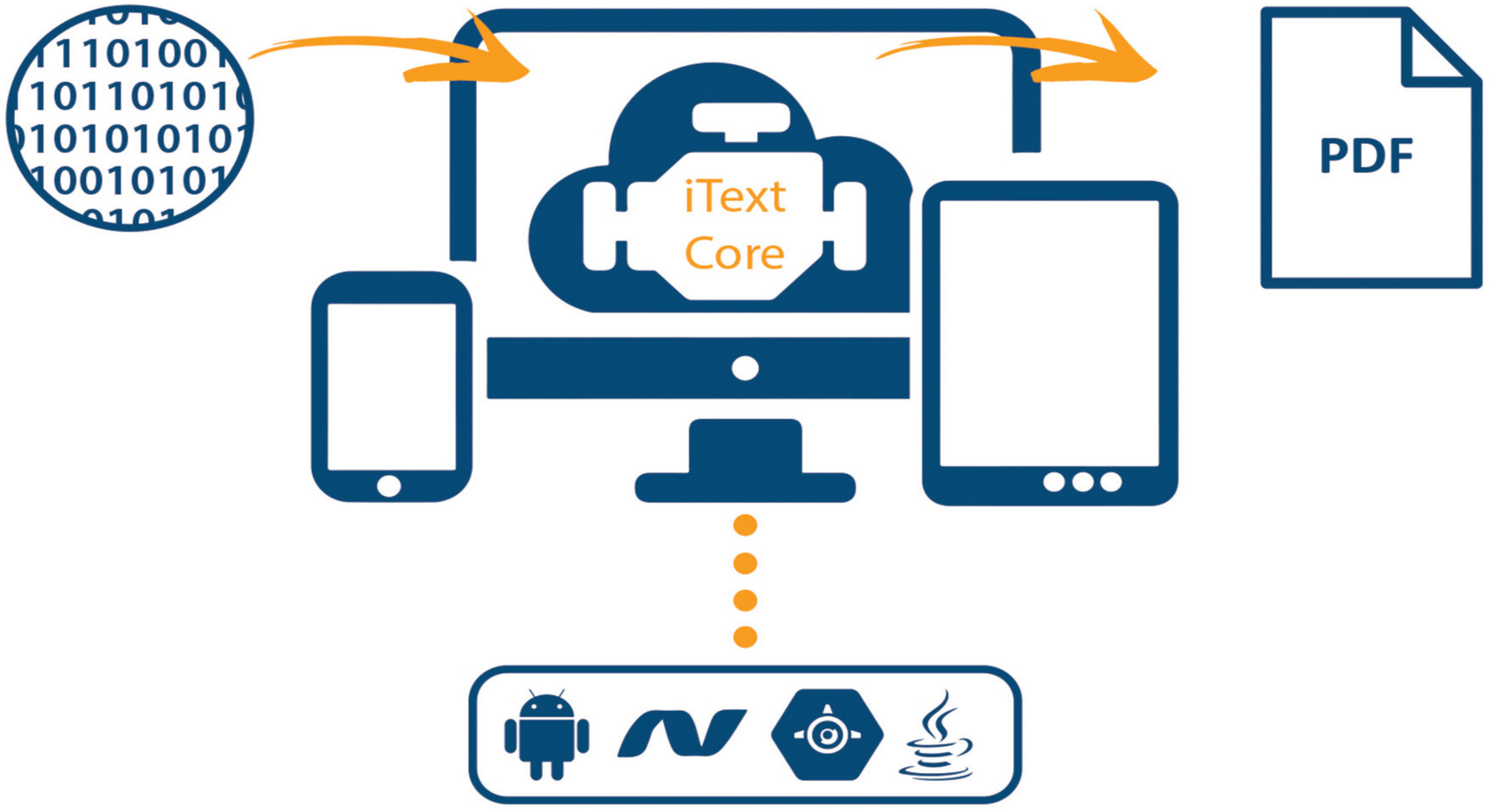
iText7 Jump-Start Tutorial



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:

