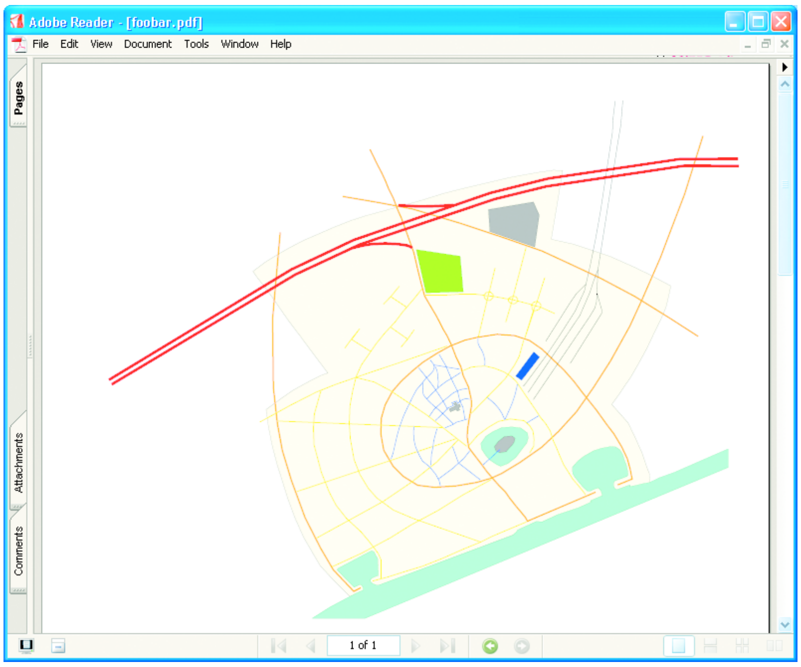

Adobe Reader - [foobar.pdf]
File Edit View Document Tools Window Help
Pages
Attachments
Comments
1 of 1

Figure 1.6 Using iText to draw graphics such as lines and shapes

Drawing a city map

Laura has made a map of the city of Foobar in the Scalable Vector Graphics (SVG) format, and throughout this book we'll attempt to create a PDF document based on this SVG file. First we'll deal with the streets (paths) and the squares (shapes), as shown in figure 1.6.

In chapter 10, the first chapter on PDF's *graphics state,* you'll learn about path construction and path-painting operators and operands. A first attempt to generate the map of Foobar appears in section 10.5.

Adding street names to the map

We'll continue discussing the graphics state in chapter 11, where you'll learn that PDF's *text state* is a subset of the graphics state. The text state will help us add the street names to the map. Figure 1.7 shows the result of a second attempt to draw the map of Foobar (see section 11.6).

The third attempt at drawing the map will use Apache Batik to parse the SVG.

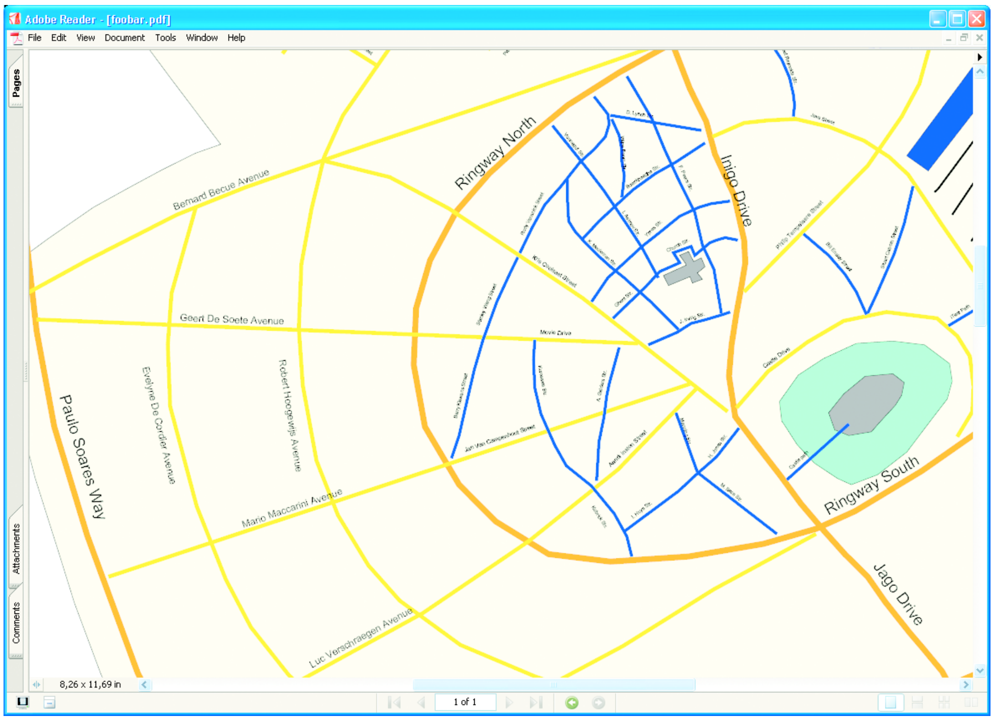


Figure 1.7 Using iText to draw text at absolute positions

Adding interactive layers to the map

Apache Batik is a library that can parse an SVG file and draw the paths, shapes, and text that are described in the form of XML to a `java.awt.Graphics2D` object. Chapters 10 and 11 present custom iText methods that are closely related to the operators and operands listed in the PDF Reference, and chapter 12 explains that you can also use an API you probably know already: the `java.awt` package.

For our first two attempts, we used one SVG file with the graphics and one with the street names in English, but Laura also wants to add the street names in French and Dutch. This task can be achieved using PDF's *optional content* feature, discussed in chapter 12. By adding each set of street names to a different *optional content group*, Laura can give foreign students the option to look at the map in the language of their choice, as shown in figure 1.8.

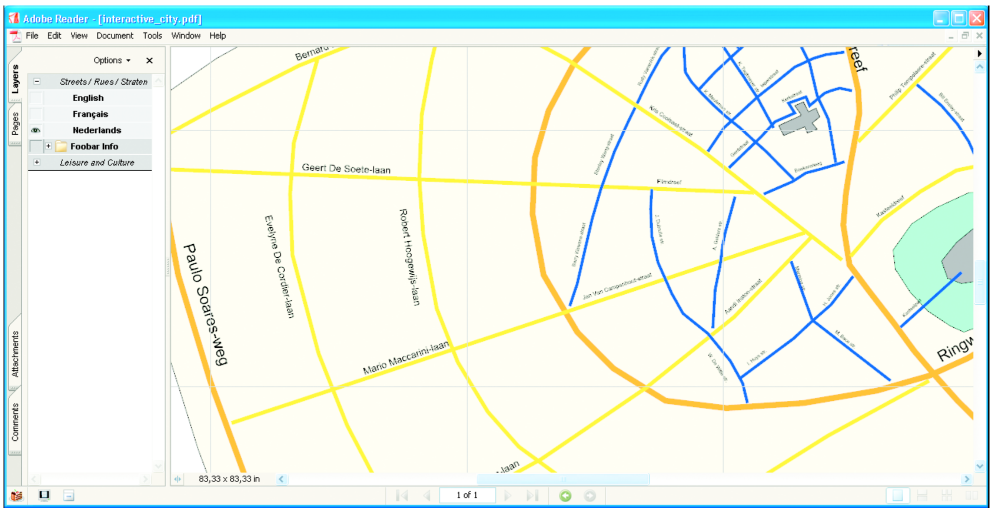


Figure 1.8 A PDF document demonstrating the use of optional content groups.

In section 12.4, we'll create a final version of the map of Foobar. Using Apache Batik, we'll parse different SVG files into different layers that can be turned on and off interactively.

This brings us to part 4, "Interactive PDF."

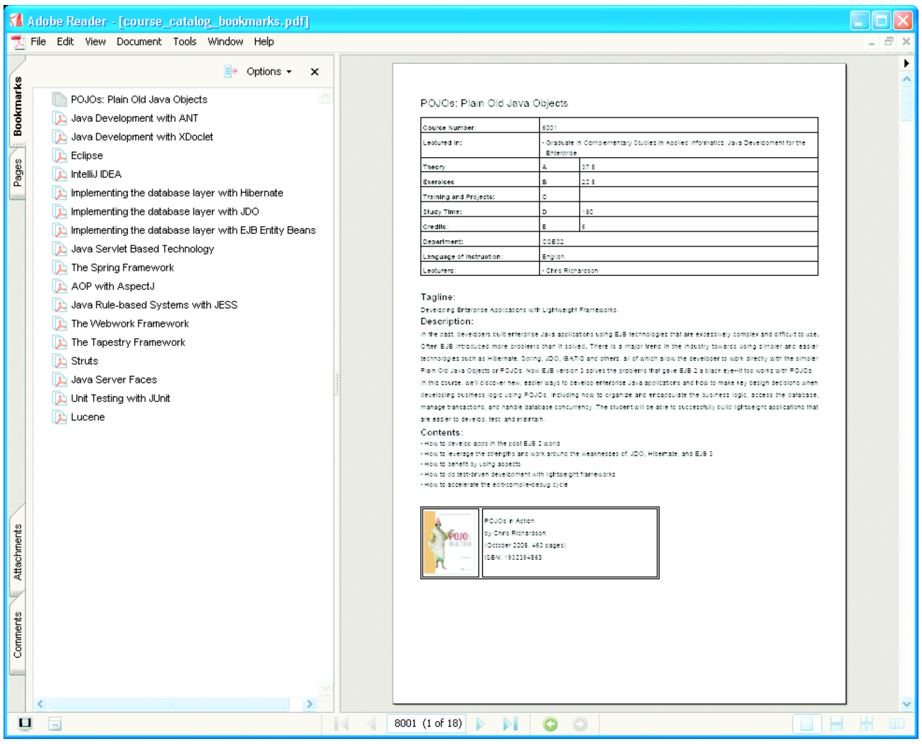
1.3.4 Producing and processing interactive documents

Laura can be hard on herself sometimes. She isn't quite satisfied with the study guide and course catalog shown in figures 1.3 and 1.4. She wants to add interactivity and extra features such as a watermark and page numbers.

Making documents interactive

Because a student's curriculum can consist of many different courses, it may be necessary to help students navigate through the course catalog. Let's add some extra links, annotations, and bookmarks to the document.

Chapter 4 discusses some building blocks with interactive features, but if you want the full assortment, you should dig into chapter 13, where you'll learn about setting viewer preferences; page labels and bookmarks; and actions and destinations. In section 13.6, we'll come back to the course catalog example and adapt it, giving it the interactive features shown in figure 1.9.



- POJOs: Plain Old Java Objects
- Java Development with ANT
- Java Development with XDoclet
- Eclipse
- IntelliJ IDEA
- Implementing the database layer with Hibernate
- Implementing the database layer with JDO
- Implementing the database layer with EJB Entity Beans
- Java Servlet Based Technology
- The Spring Framework
- AOP with AspectJ
- Java Rule-based Systems with JESS
- The Webwork Framework
- The Tapestry Framework
- Struts
- Java Server Faces
- Unit Testing with JUnit
- Lucene

POJOs: Plain Old Java Objects

Course Number:	8001	
Lectured in:	- Graduate in Complementary Studies in Applied Informatics: Java Development for the Enterprise	
Theory	A	37.5
Exercises	B	22.5
Training and Projects:	C	
Study Time:	D	180
Credits:	E	6
Department:	CSE02	
Language of Instruction:	English	
Lecturers:	- Chris Richardson	

Tagline:

Developing Enterprise Applications with Lightweight Frameworks.

