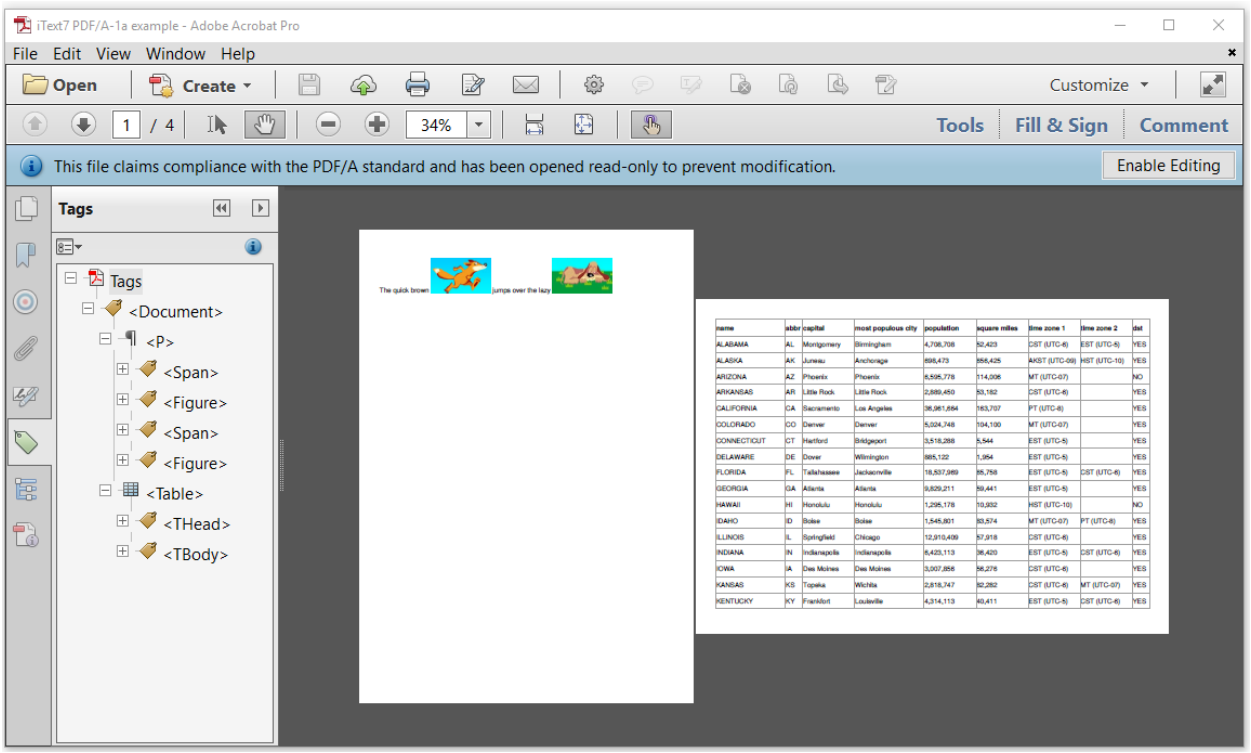


Figure 7.6: merging 2 PDF/A level A documents

When we look at the structure of the tags, we see that the <P>aragraph is now followed by a <Table>. The MergePDFADocuments[59] shows how it's done.

```
1  PdfADocument pdf = new PdfADocument(new PdfWriter(dest),
2      PdfAConformanceLevel.PDF_A_1A,
3      new PdfOutputIntent("Custom", "", "http://www.color.org",
4              "sRGB IEC61966-2.1", new FileInputStream(INTENT)));
5  //Setting some required parameters
6  pdf.setTagged();
7  pdf.getCatalog().setLang(new PdfString("en-US"));
8  pdf.getCatalog().setViewerPreferences(
9          new PdfViewerPreferences().setDisplayDocTitle(true));
10 PdfDocumentInfo info = pdf.getDocumentInfo();
11 info.setTitle("iText7 PDF/A-1a example");
12 //Create XMP meta data
13 pdf.createXmpMetadata();
14 //Create PdfMerger instance
15 PdfMerger merger = new PdfMerger(pdf);
16 //Add pages from the first document
17 PdfDocument firstSourcePdf = new PdfDocument(new PdfReader(SRC1));
18 merger.addPages(firstSourcePdf, 1, firstSourcePdf.getNumberOfPages());
19 //Add pages from the second pdf document
20 PdfDocument secondSourcePdf = new PdfDocument(new PdfReader(SRC2));
21 merger.addPages(secondSourcePdf, 1, secondSourcePdf.getNumberOfPages());
22 //Merge
23 merger.merge();
24 //Close the documents
25 firstSourcePdf.close();
26 secondSourcePdf.close();
27 pdf.close();
```

This example is assembled using parts of two examples we've already seen before:

- Lines 1 to 13 are almost identical to the first part of the UnitedStates_PDFA_3a[60] example we've used in the previous section, except that we now use `PdfAConformanceLevel.PDF_A_1A` and that we don't need a `Document` object.
- Lines 14 to 27 are identical to the last part of the 88th_Oscar_Combine[61] example of the previous chapter. Note that we use a `PdfDocument` instance instead of a `PdfADocument`; the `PdfADocument` will check if the source documents comply.

[59] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1811-c07e04_mergepdfadocuments.java
[60] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-7#1810-c07e03_unitedstates_pdfa_3a.java
[61] http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-6#1785-c06e04_88th_oscar_combine.java

There's a lot more to be said about PDF/UA and PDF/A, and even about other sub-standards. For instance: there's a German standard for invoicing called ZUGFeRD[62] that is built on top of PDF/A-3, but let's save that for another tutorial.

Summary

In this chapter, we've discovered that there's more to PDF than meets the eye. We've learned how to introduce structure into our documents so that they are accessible for the blind and the visually impaired. We've also made sure that our PDFs were self-contained, for instance by embedding fonts, so that our documents can be archived for the long term.

We'll need several other tutorials to cover the functionality covered in this tutorial in more depth, but these seven chapters should already give you a good impression of what you can do with iText 7.

[62] http://developers.itextpdf.com/content/zugferd-future-invoicing