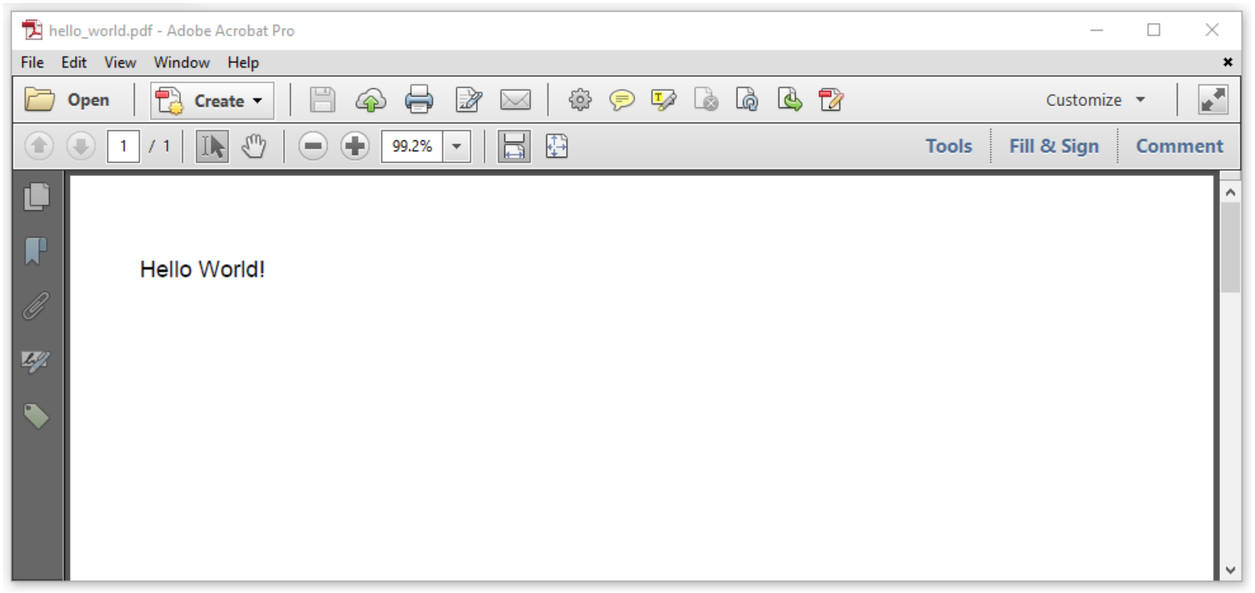
[6]http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/examples/chapter-1#1723-c01e01_helloworld.java

```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.

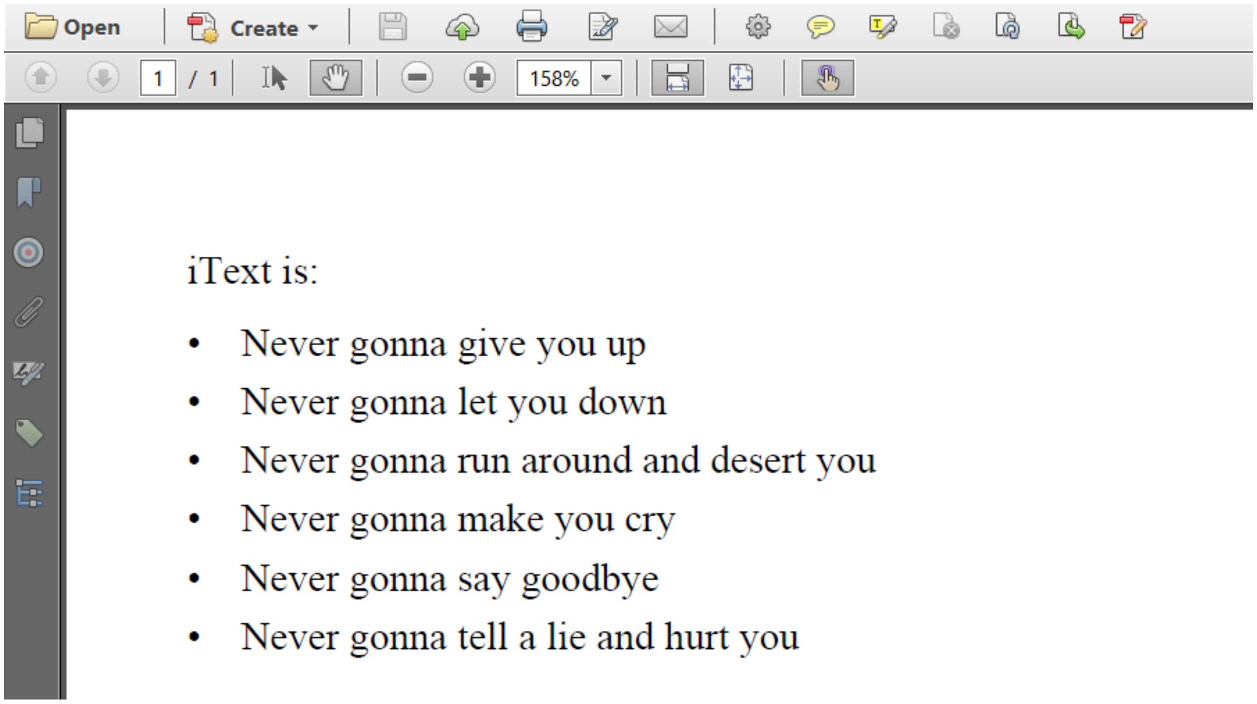
Figure 1.1 shows what the resulting PDF document looks like:


hello_world.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
99.2%
Tools
Fill & Sign
Comment

Hello World!

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.



iText is:

- Never gonna give you up
- Never gonna let you down
- Never gonna run around and desert you