Description:

In the past, developers built enterprise Java applications using EJB technologies that are excessively complex and difficult to use. Often EJB introduced more problems than it solved. There is a major trend in the industry towards using simpler and easier technologies such as Hibernate, Spring, JDO, IBATIS and others, all of which allow the developer to work directly with the simpler Plain Old Java Objects or POJOs. Now EJB version 3 solves the problems that gave EJB 2 a black eye--it too works with POJOs. In this course, we'll discover new, easier ways to develop enterprise Java applications and how to make key design decisions when developing business logic using POJOs, including how to organize and encapsulate the business logic, access the database, manage transactions, and handle database concurrency. The student will be able to successfully build lightweight applications that are easier to develop, test, and maintain.

Contents:

- How to develop apps in the post EJB 2 world
- How to leverage the strengths and work around the weaknesses of: JDO, Hibernate, and EJB 3
- How to benefit by using aspects
- How to do test-driven development with lightweight frameworks
- How to accelerate the edit-compile-debug cycle

POJOs in Action
by Chris Richardson
(October 2005, 450 pages)
ISBN: 1932394583

Figure 1.9 A PDF document demonstrating some interactive features.

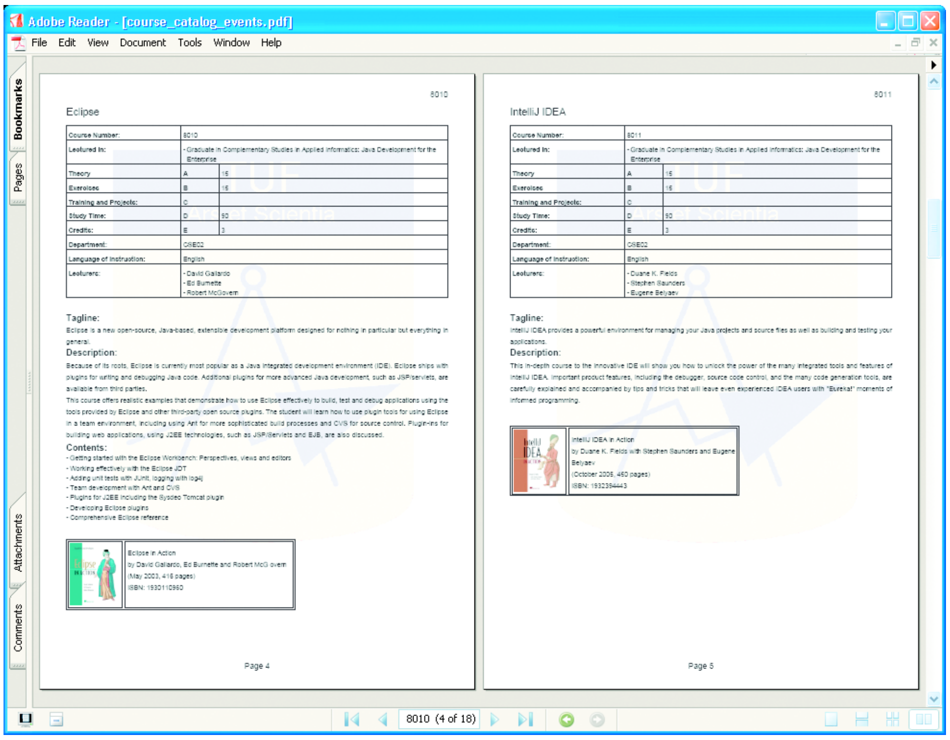
Adding watermarks and page numbers

Figure 1.10 shows pages 4 and 5 of the course catalog. The course number has been added as a header, and every file has the university's logo as its watermark.

In chapter 14, "Automating PDF Creation," you'll learn about page events that let you add content (such as watermarks or page numbers) automatically every time a new page is triggered.

Using iText in a web application

You may have wondered what the letter *i* in iText stands for. You'll find out while reading about *interactive* PDF. You already know that iText was initially designed to generate PDF in a web application and that its original purpose was to serve text interactively based on a user specific query. It's easy to adapt the code of the examples so that they can be integrated in a web application, as long as you know how to avoid some specific browser-related issues.


Adobe Reader - [course_catalog_events.pdf]
File Edit View Document Tools Window Help
Bookmarks
Pages
Attachments
Comments
TUF
Ars et Scientia
8010 (4 of 18)

8010

Eclipse

Course Number:	8010	
Lectured In:	- Graduate in Complementary Studies in Applied Informatics: Java Development for the Enterprise	
Theory	A	15
Exercises	B	15
Training and Projects:	C	
Study Time:	D	90
Credits:	E	3
Department:	CSE02	
Language of Instruction:	English	
Lecturers:	- David Gallardo - Ed Burnette - Robert McGovern	

Tagline:
Eclipse is a new open-source, Java-based, extensible development platform designed for nothing in particular but everything in general.

Description:
Because of its roots, Eclipse is currently most popular as a Java integrated development environment (IDE). Eclipse ships with plugins for writing and debugging Java code. Additional plugins for more advanced Java development, such as JSP/servlets, are available from third parties.
This course offers realistic examples that demonstrate how to use Eclipse effectively to build, test and debug applications using the tools provided by Eclipse and other third-party open source plugins. The student will learn how to use plugin tools for using Eclipse in a team environment, including using Ant for more sophisticated build processes and CVS for source control. Plugin-ins for building web applications, using J2EE technologies, such as JSP/Servlets and EJB, are also discussed.

Contents:
- Getting started with the Eclipse Workbench: Perspectives, views and editors
- Working effectively with the Eclipse JDT
- Adding unit tests with JUnit, logging with log4j
- Team development with Ant and CVS
- Plugins for J2EE including the Sysdeo Tomcat plugin
- Developing Eclipse plugins
- Comprehensive Eclipse reference

Eclipse in Action
by David Gallardo, Ed Burnette and Robert McG overn
(May 2003, 416 pages)
ISBN: 1930110960

Page 4

8011

IntelliJ IDEA

Course Number:	8011	
Lectured In:	- Graduate in Complementary Studies in Applied Informatics: Java Development for the Enterprise	
Theory	A	15
Exercises	B	15
Training and Projects:	C	
Study Time:	D	90
Credits:	E	3
Department:	CSE02	
Language of Instruction:	English	
Lecturers:	- Duane K. Fields - Stephen Saunders - Eugene Belyaev	

Tagline:
IntelliJ IDEA provides a powerful environment for managing your Java projects and source files as well as building and testing your applications.

Description:
This in-depth course to the innovative IDE will show you how to unlock the power of the many integrated tools and features of IntelliJ IDEA. Important product features, including the debugger, source code control, and the many code generation tools, are carefully explained and accompanied by tips and tricks that will leave even experienced IDEA users with "Eureka!" moments of informed programming.

IntelliJ IDEA in Action
by Duane K. Fields with Stephen Saunders and Eugene Belyaev
(October 2005, 450 pages)
ISBN: 1932394443

Page 5

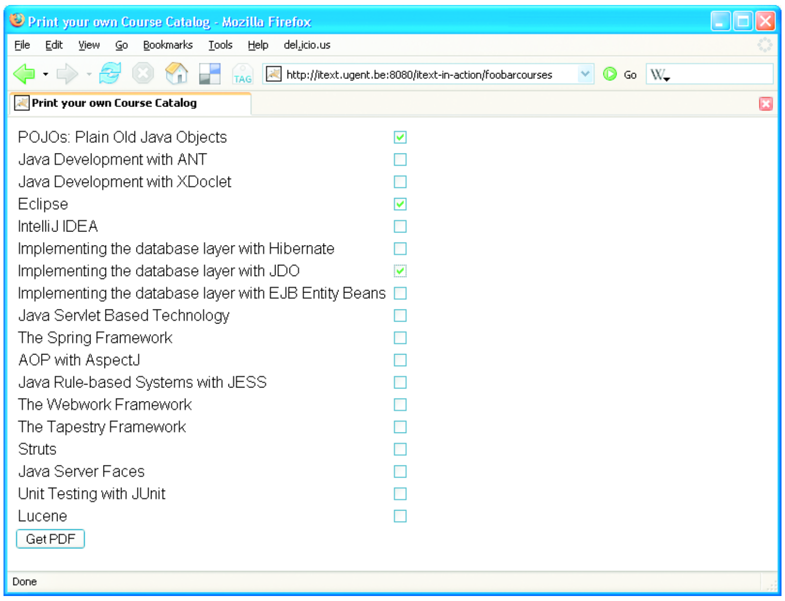
Figure 1.10 Using page events to add page numbers and watermarks

You can write a web application that is able to create a personalized course catalog for every student. Figure 1.11 shows a simple HTML form with the different courses that are in the catalog. This form was created dynamically based on the bookmarks inside the course catalog PDF.

Students can select the courses that interest them and create a personalized version of the course catalog. Figure 1.12 shows a PDF file containing information about the three courses that were selected in the HTML form shown in figure 1.11. Note that this screenshot also demonstrates the use of the Pages panel.

Chapter 17 lists the common pitfalls you should avoid when integrating iText in a web application. The source code used to produce the web pages shown in figures 1.11 and 1.12 can be found in section 17.2.

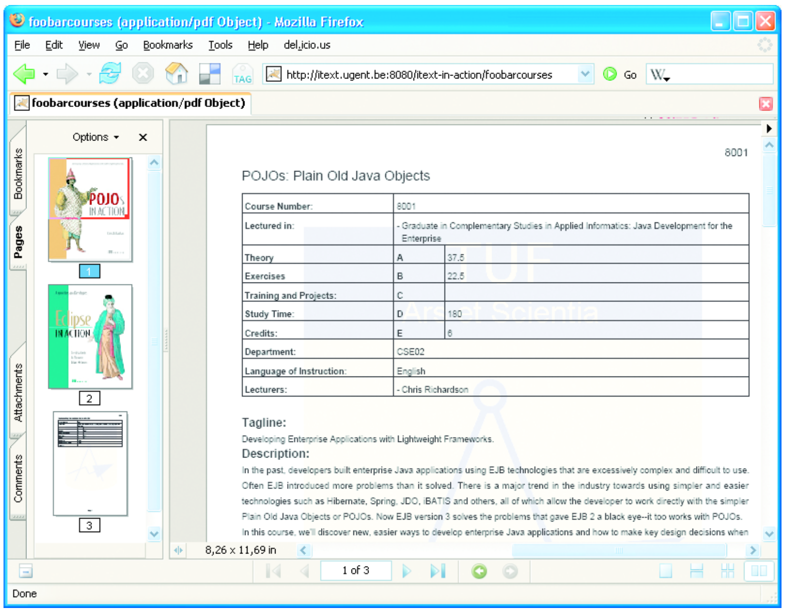
Notice that we've skipped chapters 15 and 16. These two chapters introduce the theory for another example that begins in section 17.2 and is completed in section 18.4.