- Never gonna make you cry
- Never gonna say goodbye
- Never gonna tell a lie and hurt you

Figure 1.2: List example

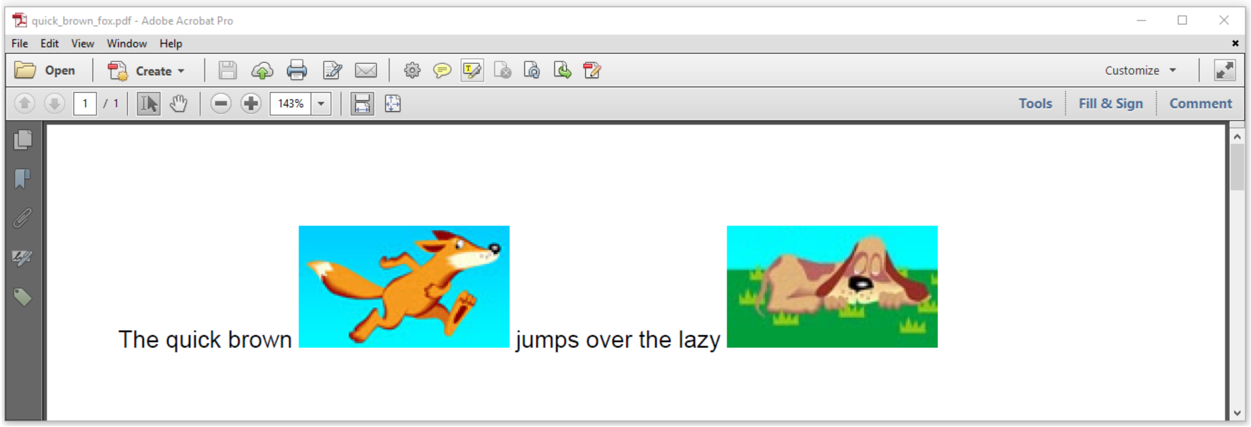
The RickAstley[7] example shows how this is done:

[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`.


quick_brown_fox.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
143%
Tools
Fill & Sign
Comment

The quick brown [fox] jumps over the lazy [dog]

Figure 1.3: Image example

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

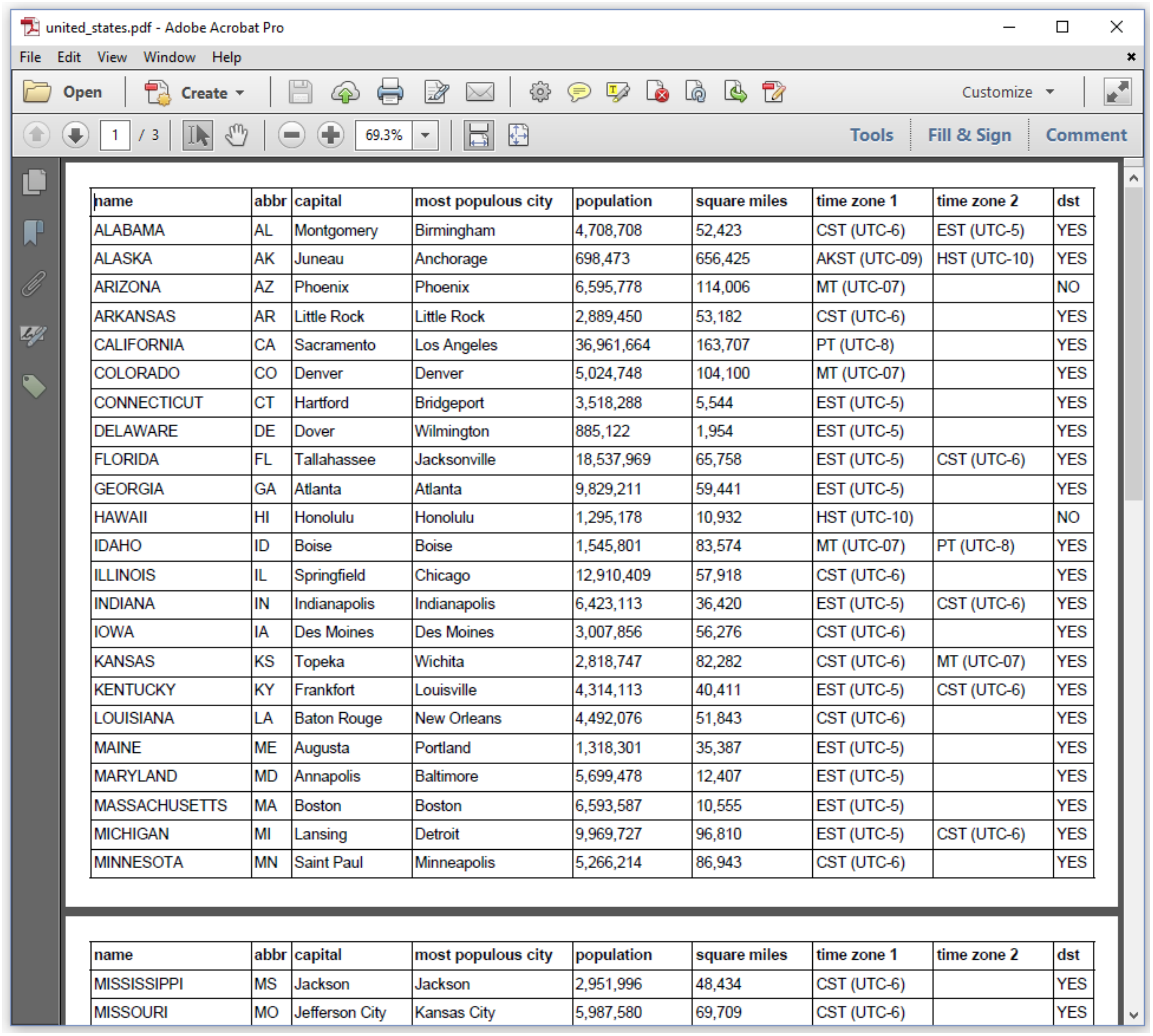
```java
Image fox = new Image(ImageFactory.getImage(FOX));
Image dog = new Image(ImageFactory.getImage(DOG));
Paragraph p = new Paragraph("The quick brown ")
            .add(fox)
            .add(" jumps over the lazy ")
            .add(dog);
document.add(p);
```

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


united_states.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 3
69.3%
Tools
Fill & Sign
Comment

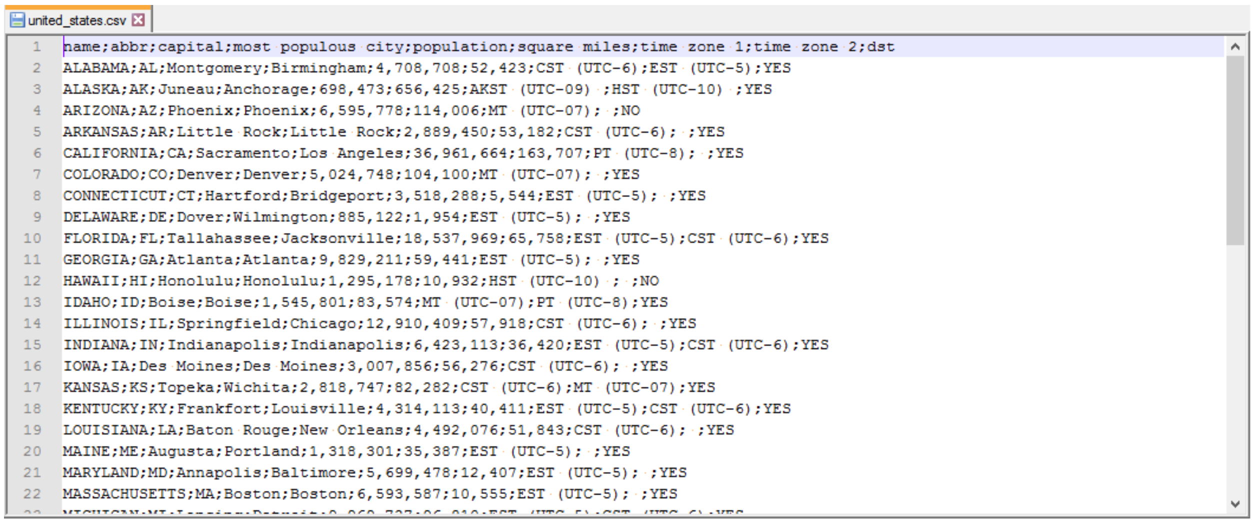
name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
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