Print your own Course Catalog - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://itext.ugent.be:8080/itext-in-action/foobarcourses
Go
Print your own Course Catalog
Done

POJOs: Plain Old Java Objects	✓
Java Development with ANT	☐
Java Development with XDoclet	☐
Eclipse	✓
IntelliJ IDEA	☐
Implementing the database layer with Hibernate	☐
Implementing the database layer with JDO	✓
Implementing the database layer with EJB Entity Beans	☐
Java Servlet Based Technology	☐
The Spring Framework	☐
AOP with AspectJ	☐
Java Rule-based Systems with JESS	☐
The Webwork Framework	☐
The Tapestry Framework	☐
Struts	☐
Java Server Faces	☐
Unit Testing with JUnit	☐
Lucene	☐

Get PDF

Figure 1.11 An HTML form listing the different courses in the course catalog


foobarcourses (application/pdf Object) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://itext.ugent.be:8080/itext-in-action/foobarcourses
Go
foobarcourses (application/pdf Object)
Options
Bookmarks
Pages
Attachments
Comments
1
2
3
8,26 x 11,69 in
1 of 3
Done

8001

POJOs: Plain Old Java Objects

Course Number:	8001	
Lectured in:	- Graduate in Complementary Studies in Applied Informatics: Java Development for the Enterprise	
Theory	A	37.5
Exercises	B	22.5
Training and Projects:	C	
Study Time:	D	180
Credits:	E	6
Department:	CSE02	
Language of Instruction:	English	
Lecturers:	- Chris Richardson	

Tagline:
Developing Enterprise Applications with Lightweight Frameworks.

Description:
In the past, developers built enterprise Java applications using EJB technologies that are excessively complex and difficult to use. Often EJB introduced more problems than it solved. There is a major trend in the industry towards using simpler and easier technologies such as Hibernate, Spring, JDO, iBATIS and others, all of which allow the developer to work directly with the simpler Plain Old Java Objects or POJOs. Now EJB version 3 solves the problems that gave EJB 2 a black eye--it too works with POJOs. In this course, we'll discover new, easier ways to develop enterprise Java applications and how to make key design decisions when

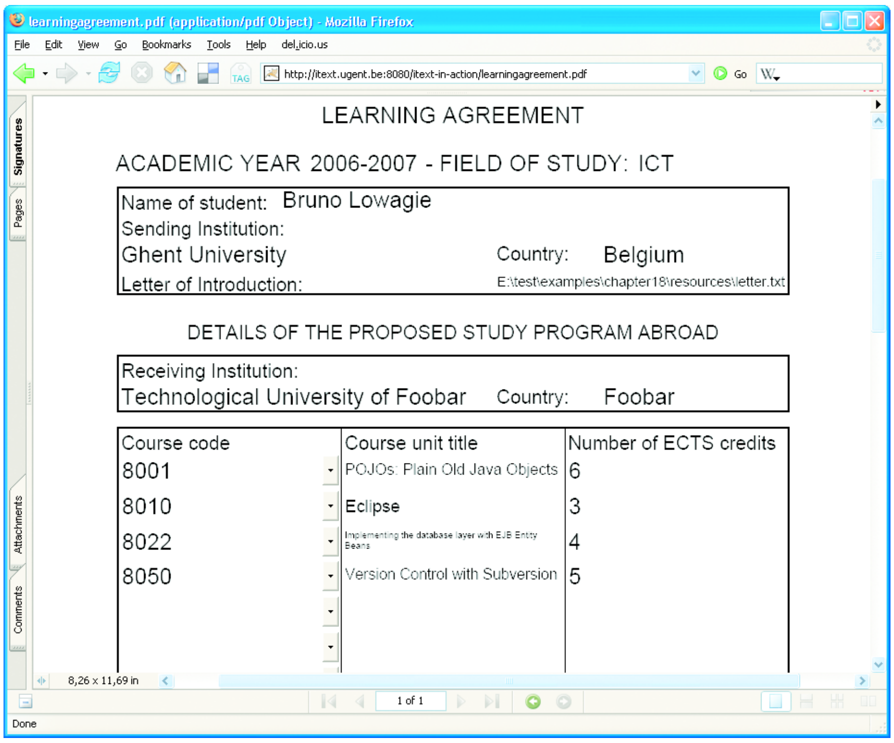
Figure 1.12 A PDF served by a web application containing a personalized course catalog

Creating and filling forms using iText

Exchange students who want to study at the TUF have to fill out a Learning Agreement form, and Laura wants to make this form available online. Students can print this form, fill it out manually, and send it to the university, but it would be nice if they also had the option to submit it online. That way, the courses they've chosen can be preregistered in the database, and when the student arrives on campus, the document can be checked and signed (manually or with a digital signature).

Figure 1.13 shows a PDF document with fillable form fields (the technical term is *AcroFields* in an *AcroForm*); the document is opened in the Adobe Reader browser plug-in. It can be submitted to a server.

Chapter 15 explains how you can create such a form using iText, and chapter 16 explains how you can fill in the form fields programmatically. We'll also *flatten the form* to create a registration card for the students, and you'll learn how to add a digital signature to a PDF file.


learningagreement.pdf (application/pdf Object) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://itext.ugent.be:8080/itext-in-action/learningagreement.pdf
Go
Signatures
Pages
Attachments
Comments
8,26 x 11,69 in
1 of 1
Done

LEARNING AGREEMENT

ACADEMIC YEAR 2006-2007 - FIELD OF STUDY: ICT

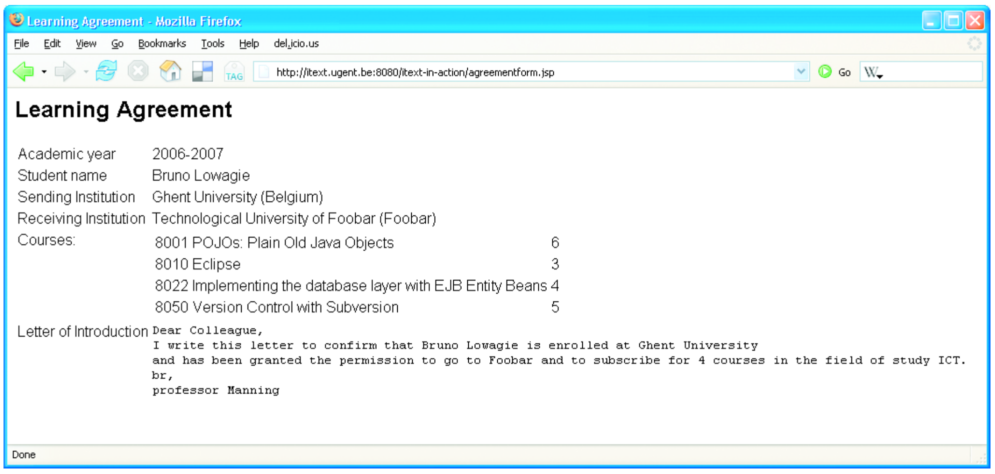
Name of student: Bruno Lowagie
Sending Institution:
Ghent University Country: Belgium
Letter of Introduction: E:\test\examples\chapter18\resources\letter.txt

DETAILS OF THE PROPOSED STUDY PROGRAM ABROAD

Receiving Institution:
Technological University of Foobar Country: Foobar

Course code	Course unit title	Number of ECTS credits
8001	POJOs: Plain Old Java Objects	6
8010	Eclipse	3
8022	Implementing the database layer with EJB Entity Beans	4
8050	Version Control with Subversion	5

Figure 1.13 A PDF form in a browser


Learning Agreement - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://itext.ugent.be:8080/itext-in-action/agreementform.jsp
Go
Done

Learning Agreement

Academic year	2006-2007	
Student name	Bruno Lowagie	
Sending Institution	Ghent University (Belgium)	
Receiving Institution	Technological University of Foobar (Foobar)	
Courses:	8001 POJOs: Plain Old Java Objects	6
	8010 Eclipse	3
	8022 Implementing the database layer with EJB Entity Beans	4
	8050 Version Control with Subversion	5

Letter of Introduction
```
Dear Colleague,
I write this letter to confirm that Bruno Lowagie is enrolled at Ghent University
and has been granted the permission to go to Foobar and to subscribe for 4 courses in the field of study ICT.
br,
professor Manning
```

Figure 1.14 Displaying the data that was submitted using a PDF AcroForm

In figure 1.14, a Java Server Pages (JSP) page displays the data that was sent to the server after submitting the form shown in figure 1.13.

Chapter 16 explains the different means that are available to retrieve the text values of the parameters that were submitted in the form of an (X)FDF file, but you'll need to read chapter 18 to understand how to extract the letter of introduction that was submitted as a file attachment.

1.3.5 Making the dream come true

Suddenly there is applause in the conference room. Laura abruptly wakes from her daydream to find everyone looking at her. The chairman of the committee nods at Laura in a consenting way, and says, "Well, Laura, those are some good ideas you've been sharing with us. Why not make a project out of them?"

Only then Laura does realize she hasn't been as quiet as she had intended. She has been speaking out loud, sharing her dreams and ideas with the complete committee, which is now, to her surprise, applauding her. For a moment she panics, but soon she calms down. Why wouldn't it be possible to make this dream come true?

I hope you'll understand that any resemblance to a real university or real persons, living or dead, is purely coincidental. There is no city of Foobar. Nor does this fictitious city have a Technological University. And there most certainly isn't any rivalry between the different fictitious departments; I made that up to add some spice to the story. And yet, if you've read the preface, you know where the inspiration to write this story came from. Stories like this happen to developers all the time; iText was born from a situation that was similar to the one Laura is facing now. This story could happen to you too. If it does, you don't have to worry about document problems anymore—this book can solve most of them for you.

1.4 Summary

The iText API was conceived for a specific reason: It allows developers to produce PDF files on the fly. The short history on the origin of the library made it clear that iText can easily be built into a web application to serve PDF documents to a browser dynamically.

We talked about the different ports of iText, but we chose to write all the book samples in Java, using the original iText. We compiled and executed a first example as a simple standalone application, and we also opened the iText toolbox. The toolbox was written to demonstrate some of the iText functionality from a simple GUI; you don't need to write any source code to use it.

The final section of this chapter offered you an à la carte view of what is possible with iText. Every figure in this section corresponds with a milestone in the iText learning process. If you plan on reading this book sequentially, you can use the corresponding sections as exercises to get acquainted with the functionality you've acquired earlier in the chapter.

If you intend to read this book to help you with a specific assignment, and your Chief Technology Officer (CTO) or your customer demands a proof of concept before you're allowed to start coding, just follow the pointers accompanying each screenshot in this section. You'll notice that most of the Foobar examples are XML based. You can feed these ready-made solutions with an XML file adapted to another working environment or another line of business—for instance, replacing students with customers and courses with products. After only a few hours of work, you should be able to convince your CTO or customer that iText may be the answer to their prayers.

I can't guarantee you won't have to do any extra programming to integrate the examples into your final application—but hey, wouldn't we all be out of work if the contrary were true?

2 PDF engine jump-start

This chapter covers

- Hello World, Hello iText
- Creating a PDF document in five steps
- Manipulating PDF: the basics

If you're new to iText, reading this chapter will be like your first day on a new job. Somebody gives you a quick tour of the building and makes you shake hands with people you don't know, and all the while you're hoping you'll be able to remember all of their names. At the end of the day, you may have the feeling you haven't done anything substantial, but really, you've done something important: You've said "hello" to everyone.

In this chapter, you'll create new PDF documents in five easy steps, and you'll learn several ways to implement one of those steps: adding content. You'll also learn how to read and manipulate existing PDF files using several iText classes.

Whereas the previous chapter gave you an overview of parts 2, 3, and 4 using screenshots of some real-world PDF documents, this chapter presents the most important mechanisms in iText. These mechanisms will return in almost every example.

2.1 Generating a PDF document in five steps

Following the principle that you shouldn't try to run before you can walk, we'll start with a simple PDF file. Figure 2.1 shows you a one-page PDF document saying nothing more than "Hello World".

The code that was used to generate this "Hello World" PDF is shown in listing 2.1. Note that the numbers to the side indicate the different steps.