```
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.

```java
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:

```
canvas.moveTo(-406, 0)
              .lineTo(406, 0)
              .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.

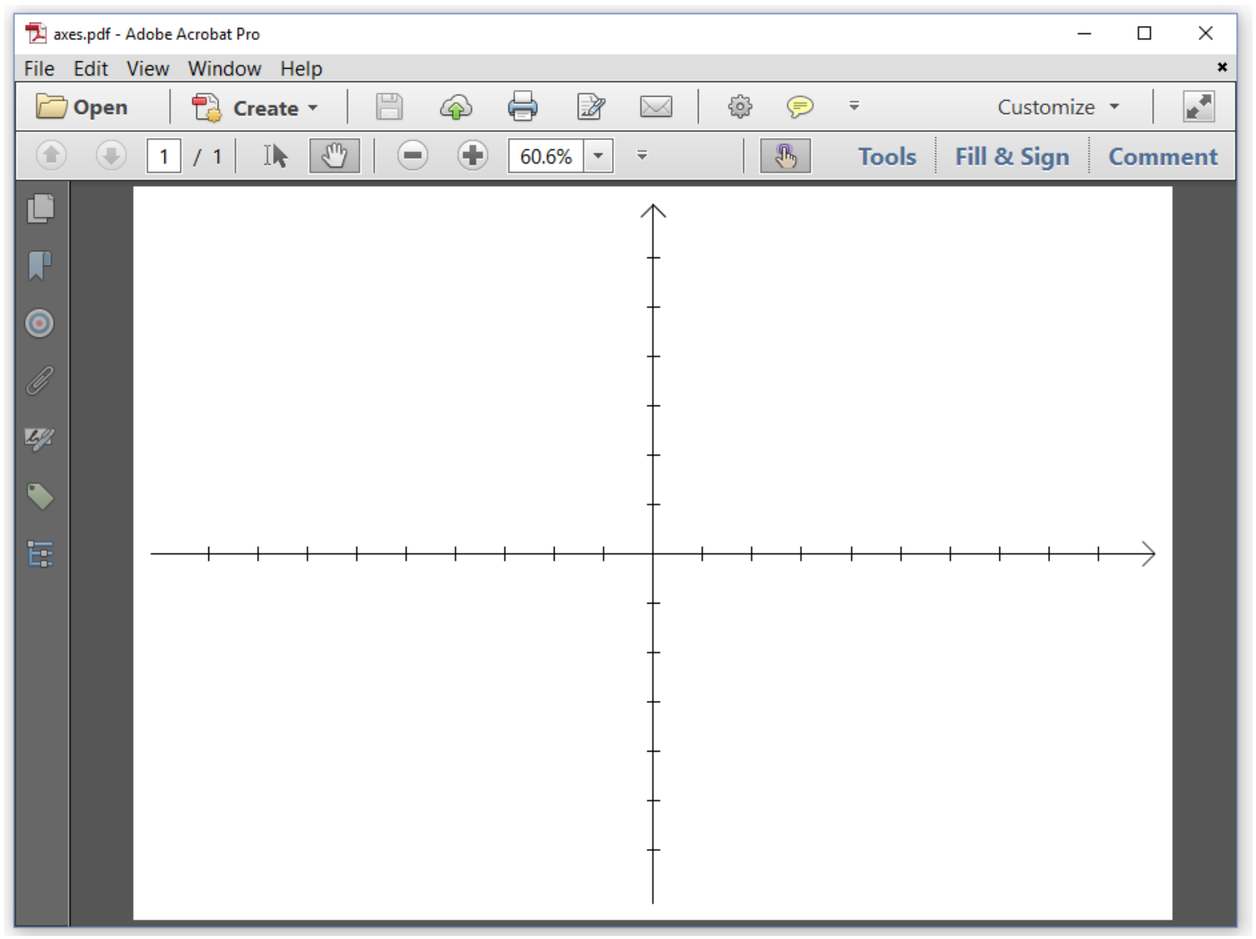


Figure 2.1: drawing an X and Y axis

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:

```java
//Draw X axis
canvas.moveTo(-(ps.getWidth() / 2 - 15), 0)
        .lineTo(ps.getWidth() / 2 - 15, 0)
        .stroke();
//Draw X axis arrow
canvas.setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.ROUND)
        .moveTo(ps.getWidth() / 2 - 25, -10)
        .lineTo(ps.getWidth() / 2 - 15, 0)
        .lineTo(ps.getWidth() / 2 - 25, 10).stroke()
        .setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.MITER);
//Draw Y axis
canvas.moveTo(0, -(ps.getHeight() / 2 - 15))