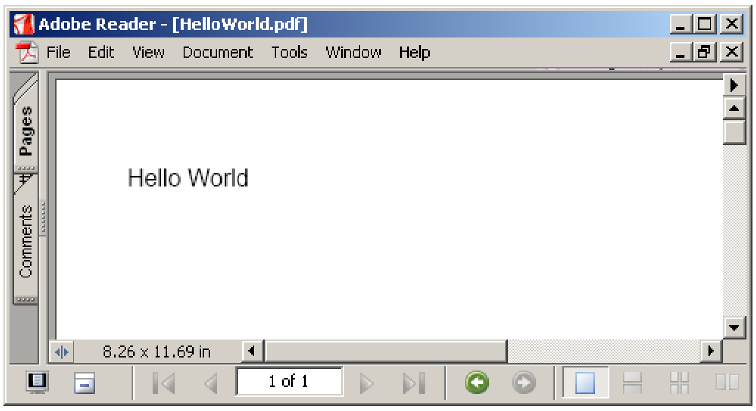

Adobe Reader - [HelloWorld.pdf]
File Edit View Document Tools Window Help
Pages
Comments
Hello World
8.26 x 11.69 in
1 of 1

Figure 2.1 Output of most of the "Hello World "examples in this chapter

Listing 2.1 Creating a HelloWorld.pdf in five steps

```
/* chapter02/HelloWorld.java */
Document document = new Document();   ❶
try {
  PdfWriter.getInstance(document,
    new FileOutputStream("HelloWorld.pdf"));   ❷
  document.open();   ❸
  document.add(
    new Paragraph("Hello World"));   ❹
} catch (Exception e) {
  // handle exception
}
document.close();   ❺
```

We'll devote a separate subsection to each of these five steps:

Step ❶ Create a `Document`.
Step ❷ Get a `DocWriter` instance (in this case, a `PdfWriter` instance)
Step ❸ Open the `Document`.
Step ❹ Add content to the `Document`.
Step ❺ Close the `Document`.

In every subsection, we'll focus on one specific step. You'll apply small changes to step ❶ in the first subsection, to step ❷ in the second, and so forth. This way, you'll create several new documents that are slightly different from the one in figure 2.1. You can hold these variations on the original "Hello World" PDF against a strong light (literally or not) and discover the differences and/or similarities caused by the small source code changes. In the final subsection (corresponding with step ❺), we'll weigh the design pattern used for iText against the Model-View-Controller (MVC) pattern.

2.1.1 *Creating a new document object*

`Document` is the object to which you'll add *content*: the document data and metadata. Upon creating the `Document` object, you can define the page size, the page color, and the margins of the first page of your PDF document. In listing 2.1, step ❶, a `Document` object is created with default values.

You can use a `com.lowagie.text.Rectangle` object to create a document with a custom size. Replace step ❶ in listing 2.1 with this snippet:

```
/* chapter02/HelloWorldNarrow.java */
Rectangle pageSize = new Rectangle(216f, 720f);
Document document = new Document(pageSize);
```

The two `float` values passed to the `Rectangle` constructor are the width and the height of the page. These values represent *user units*. By default, a user unit corresponds with the typographic unit of measurement known as the *point.* There are 72 points in one inch. You've defined a width of 216 pt (3 in) and a height of 720 pt (10 in). If you open the resulting PDF in Adobe Reader and look at the tab File > Document Properties > Description, you can check whether the document indeed measures 3 x 10 in.

Page size

Theoretically, you could create pages of any size, but the PDF specification[1] imposes limits depending on the PDF version of the document that contains those pages. For PDF 1.3 or earlier, the minimum page size is 72 x 72 units (1 x 1 in); the maximum is 3,240 x 3,240 units (45 x 45 in). Later versions have a minimum size of 3 x 3 units (approximately 0.04 x 0.04 in) and a maximum of 14,400 x 14,400 units (200 x 200 in).

We'll discuss some other, more general version limitations in chapter 3.

FAQ *Are there methods in iText to convert points into inches, inches into meters, and so forth?* No. You'll notice that all measurements are done in points and occasionally in thousandths of points (see chapter 9). The conversion from and to the metric system and other systems of measurement has to be handled in your code. Remember that 1 in = 2.54 cm = 72 points.

In most cases, you'll probably prefer using a standard paper size. If you want to write a letter to the world using the standard letter format, you have to change step ❶ like this:

```
/* chapter02/HelloWorldLetter.java */
Document document = new Document(PageSize.LETTER);
```

This creates a PDF document sized at 8.5 x 11 in, whereas the first "Hello World" example was created with the default page size DIN A4 (8.26 x 11.69 in or 210 x 297 mm).

[1] Adobe Systems Inc., *PDF Reference*, fifth edition, Appendix H, section 3, "Implementation notes," http://partners.adobe.com/public/developer/pdf/index_reference.html.

NOTE A4 is the most common paper size in Europe, Asia, and Latin America. It's specified by the International Standards Organization (ISO). ISO paper sizes are based on the metric system. The height divided by the width of all these formats is the square root of 2 (1.4142).

`PageSize` is a class written for your convenience. It contains nothing but a list of `static final Rectangle` objects, offering a selection of standard paper sizes: `A0` to `A10`, `B0` to `B5`, `LEGAL`, `LETTER`, `HALFLETTER`, `_11x17`, `LEDGER`, `NOTE`, `ARCH_A` to `ARCH_E`, `FLSA`, and `FLSE`. The orientation of most of these formats is Portrait. You can change this to Landscape by invoking the `rotate` method on the `Rectangle`. Step ❶ now looks like this:

```
/* chapter02/HelloWorldLandscape.java */
Document document = new Document(PageSize.LETTER.rotate());
```

Another way to create a `Document` in Landscape is to create a `Rectangle` object with a width that is greater than the height:

```
/* chapter02/HelloWorldLandscape2.java */
Document document = new Document(new Rectangle(792, 612));
```

The results of both Landscape examples look the same in Adobe Reader. The Reader's Description tab doesn't show any difference in size. Both PDF documents have a page size of 11 x 8.5 in (instead of 8.5 x 11 in), but there are subtle differences internally:

- In the first file, the page size is defined with a size that has a width lower than the height, but with a rotation of 90 degrees.
- The second file has the page size you defined without any rotation (a rotation of 0 degrees).

This difference will matter when you want to manipulate the PDF.

Page color

If you use a `Rectangle` as `pageSize` parameter, you can also change the background color of the page. In the next example, you change the background color to cornflower blue by setting the color of the `Rectangle` with `setBackgroundColor`:

```
/* chapter02/HelloWorldBlue.java */
Rectangle pagesize = new Rectangle(612, 792);
pagesize.setBackgroundColor(new Color(0x64, 0x95, 0xed));
Document document = new Document(pagesize);
```

The `Color` class used in this example is `java.awt.Color`; the colorspace is Red-Green-Blue (RGB) in this case. If you need another colorspace—for instance, Cyan-Magenta-Yellow-Black (CMYK)—you can use the class com.lowagie.text.-pdf.ExtendedColor. You can find a class diagram of the color classes in appendix A, section A.8; you'll read all about colors in chapter 11.

The iText API includes a third constructor of the Document class that we didn't discuss yet. This constructor not only takes a Rectangle as a parameter, but four float values as well.

Page margins

In step ❹ of the example, you add a Paragraph object to the document. This paragraph contains the words "Hello World," but how does iText know where to put those words on the page? The answer is simple: When adding basic building blocks such as Paragraph, Phrase, Chunk, and so forth to a document, iText keeps some space free at the left, right, top, and bottom. These are the margins of your document. All the "Hello World" examples you've created so far have default margins of half an inch (36 units in PDF). Let's change step ❶ one last time:

```
/* chapter02/HelloWorldMargins.java */
Document document = new Document(PageSize.A5, 36, 72, 108, 180);
```

The PDF document now has a left margin of 36 pt (0.5 in), a right margin of 72 pt (1 in), a top margin of 108 pt (1.5 in), and a bottom margin of 180 pt (2.5 in).

You can mirror the margins by adding a line of code after step ❷:

```
/* chapter02/HelloWorldMirroredMargins.java */
document.setMarginMirroring(true);
```

In this example, all the odd pages have a left margin of 36 pt and a right margin of 72 pt. For the even pages, it's the other way around.

2.1.2 Getting a DocWriter instance

Once you have a document instance, you need to decide if you'll write the document to a file, to memory, or to the output stream of a Java servlet. You also need to decide if you'll produce PDF or another format that is supported by iText.

Step ❷ combines these two actions:

- It tells the DocWriter to which OutputStream the resulting document should be written.
- It associates a Document with an implementation of the abstract DocWriter class. In this book, we focus on the class PdfWriter because we're interested in generating PDF. It can be useful to know that you can also get a DocWriter instance that produces RTF (using RtfWriter2) or HTML (using HtmlWriter).

These writers translate the content you're adding to the `Document` object into the syntax of some specific document format (PDF, RTF, or HTML).

The class diagram in appendix A, section A.1, shows how the different `DocWriter` classes relate to each other. In the upper-left corner, you'll recognize the `Document` object. One of the member values is an `ArrayList` of listeners. These listeners implement the `DocListener` interface. For instance, if you add an element to the document, the document forwards it to the `add` method of its listeners. The `DocListener` interface is implemented by different subclasses of the abstract class `DocWriter`.

As you can see in the class diagram, the constructors of these classes are protected. You can only create them using the `public static getInstance()` method. This method creates the writer and adds the newly created object as a listener to the document. If necessary, some helper classes are created for internal use by iText only; see, for instance, the `PdfDocument` or `RtfDocument` object.

Creating the same document in different formats

Let's add some extra lines to step ❷ and see what happens:

```
/* chapter02/HelloWorldMultiple.java */
PdfWriter.getInstance(document,
  new FileOutputStream("HelloWorldMultiple.pdf"));
RtfWriter2.getInstance(document,
  new FileOutputStream("HelloWorldMultiple.rtf"));
HtmlWriter.getInstance(document,
  new FileOutputStream("HelloWorldMultiple.htm"));
```

Because you're careful only to use code that is valid for all three presentation formats (PDF, RTF, and HTML), you're able to generate three different files (of different types) using the same code for steps ❶, ❸, ❹, and ❺. Note that this approach won't work with all the building blocks described in this book.

Choosing an OutputStream

While you're adding content to the document, the writer instance gradually writes PDF, RTF, or HTML syntax to the output stream. So far, you've written simple PDF, RTF, and HTML documents to a file using the `java.io.FileOutputStream`. Most examples in this book are written this way so you can try the examples on your own machine without having to install additional software such as a web server or a J2EE container.

In real-world applications, you may want to write a PDF byte stream to a browser (to a `ServletOutputStream`) or to memory (to a `ByteArrayOutputStream`).

All of this is possible with iText; you can write to any `java.io.OutputStream` you want. If you want to write a PDF document to the `System.out` to see what PDF looks like on the inside, you can change step ❷ like this:

```
/* chapter02/HelloWorldSystemOut.java */
PdfWriter.getInstance(document, System.out);
```

If you try this example, you won't recognize the words "Hello World" in the output; but you'll notice different structures: objects marked `obj`, dictionaries between << and >> brackets, and a lot of binary gibberish. In chapter 18, we'll look under the hood of iText and PDF, and you'll learn to distinguish the different parts that make up a PDF file. But this is stuff for people who really want to dig into the Portable Document Format; you're probably more interested in seeing how to serve a PDF file in a web application.

Class `javax.servlet.ServletOutputStream` extends `java.io.OutputStream`, so you could try getting an instance of `PdfWriter` with `response.getOutputStream()` as a second parameter. This works on some—but, unfortunately not all—browsers. Chapter 17 will tell you how to avoid the many pitfalls you're bound to encounter once you start integrating iText (or any other dynamic PDF-producing tool) in a J2EE web application. Notice that those problems are in most cases browser-related, *not* iText-related.

For now, let's look at something simpler: opening the document.

2.1.3 Opening the document

Java programmers may not be used to having to open streams before being able to add content. You create a new stream and write `bytes`, `chars`, and `Strings` to it right away.

With iText, it's mandatory to open the document first. When a document object is opened, a lot of initializations take place in iText. If you use the parameterless `Document` constructor and you want to change page size and margins with the corresponding setter methods, it's important to do this *before* opening the document. Otherwise the default page size and margins will be used for the first page, and your page settings will only be taken into account starting from the second page.

The following snippet opens a document in which the first page is letter size, landscape oriented, with a left margin of 0.5 in, a right margin of 1 in, a top margin of 1.5 in, and a bottom margin of 2 in:

```
/* chapter02/HelloWorldOpen.java */
Document document = new Document();
```

```
PdfWriter.getInstance(document, new
  FileOutputStream("HelloWorldOpen.pdf"));
document.setPageSize(PageSize.LETTER.rotate());
document.setMargins(36, 72, 108, 144);
document.open();
```

One of the most common questions iText users ask is why page settings apply to all pages but the first. The answer is almost always the same: You've added the desired behavior *after* opening the `Document` instead of *before*.

Many document types keep version information and metadata in the file header. That's why you should always set the PDF version and add the metadata before opening the document.

The PDF header

When `document.open()` is invoked, the iText `DocWriter` starts writing its first bytes to the `OutputStream`. In the case of `PdfWriter`, a PDF header is written, and by default it looks like this:

```
%PDF-1.4
%âãÏÓ
```

The first line shows the PDF version of the document; that's obvious. The second line may seem a little odd. It starts with a percent symbol, which means it's a PDF comment line; thus it doesn't seem to have any function. It isn't necessary to add this line, but doing so is recommended to ensure the "proper behavior of file transfer applications that inspect data near the beginning of a file to determine whether to treat the file's content as text or as binary."[2]

PDF documents are binary files. Some systems or applications may not preserve binary characters, and this almost inevitably makes the PDF file corrupt. According to the PDF Reference, this problem can be avoided by including at least four binary characters (codes greater than 127) in a comment near the beginning of the file to encourage "binary treatment."

For the time being, iText generates PDF files with version 1.4 by default. If you look at table 2.1, you'll notice that version 1.4 is rather old.

If you want to use functionality that is available only in a PDF version other than v1.4, you can change the default PDF version with the method `PdfWriter.-`

[2] See section 3.4.1 of the PDF Reference version 1.6.

Table 2.1 Overview of the PDF versions

PDF version	Year	iText constant
PDF-1.0	1993	-
PDF-1.1	1994	-
PDF-1.2	1996	PdfWriter.VERSION_1_2
PDF-1.3	1999	PdfWriter.VERSION_1_3
PDF-1.4	2001	PdfWriter.VERSION_1_4
PDF-1.5	2003	PdfWriter.VERSION_1_5
PDF-1.6	2004	PdfWriter.VERSION_1_6

`setPdfVersion()`, using one of the static values displayed in the third column of table 2.1:

```
/* chapter02/HelloWorldVersion_1_6.java */
Document document = new Document();
PdfWriter writer = PdfWriter.getInstance(document,
  new FileOutputStream("HelloWorld_1_6.pdf"));
writer.setPdfVersion(PdfWriter.VERSION_1_6);
document.open();
```

This file is intended to be viewed in Adobe Reader 7.0 or later. If you use an older version of Adobe Reader, you'll get a warning (Acrobat Reader 3.0 and later) or even an error (all versions before Acrobat Reader 3.0). The cause of this error will be explained in the next chapter.

FAQ *Why doesn't iText generate PDF in the latest PDF version by default?* The iText developers consider themselves to be early adopters of the newest versions in many ways, but with respect to the end users of their software, they deliberately didn't use the most recent version. An end user may still be using a viewer that only supports older PDF versions.

Changing the version number of the PDF has to be done before opening the document, because you can't change the header once it's written to the `OutputStream`.

The metadata of a PDF document is kept in an *info dictionary*. This dictionary is a PDF object that can be put anywhere in the PDF. In theory, it would be possible to add metadata after opening the document when producing PDF only, but in practice iText doesn't allow this. This was a design decision—an attempt to keep the code to produce HTML, RTF, and PDF as uniform as possible.

Adding metadata

Let's rewrite the HelloWorldMultiple example and change it into HelloWorld-Metadata:

```
/* chapter02/HelloWorldMetadata.java */
document.addTitle("Hello World example");
document.addSubject("This example shows how to add metadata");
document.addKeywords("Metadata, iText, step 3, tutorial");
document.addCreator("My program using iText");
document.addAuthor("Bruno Lowagie");
document.addHeader("Expires", "0");
document.open();
```

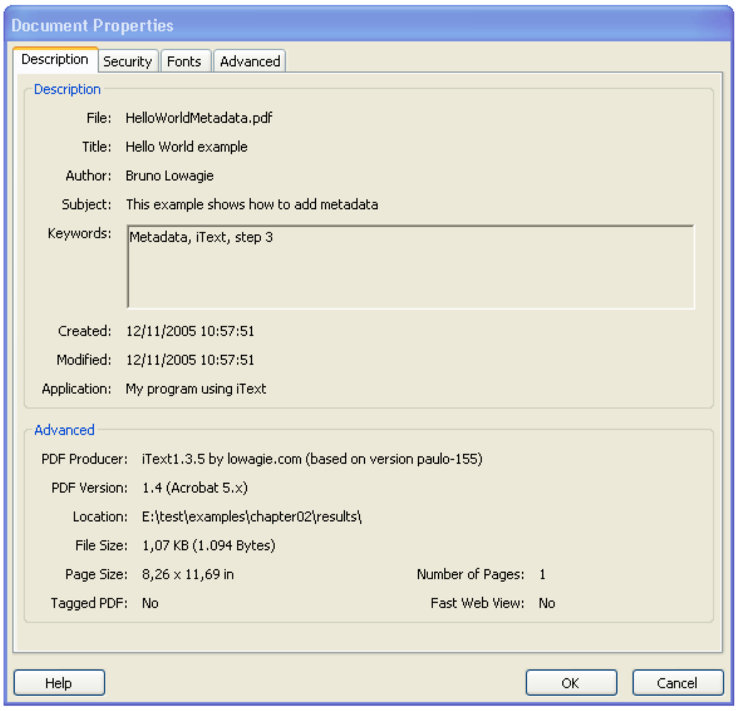
In HTML, all this information is stored in the `<head>` section of the resulting file:

```
<head>
  <title>
    Hello World example
  </title>
  <meta name="subject" content="This example shows how to add metadata" />
  <meta name="keywords" content="Metadata, iText, step 3" />
  <!-- Creator: My program using iText -->
  <meta name="author" content="Bruno Lowagie" />
  <meta name="Expires" content="0" />
  <!-- iText 1.4 (by lowagie.com) -->
  <!-- CreationDate: Wed Dec 28 09:44:40 CET 2006 -->
</head>
```

In PDF, the metadata passed to `addHeader` is added as a key-value pair to the PDF info dictionary. This example adds the `Expires` key. This has no meaning in the PDF syntax, so it won't have any effect on the PDF file. Figure 2.2 shows how the metadata added to the info dictionary is visualized in the File > Document Properties > Description dialog box.

Don't change the producer information and the creation date. If you ever need support from the mailing list, the producer information will tell which iText version you're using. In figure 2.2, you can immediately see that an old version of iText is being used (iText 1.3.5 dates from October 2005).

If you experience a problem with an iText-generated PDF file, you can use this version number to check whether the problem is caused by a bug that has been fixed in a more recent version.



Description

File: HelloWorldMetadata.pdf
Title: Hello World example
Author: Bruno Lowagie
Subject: This example shows how to add metadata
Keywords: Metadata, iText, step 3
Created: 12/11/2005 10:57:51
Modified: 12/11/2005 10:57:51
Application: My program using iText

Advanced

PDF Producer: iText1.3.5 by lowagie.com (based on version paulo-155)
PDF Version: 1.4 (Acrobat 5.x)
Location: E:\test\examples\chapter02\results\
File Size: 1,07 KB (1.094 Bytes)
Page Size: 8,26 x 11,69 in Number of Pages: 1
Tagged PDF: No Fast Web View: No

Figure 2.2 Document properties of HelloWorldMetadata.pdf.

FAQ *How do you retrieve the producer information programmatically?* The iText version, displayed as the producer information in the document properties, can also be retrieved programmatically with the static method `Document.getVersion()`. If you look into the iText source code, you'll see that this method and the corresponding `private static final String ITEXT_VERSION` may only be changed by Paulo Soares and Bruno Lowagie. The underlying philosophy of this restriction is purely a matter of courtesy. You can use iText for free, but in return you implicitly have to give the product some publicity. The iText developers hope you don't mind granting them this small favor. It's better than having a watermark saying "free trial version" spoiling every page of your document. Besides, the average end user never looks at the Advanced section of the Document Properties and thus is never confronted with this hidden persuader.

Now that you've added metadata and opened the document, you can start adding real data.

2.1.4 Adding content

This chapter explains the elementary mechanics of iText. Once these are understood, you can start building real-world applications with real-world content. You can copy and paste steps ❶, ❷, ❸, and ❺ from any Hello World example into your own applications; the principal part of your job will be implementing step ❹: adding content to the PDF document.

There are three ways to do this:

- *The easy way*—Using iText's basic building blocks
- *As a PDF expert*—Using iText methods that correspond with PDF operators and operands
- *As a Java expert*—Using Graphics2D methods and the paint method in Swing components

Listing 2.1 generated a "Hello World" PDF the easy way; now let's create the same PDF file using alternative techniques.

Using building blocks

In listing 2.1, you used a `Paragraph` object to add the words "Hello World" to the document. `Paragraph` is one of the many objects that will be discussed in part 2 of this book, "Basic building blocks." These building blocks will let you programmatically compose a document in a programmer-friendly way without having to worry about layout issues. Each of these building blocks has its own set of methods to parameterize properties such as the leading, indentation, fonts, colors, border widths, and so forth. iText does all the formatting based on these properties.

Note that iText is not a tool to *design* a document. It's not a word processor, nor is it a What You See Is What You Get (WYSIWYG) tool—otherwise I would have called it user-friendly instead of programmer-friendly. It's a library that lets you, the developer, produce PDF documents on the fly—for example, when you want to publish the content of a database in nice-looking reports. In part 2, we'll start with simple text elements and images, but the key chapters will be chapter 6, "Constructing tables," and chapter 7, "Constructing columns." Remember that if you use iText's basic building blocks, you don't need to know anything about PDF.

In some cases, this limited set of building blocks won't be sufficient for your needs, and you'll have to use one of the alternatives.

Low-level PDF generation

The content of every page in a PDF file is defined inside a *content stream*. In chapter 18, “Under the hood,” we’ll look inside a PDF document. You’ll learn that the content stream of a page is a PDF object of type `stream`. Listing 2.2 shows the uncompressed content stream of the “Hello World” page created with listing 2.1.