        .lineTo(0, ps.getHeight() / 2 - 15)
        .stroke();
//Draw Y axis arrow
canvas.saveState()
        .setLineJoinStyle(PdfCanvasConstants.LineJoinStyle.ROUND)
        .moveTo(-10, ps.getHeight() / 2 - 25)
        .lineTo(0, ps.getHeight() / 2 - 15)
        .lineTo(10, ps.getHeight() / 2 - 25).stroke()
        .restoreState();
//Draw X serif
for (int i = -((int) ps.getWidth() / 2 - 61);
    i < ((int) ps.getWidth() / 2 - 60); i += 40) {
    canvas.moveTo(i, 5).lineTo(i, -5);
}
//Draw Y serif
for (int j = -((int) ps.getHeight() / 2 - 57);
    j < ((int) ps.getHeight() / 2 - 56); j += 40) {
    canvas.moveTo(5, j).lineTo(-5, j);
}
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.

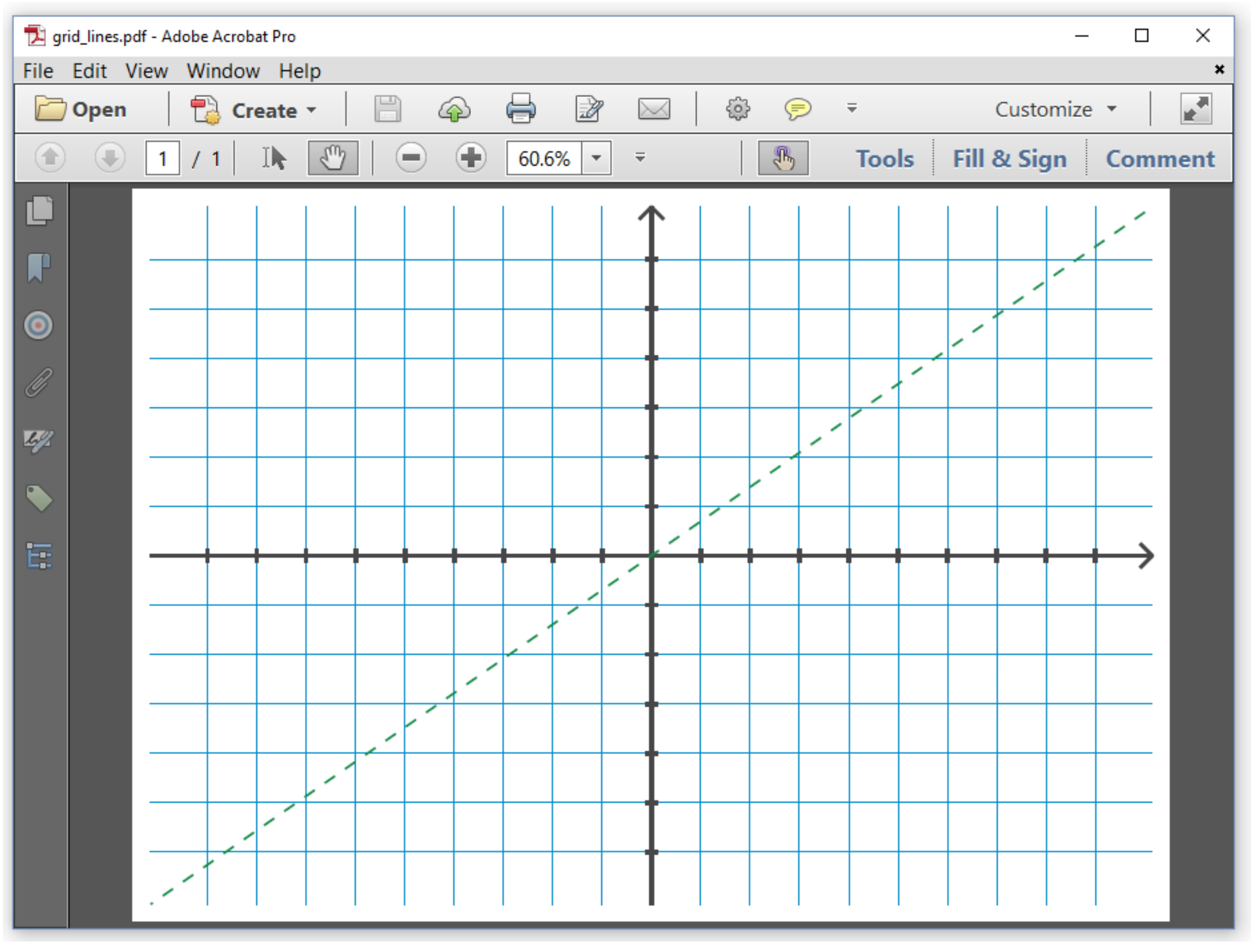

grid_lines.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
60.6%
Tools
Fill & Sign
Comment

Figure 2.2: drawing a grid

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

```java
Color grayColor = new DeviceCmyk(0.f, 0.f, 0.f, 0.875f);
Color greenColor = new DeviceCmyk(1.f, 0.f, 1.f, 0.176f);
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:

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

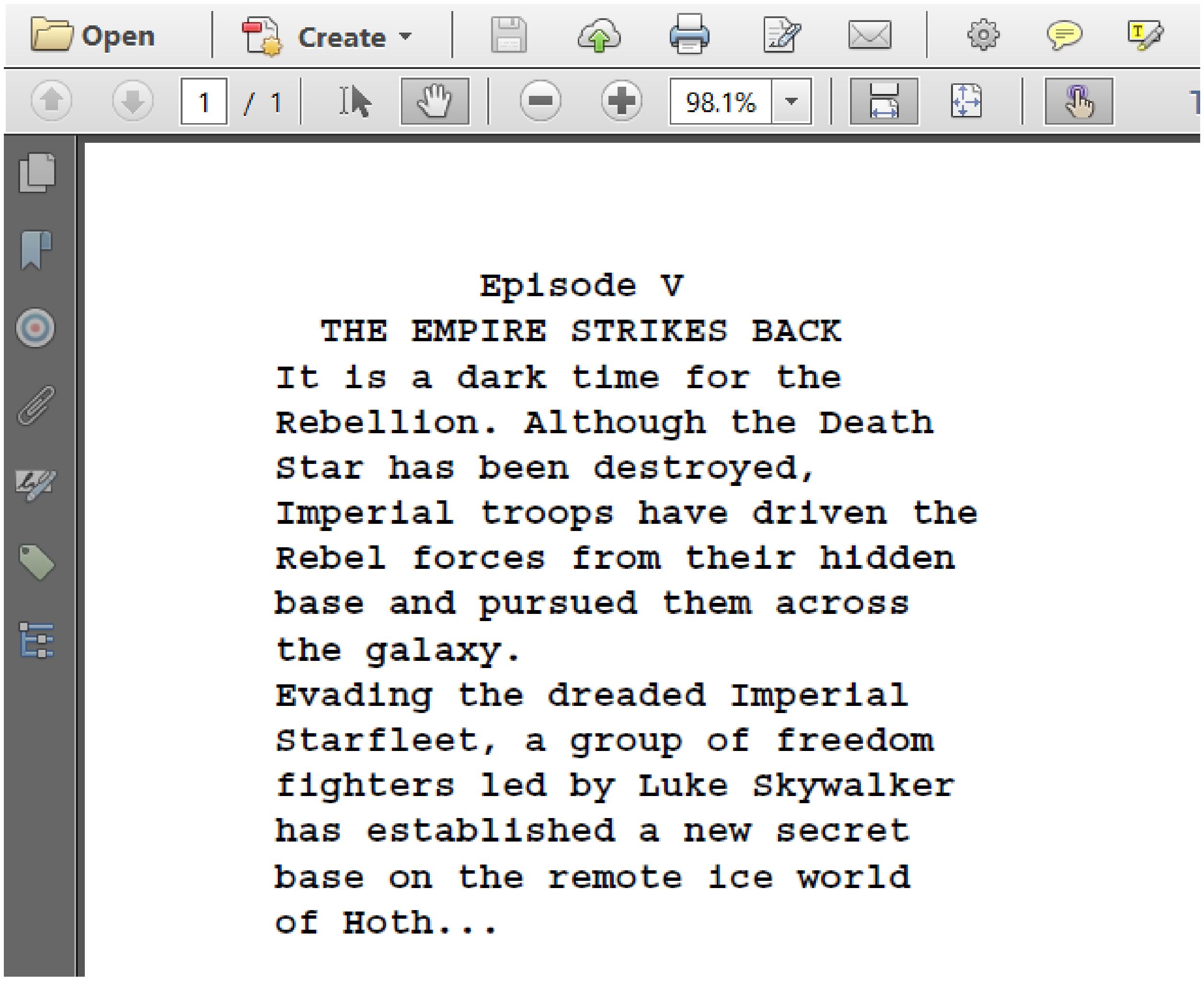
There are many possible variations to define a line dash, but in this case, we are defining the line dash using three parameters. The length of the dash is 10 user units; the length of the gap is 10 user units; the phase is 8 user units —the phase defines the distance in the dash pattern to start the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