Listing 2.2 Content stream of the Hello World page

```
<</Length 55>>stream
q
BT
36 806 Td
0 -18 Td
/F1 12 Tf
(Hello World)Tj
ET
Q

endstream
```

You immediately recognize the words “Hello World”; after reading part 3, you’ll also understand the meaning of the other PDF operators and operands that are between the keywords `stream` and `endstream`. When you use basic building blocks, you add these operators and operands internally using an object called `PdfContentByte`.

iText allows you to grab this object so that you can address it directly—with the method `PdfWriter.getDirectContent()`, for example. Starting from the original listing 2.1, you could replace step ❹ with the following lines:

```
/* chapter02/HelloWorldAbsolute.java */
PdfContentByte cb = writer.getDirectContent();
BaseFont bf = BaseFont.createFont(
    BaseFont.HELVETICA, BaseFont.CP1252, BaseFont.NOT_EMBEDDED);
cb.saveState();             // q
cb.beginText();             // BT
cb.moveText(36, 806);       // 36 806 Td   ❶
cb.moveText(0, -18);        // 0 -18 Td   ❷
cb.setFontAndSize(bf, 12);  // /F1 12 Tf
cb.showText("Hello World"); // (Hello World)Tj   ❸
cb.endText();               // ET
cb.restoreState();          // Q
```

I have added the corresponding PDF operators and operands in a comment section after each line.

First you move the cursor to the starting position ❶. The default margin to the right was 36 units. Note that the lower-left corner of the page is used as the origin of the coordinate system by default. The height of the page (`PageSize.A4.height()`) is 842 units. You subtract the top margin: 842 – 36 = 806 units. That's the starting position: x = 36; y = 806.

Subsequently, you move down 18 units ❷. This is the line spacing. In the PDF Reference, as well as in iText, the line spacing is called the *leading*. You could reduce these two lines to one: `cb.moveText(36, 788);` that's the position where you add the "Hello World" paragraph using `showText` ❸. The other methods set the state, define a text block, and set the font and font size.

You can print the file that was generated using the first example (HelloWorld.pdf) and the file generated using this code snippet (HelloWorldAbsolute.pdf), hold them both to a strong light, and see that their output is identical. You may ask why one would go through the trouble of learning how to write PDF syntax when adding a simple line of code in current iText versions will do the work for you. But you have to take into account that this isn't really a representative example.

In real-world examples, you'll often write to the direct content using the `PdfContentByte` object—for example, to add page numbers or a page header or footer at an absolute position. This `PdfContentByte` object offers you a maximum of flexibility and PDF power, as long as you take into account the words of Spider-Man's Uncle Ben: "With great power, there comes great responsibility." If you use `PdfContentByte`, it's advised that you know something about PDF syntax.

Don't panic—it won't be necessary to read the complete PDF Reference. Chapters 10 and 11 of this book will explain everything you need to know. You'll learn about PDF's *graphics state* and *text state*, and we'll discuss the PDF coordinate system and most of the operators and operands that are available.

If you want to avoid this low-level PDF functionality, chapter 12 talks about a third way to add content to a page: using the Java Abstract Windowing Toolkit (AWT).

Using java.awt.Graphics2D

In the original Star Trek series, the character Leonard "Bones" McCoy is often heard to say things like "I'm a doctor, not a bricklayer!" You may now be having a similar reaction—"I'm a Java developer, not a PDF specialist. I want to use iText so that I can avoid learning PDF syntax!"

If that is the case, I have good news for you. The class `PdfContentByte` has a series of `createGraphics()` methods that let you create a subclass of the abstract Java class `java.awt.Graphics2D` called `com.lowagie.text.pdf.PdfGraphics2D`. This subclass overrides all the `Graphics2D` methods, translating them to `PdfContentByte` calls behind the scenes.

Once again, you replace step ❹ in listing 2.1:

```
/* chapter02/HelloWorldGraphics2D.java */
PdfContentByte cb = writer.getDirectContent();
Graphics2D graphics2D =
  cb.createGraphics(PageSize.A4.width(), PageSize.A4.height());
graphics2D.drawString("Hello World", 36, 54);
graphics2D.dispose();
```

You can compare the result of this example to the "Hello World" files you produced using the basic building block or low-level approach. They're identical.

This third way of adding content is especially interesting if you're writing GUI applications using Swing components or objects derived from `java.awt.Component`. These objects can paint themselves to a `Graphics2D` object, and therefore they can also paint themselves to PDF using iText's `PdfGraphics2D` object. Chapter 12 will show you how to write the content displayed on the screen in a GUI application to a PDF file. What you see on the screen is what you'll get on paper. There is no PDF syntax involved; it's just standard Java.

FAQ *How do you solve X problems?* On UNIX systems, people working with this `PdfGraphics2D` object—or even with simple methods that use the `java.awt.Color` class—may encounter X11 problems that prompt this error message: *Can't connect to X11 window server using* xyz *as the value of the DISPLAY variable.*

The Sun AWT classes on UNIX and Linux have a dependency on the X Window System: You must have X installed in the machine; otherwise none of the packages from `java.awt` will be installed. When you use the classes, they expect to load X client libraries and to be able to talk to an X display server. This makes sense if your client has a GUI. Unfortunately, it's required even if your client uses AWT but, like iText, doesn't have a GUI.

You can work around this issue by running the AWT in headless mode by starting the Java Virtual Machine (JVM) with the parameter `java.awt.headless=true`.

Another solution is to run an X server. If you don't need to display anything, a virtual X11 server will do.

You've said "Hello" to the world many times, creating PDF documents from scratch in many different ways. You may have an idea by now of which approach suits your needs best. Only one step is left, which you must not forget—or you'll end up with a PDF file that misses its *cross-reference table* and its *trailer*—two important structures that are mandatory in a PDF file.

2.1.5 Closing the document

Let's restate the five steps to create a PDF document:

1. Create a `Document`.
2. Create a `PdfWriter` using `Document` and `OutputStream`.
3. Open the `Document`.
4. Add content to the `Document`.
5. Close the `Document`.

Some people may express serious doubts about this choice of design, because the iText approach seems to be in violation of the MVC pattern. You may ask why iText wasn't designed like this:

Model

1. Create a `Document`.
2. Add content to the `Document`.

View

3. Create a `PdfWriter/RtfWriter/`... using `OutputStream`.
4. Write the `Document` using `PdfWriter/RtfWriter/`....

The advantage of such a design, as advocates of the MVC pattern keep telling me, is that the `Document` would then act as an Object-Oriented (OO) model, encapsulating the document data—the content—so that it can be arbitrarily written to any specific output location and/or format on demand.

Design pattern

The iText design was inspired by the *builder pattern*, a pattern that's used to create a variety of complex objects from one source object. With iText, when you're adding content (step ❹), you've already decided how and where this content should be written (step ❷), thus mixing content encapsulation with generation and presentation. Is that so bad? Please look at the other side of the coin before answering this question.

Imagine you have a document consisting of more than 10,000 pages. Are you really going to keep all those pages in memory, risking an `OutOfMemoryError` before writing even the first byte of the document representation? Will you store the content in another format, in an object in memory, or in XML on the file system, before you convert it to PDF or RTF? The answer to these questions could be yes, but you'd only need to do this if you wanted to examine the contents of the document programmatically (which is beyond the scope of iText) or if you didn't find out which output format you wanted until you finished gathering the data. These are typically issues that are difficult, if not impossible, to solve when you're dealing with very large documents. If you compare document generation to XML parsing, the advantages of iText are similar to the advantages of the Simple API for XML (SAX) over the Document Object Model (DOM). Any DOM variant is well known to be suitable only when the data won't be very large, and SAX is provided as an alternative for parsing extremely large XML documents. Behind the scenes, SAX is often used to build the DOM tree. By analogy, you can build an MVC-compliant application that uses iText as the underlying engine to create the View. You can store the Model in a custom service object, create a `Document` instance to which you add a listener, and finally pass it to your service object, so that your object can write its content to the iText `Document`. That isn't a bad design. As a matter of fact, lots of applications use iText for that purpose.

Nevertheless, there are many projects for which this design just doesn't work. Think of business processes that have to be very fast—for instance, the creation of large documents that must be served in a web application, or batch jobs that take a whole night. In such circumstances, you'll be happy iText works the way it does. One of iText's strengths is its high performance. During step ❹, iText writes and flushes all kinds of objects to the `OutputStream`, the most important objects being the page dictionaries and page streams of all the pages as soon as they're completed. All these objects become eligible for garbage collection, keeping the amount of memory used relatively low compared to some other PDF-producing tools. You can't achieve this if you don't specify the `DocWriter` and the `OutputStream` first.

PDF cross-reference table and trailer

Upon closing the `Document`, the PDF objects that have to be kept in memory (because they must be updated from time to time) are written to the `OutputStream`. These include the following:

- The *PDF cross-reference table*, an important table that contains the byte positions of the PDF objects
- The *PDF trailer*, which contains information that enables an application to quickly find the start of the cross-reference table and certain special objects, such as the info dictionary

Finally, the `String %%EOF` (End of File) is added. After all this is done, the `OutputStream` created in step ❷ is flushed and closed. You've successfully created a PDF file.

The next chapter will list different types of PDF, not all of which are supported in iText. I'll use the phrase *traditional PDF* to refer to the most common type of PDF. Traditional PDF is intended to be a read-only, graphical format; it's designed to be electronic paper. When text is printed on paper, you can't add an extra word in the middle of a sentence and expect the layout of the paragraph to adapt automatically. The same is true for traditional PDF; it's not a format that is suited for editing. This doesn't mean you can't perform a series of other operations: You can stamp a piece of paper, cut it into pieces, copy one or more sheets, and perform other changes as well. Those sorts of changes are exactly what you'll perform on a traditional PDF file with iText classes such as `PdfStamper` and/or `PdfCopy`.

You'll also use `PdfStamper` to fill in the fields of a PDF form programmatically. Such a PDF document has a series of fields at specific coordinates on one or more pages. An end user can fill in these fields, but you, as a developer, can also use a PDF form as a template; iText is able to retrieve the absolute position of each field and add data at these coordinates.

All this functionality will be introduced in the next section, which discusses manipulation classes.

2.2 Manipulating existing PDF files

Imagine you're selling audio and video equipment in a branch office of a major electronics dealer. The mother company has sent you a product catalog in PDF with hundreds of pages. It contains sections on computers, digital cameras, televisions, radios, dishwashers, and so forth. Suppose you want to distribute a similar catalog among your clientele.

You can't use the original product catalog from your dealer because you're not even selling half of the products mentioned in it. You know your customers won't be interested in kitchen equipment—they want to read about the new features of the latest-model DVD players. For that reason, you want to compose a reduced catalog that only contains the pages that are relevant for your store. If possible, each page should have a header, footer, or watermark with the name and logo of your store.

Because PDF wasn't conceived to be a word-processing format, creating this new, personalized catalog is complex. It's not sufficient to cut some pages from one PDF file and paste them into another. Searching the Internet, you'll find lots of small tools and applications that offer this specialized functionality—such as Pdftk, jImposition, and SheelApps PDFTools—but if you study these more closely, you'll find that most of them use iText under the hood (even tools that cost several hundred dollars).

Before spending any money or time on a tool that may or may not solve your problem, look at the upcoming subsections. They will show you how these tools work, and you'll be able to tailor your own PDF-manipulation solution using the iText API directly. You'll learn that the `PdfCopy` class is best suited to copy a selection of pages from a series of different, existing PDF files. Adding new content (such as a logo, page numbers, or a watermark) is best done with the `PdfStamper` class.