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

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

```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:

```java
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);
```

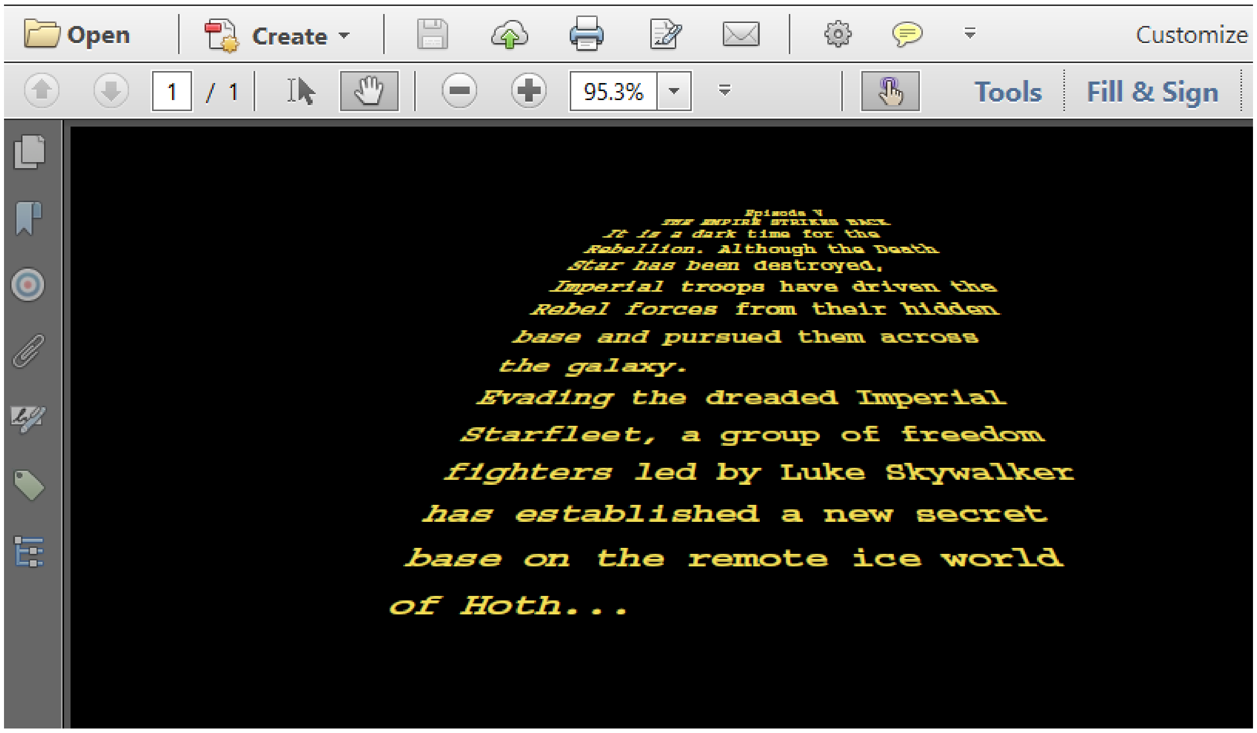
We create a `PdfFont` to show the text in Courier Bold and we change the text state so that all text that is drawn will use this font with font size 14. We also define a leading of 1.2 times this font size. The leading is the distance between the baselines of two subsequent lines of text. Finally, we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

```java
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?



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

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.