The relationship between the different manipulation classes is shown in the class diagram in appendix A section A.2. `PdfCopy` is a subclass of `PdfWriter`, whereas `PdfStamper` has an implementation class that is derived from `PdfWriter`. These classes are *writers*, they can't *read* PDF files.

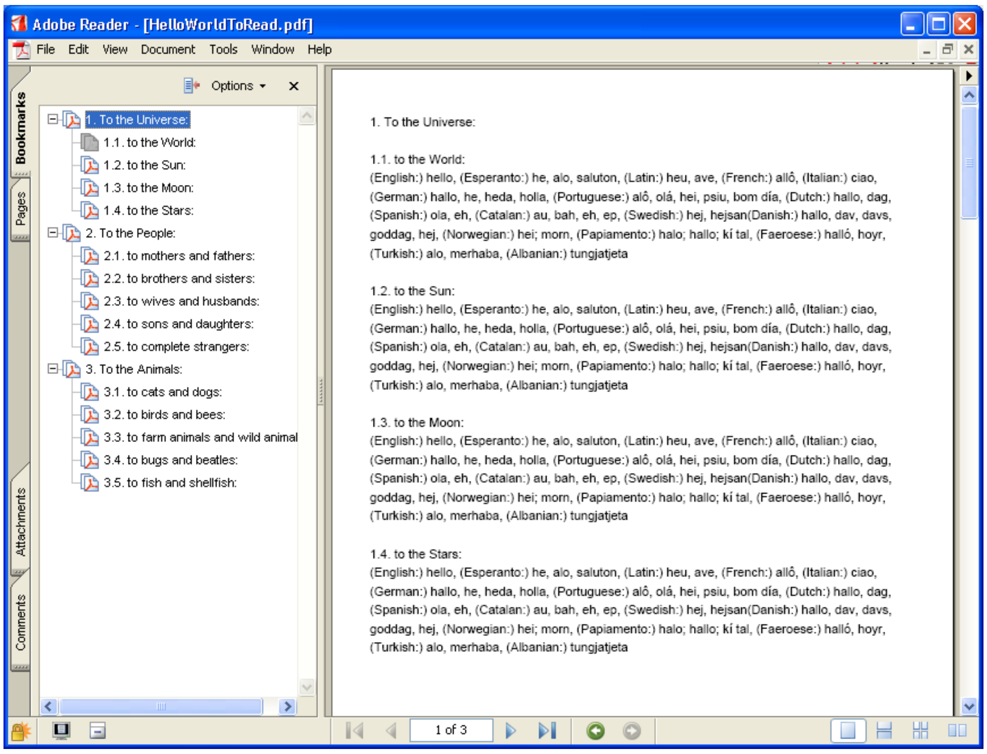
To read an existing PDF file, you need the class `PdfReader`; the actual work is done in the `PdfReaderInstance` class, but you'll never address this instance directly. As shown in the class diagram, `PdfReaderInstance` is for internal use by `PdfWriter` only.

Let's begin by examining the `PdfReader` class and find out what information you can retrieve from a PDF document before you start manipulating one or more PDF files with `PdfStamper`, `PdfCopy`, and the other classes mentioned in the class diagram.

2.2.1 Reading an existing PDF file

Before you start manipulating files, let's generate a PDF file with some functionality that is more complex than a "Hello World" document. Figure 2.3 shows the first page of the document HelloWorldToRead.pdf. As you can see, you can open the Bookmarks tab to see the outline tree of the document.

You'll learn how to create bookmarks in chapters 4 and 13. For the moment, we're only interested in `PdfReader` and how to retrieve the information from this


Adobe Reader - [HelloWorldToRead.pdf]
File Edit View Document Tools Window Help
Options
Bookmarks
Pages
Attachments
Comments
1 of 3

1. To the Universe:
1.1. to the World:
1.2. to the Sun:
1.3. to the Moon:
1.4. to the Stars:
2. To the People:
2.1. to mothers and fathers:
2.2. to brothers and sisters:
2.3. to wives and husbands:
2.4. to sons and daughters:
2.5. to complete strangers:
3. To the Animals:
3.1. to cats and dogs:
3.2. to birds and bees:
3.3. to farm animals and wild animal
3.4. to bugs and beatles:
3.5. to fish and shellfish:

1. To the Universe:

1.1. to the World:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom día, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

1.2. to the Sun:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom día, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

1.3. to the Moon:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom día, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

1.4. to the Stars:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom día, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

Figure 2.3 The existing PDF file you'll inspect with PdfReader

PDF file. You'll retrieve general properties, such as the file size and PDF version, the number of pages, and the page size, and also metadata and the bookmark entries.

Document properties

The following example demonstrates how to perform some of the basic queries: determining the version of the PDF file, the number of pages, the file length, and whether the PDF was encrypted:

```
/* chapter02/HelloWorldReader.java */
PdfReader reader = new PdfReader("HelloWorldToRead.pdf");
System.out.println("PDF Version: " + reader.getPdfVersion());
System.out.println("Number of pages: " +
  reader.getNumberOfPages());
System.out.println("File length: " + reader.getFileLength());
System.out.println("Encrypted? " + reader.isEncrypted());
```

Returns 4

Returns 3

Returns 8439

Returns false

The information returned in this code snippet is related to the complete document, but you can also ask the reader for information on specific pages.

Page size and rotation

Section 2.1.1 talked about rotating the page size `Rectangle`. In the HelloWorldReader example, you create a PDF document with three pages. The first two are A4 pages in portrait orientation, and the third is rotated with the `rotate()` method.

Now you'll ask those pages for their page size:

```
/* chapter02/HelloWorldReader.java */
System.out.println("Page size p1: " + reader.getPageSize(1));
System.out.println("Rotation p1: " +
  reader.getPageRotation(1));
System.out.println("Page size p3: " +
  reader.getPageSize(3));
System.out.println("Rotation p3: " +
  reader.getPageRotation(3));
System.out.println("Size with rotation p3: " +
  reader.getPageSizeWithRotation(3));
```

Returns 595.0x842.0 (rot. 0 degrees)

Returns 0

Returns 595.0x842.0 (rot. 0 degrees)

Returns 90

Returns 842.0x595.0 (rot. 90 degrees)

If you ask for the page size with the method `getPageSize()`, you always get a `Rectangle` object without rotation (rot. 0 degrees)—in other words, the paper size without orientation. That's fine if that's what you're expecting; but if you reuse the page, you need to know its orientation. You can ask for it separately with `getPageRotation()`, or you can use `getPageSizeWithRotation()`.

The annotations alongside the code sample show the results of the `toString()` method of class `Rectangle`. The second page size query didn't return what you would expect for page three; the last one gives you the right value and indicates that the page was rotated 90 degrees.

TOOLBOX *com.lowagie.tools.plugins.InspectPDF (Properties)* If you want a quick inspection of some of the properties of your PDF file, you can do this with the InspectPDF tool in the iText Toolbox.

Not every PDF tool produces documents that are 100 percent compliant with the PDF Reference. Also, if you have the audacity to change a PDF file manually (something you should attempt only if your PDF Fu is truly mighty), the offsets of the different objects will change. This makes the PDF document corrupt, and there may be a problem if the file is read.

Reading damaged PDFs

When you open a corrupt PDF file in Adobe Reader, you get this message: *The file is damaged and can't be repaired*. `PdfReader` will probably also throw an exception when you try to read such a file; because it *is* damaged and it *can't* be repaired. There's nothing iText can do about it.

In other cases—for example, if the cross-reference table is slightly changed—Adobe Reader only shows you this warning: *The file is damaged but is being repaired*. `PdfReader` can also overcome similar small damages to PDF files. Because iText isn't necessarily used in an environment with a GUI, no alert box is shown, but you can check whether a PDF was repaired by using the method `isRebuilt()`:

```
/* chapter02/HelloWorldReader.java */
System.out.println("Rebuilt? " + reader.isRebuilt());
```

When trying to manipulate a large document, another problem can occur: You can run out of memory. Augmenting the amount of memory that can be used by the JVM is one way to solve this problem, but there's an alternative solution.

PdfReader and memory use

When constructing a `PdfReader` object the way you did in the previous examples, all pages are read during the initialization of the reader object. You can avoid this by using another constructor:

```
/* chapter02/HelloWorldPartialReader.java */
PdfReader reader;
long before;
before = getMemoryUse();
reader = new PdfReader(                    | Does full read of
  "HelloWorldToRead.pdf", null);           | PDF file
System.out.println("Memory used by the full read: "      | Returns about
  + (getMemoryUse() - before));                          | 30 KB
before = getMemoryUse();
reader = new PdfReader(                                         | Does partial
  new RandomAccessFileOrArray("HelloWorldToRead.pdf"), null);  | read of PDF file
System.out.println("Memory used by the partial read: "     | Returns about
  + (getMemoryUse() - before));                            | 3.5 KB
```

The size of HelloWorld.pdf is about 5 KB. If you do a full read, a little less than 30 KB of the memory is used by the (uncompressed) content and the iText objects that contain the object. By using the object `com.lowagie.text.pdf.RandomAccesFileOrArray` in the `PdfReader` constructor, barely 3.5 KB of the memory is used initially. More memory will be used as soon as you start working with the object, but `PdfReader` won't cache unnecessary objects. If you're dealing with large documents, consider using this constructor.

Now that you've tackled some problems with corrupt or large PDFs, you can go on retrieving information.

Retrieving bookmarks

In figure 2.3, the Bookmarks tab is open. The class `com.lowagie.text.pdf.SimpleBookmark` can retrieve these bookmarks if you pass it a `PdfReader` object. You can retrieve the bookmarks in the form of a `List`:

```
/* chapter02/HelloWorldBookmarks.java */
PdfReader reader = new PdfReader("HelloWorldToRead.pdf");
List list = SimpleBookmark.getBookmark(reader);
```

This is an `ArrayList` containing a `Map` with the properties of the bookmark entries. If you run this example, the titles of the outline tree shown in figure 2.3 is written to `System.out`.

With the static method `SimpleBookmark.exportToXML`, this list of bookmarks can also be exported to an XML file:

```
/* chapter02/HelloWorldBookmarks.java */
SimpleBookmark.exportToXML(list,
  new FileOutputStream("bookmarks.xml"), "ISO8859-1", true);
```

You'll learn more about the bookmark properties and about the structure of this XML file in chapter 13.

TOOLBOX *`com.lowagie.tools.plugins.HtmlBookmarks` (Properties)* Suppose you have many PDFs on your web site, all having an extensive table of contents in the form of an outline tree. Wouldn't it be great to be able to extract these outlines and serve them to site visitors in the form of an HTML index file with links to every entry in the PDF outline tree? That way, if visitors are looking for a specific chapter, they don't have to download and browse every PDF file. Instead, they can browse through the HTML files first and click a link to go to a specific page within a PDF file. The HtmlBookmarks tool offers such index files—the only thing you have to do is to provide a Cascading Style Sheets (CSS) file that goes with it.

Metadata can also contain information that is useful to display in an HTML file before the visitor of your site downloads the complete document. You can use `PdfReader` to extract the metadata from the PDF files in your repository and store this information somewhere so that the repository can be searched.

Reading metadata

When you created the file HelloWorldToRead.pdf, you added metadata. The PDF-specific metadata of the document is kept in the PDF info dictionary. `PdfReader` can retrieve the contents of this dictionary as a `(Hash)Map` using the method `getInfo()`:

```
/* chapter02/HelloWorldReadMetadata.java */
PdfReader reader = new PdfReader("HelloWorldToRead.pdf");
Map info = reader.getInfo();
String key;
String value;
for (Iterator i = info.keySet().iterator(); i.hasNext(); ) {
  key = (String) i.next();
  value = (String) info.get(key);
  System.out.println(key + ": " + value);
}
```

Now that you've retrieved the metadata, let's try to change the `Map` returned by `getInfo()`. This will introduce the `PdfStamper` class.

2.2.2 *Using PdfStamper to change document properties*

`PdfStamper` is the class you'll use if you want to manipulate a single document. This is how you create an instance of `PdfStamper`:

```
/* chapter02/HelloWorldAddMetadata.java */
PdfReader reader = new PdfReader("HelloWorldNoMetadata.pdf");
System.out.println("Tampered? " + reader.isTampered());
PdfStamper stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldStampedMetadata.pdf"));
System.out.println("Tampered? " + reader.isTampered());
```

Notice that as soon as you create a `PdfStamper` object, the reader is *tampered*—that is, the `PdfStamper` instance alters the reader behind the scenes so it can't be used with any other `PdfStamper` instance. `PdfStamper` is often used to stamp data from a database on the same document over and over again. For example, suppose you've created a standard letter for your customers using Acrobat. You have all the names of your customers in a database. Now you want to merge the results of a database query with this letter. You can do this by reading the original PDF with `PdfReader` and stamping it with `PdfStamper`.

FAQ *Why do I get an exception when I try to create a `PdfStamper` instance?* Novice iText users often make the mistake of trying to reuse the reader instance. A `DocumentException` will be thrown, saying: *The original document was reused. Read it again from file*. This is normal: `PdfStamper` needs a unique and exclusive `PdfReader` object. Tampered reader objects can't be reused.

Note that it's impossible to write to the file you're reading. `PdfReader` does random-access file reading on the original file, so it's important to realize that the original and the manipulated file can't have the same name. Few programs read a file and change it at the same time; most of them write to a temporary file and replace the original file afterward. If that's what you want, that's how you should implement it; but you can also read the original file into a byte array, create the `PdfReader` object using this array, and write the output of the stamper to a file with the same name as the original PDF.

That being said, you can write some code to change the metadata of an existing PDF file. You get the information `(Hash)Map` from the reader ❶, add some extra keys and values ❷, and then add it to the stamper object with the method `setMoreInfo()` ❸:

```
/* chapter02/HelloWorldAddMetadata.java */
Map info = reader.getInfo();   <-❶
info.put("Subject", "Hello World");
info.put("Author", "Bruno Lowagie");      ❷
stamper.setMoreInfo(info);   <-❸
stamper.close();   <-❹
```

Don't forget to close the stamper ❹! Otherwise you'll end up with a file of 0 KB.

In the next chapter, you'll learn how to use `PdfStamper` to change other properties of a PDF file, such as the compression, the encryption, and the user permissions of a file. The rest of this chapter will focus on adding content to an existing PDF file.

2.2.3 Using PdfStamper to add content

Let's return to our earlier example. You're selling audio and video equipment, and you want to send a standard letter to all of your customers telling them about the personalized catalog they can order. This letter is provided as a PDF document containing a PDF form. In this case, the form's fields (called *AcroFields*) correspond to the fields of individual records in your customer database. You can now use iText to fill in those fields.

Filling in a form

It's possible to create a document containing a PDF form (also called an *AcroForm*) with iText, and you'll learn more about that in chapter 15; but using an end-user tool like Acrobat is a better way to make a quality design. Chapter 16 will explain how to fill and process forms. This is a crash course on document manipulation, so let's have a small taste of form functionality.

You start with a simple PDF saying "Hello Who?" The word "Who?" is gray deliberately; you may not notice that it's a form field just by looking at it, but if you hover the cursor over this word, you'll see the cursor changes from a little hand into an I-bar. Click the area, and you can edit the word. One possible use of a PDF form is to have people fill in the form and submit it, but for now you're more interested in using the form as a template and filling it out programmatically:

```
/* chapter02/HelloWorldForm.java */
PdfReader reader = new PdfReader("HelloWorldForm.pdf");
PdfStamper stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldFilledInForm.pdf"));
AcroFields form = stamper.getAcroFields();          <-- Gets form from stamper
form.setField("Who", "World");   <-- Sets field in form
stamper.close();
```

Granted, the design of this HelloWorldForm is simple, but that doesn't matter. You can create forms with multiple fields in a complex design; it won't make your code more complex. You just ask the `PdfStamper` object for its `AcroFields` object and change the value of all the fields inside the form.

This example changes the word "Who?" that was in the Who field into the word "World." The result is a new PDF file that still contains a form; but it now says "Hello World" instead of "Hello Who?" If you click the word "World," you can change it into something else. This may not always be what you want; in some cases, you don't want the end user to know you have used a PDF form as a template. The resulting PDF shouldn't be interactive once it's filled in.

That's why you'll *flatten* the form. Flattening means there are no longer any editable field in the new PDF. The field content is added at the position where the field was defined; an end user can't change the text:

```
/* chapter02/HelloWorldForm.java */
stamper.setFormFlattening(true);
```

In chapter 16, you'll discover lots of tips and tricks to optimize the process of filling and flattening a PDF form—for example, how to make sure the text fits the field, or how to use a field as a placeholder for an image.

But what if you need to add content to an existing PDF document *without* a form? Can you still use it as a template and add extra content? The answer is yes, you can—if you know where (on which coordinates) to add the new content.

Adding content to pages

Think of the personalized catalog you want to compose. The original catalog doesn't contain a form, but you want to take the existing PDF file, add a watermark with your company logo in the middle of each page (under the existing content), and add page numbers to the bottom of the pages. Again, you need the `PdfStamper` class to achieve this.

Do you remember the `PdfContentByte` object, which you used to add text at an absolute position? With `PdfStamper`, you can get two different `PdfContentByte` objects per page. The method `getOverContent(int pagenumber)` gives you a canvas on which to draw text and graphics that are painted on top of the existing content.

The next code snippet uses this method to add page numbers and draws a circle at an absolute position:

```
/* chapter02/HelloWorldStamper.java */
PdfContentByte over = stamper.getOverContent(i);
over.beginText();
over.setFontAndSize(bf, 18);
over.setTextMatrix(30, 30);
over.showText("page " + i);
over.endText();
over.setRGBColorStroke(0xFF, 0x00, 0x00);
over.setLineWidth(5f);
over.ellipse(250, 450, 350, 550);
over.stroke();
```

With the method `getUnderContent(int pagenumber)`, you can get a canvas that appears under the existing content. For example, you can add a watermark to every page, like this:

```
/* chapter02/HelloWorldStamper.java */
PdfReader reader = new PdfReader("HelloWorld.pdf");
PdfStamper stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldStamped.pdf"));
Image img = Image.getInstance("watermark.jpg");
img.setAbsolutePosition(200, 400);
PdfContentByte under;
int total = reader.getNumberOfPages() + 1;
for (int i = 1; i < total; i++) {
  under = stamper.getUnderContent(i);
  under.addImage(img);
}
stamper.close();
```

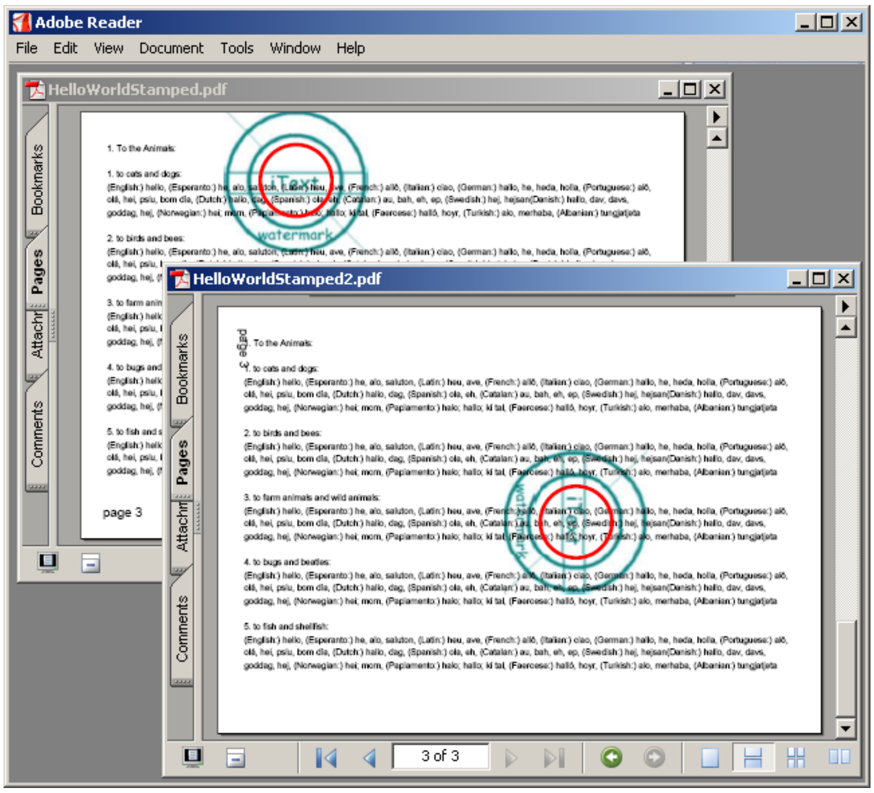
Remember the importance of page orientation. In the HelloWorld.pdf file, the third page has landscape orientation. If you're adding text, graphics, or an image at an absolute coordinate, you have to realize that the coordinate system has been changed, too. You're working on a canvas with dimensions set in `height x width` instead of `width x height`. If this isn't what you want, you can avoid it by setting `setRotateContents` to `false`:

```
/* chapter02/HelloWorldStamper2.java */
stamper.setRotateContents(false);
```

Take a close look at figure 2.4, and compare the third pages of the documents HelloWorldStamped.pdf and HelloWorldStamped2.pdf.

In HelloWorldStamped.pdf, the page rotation has been taken into account, and the text and graphics have been added so that you can read them without having to turn your head 90 degrees. This also means you should have adjusted the position of the watermark—it isn't exactly where you want it to be. In HelloWorldStamped2.pdf, the text and graphics were added as if the page was still in portrait orientation.

Not only can `PdfStamper` be used to change existing pages, but you can also insert new blank pages to which content can be added.



1. To the Animals:

1. to cats and dogs:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom dia, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

2. to birds and bees:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom dia, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

3. to farm animals and wild animals:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom dia, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

4. to bugs and beatles:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom dia, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

5. to fish and shellfish:
(English:) hello, (Esperanto:) he, alo, saluton, (Latin:) heu, ave, (French:) allô, (Italian:) ciao, (German:) hallo, he, heda, holla, (Portuguese:) alô, olá, hei, psiu, bom dia, (Dutch:) hallo, dag, (Spanish:) ola, eh, (Catalan:) au, bah, eh, ep, (Swedish:) hej, hejsan(Danish:) hallo, dav, davs, goddag, hej, (Norwegian:) hei; morn, (Papiamento:) halo; hallo; kí tal, (Faeroese:) halló, hoyr, (Turkish:) alo, merhaba, (Albanian:) tungjatjeta

Figure 2.4 Taking the page rotation into account when stamping a PDF

Inserting new pages

In the next example, you'll add a title page to an existing PDF document:

```
/* chapter02/HelloWorldStamperAdvanced.java */
stamper.insertPage(1, PageSize.A4);
PdfContentByte cb = stamper.getOverContent(1);
cb.beginText();
cb.setFontAndSize(bf, 18);
cb.setTextMatrix(36, 770);
cb.showText("Inserted Title Page");
cb.endText();
```

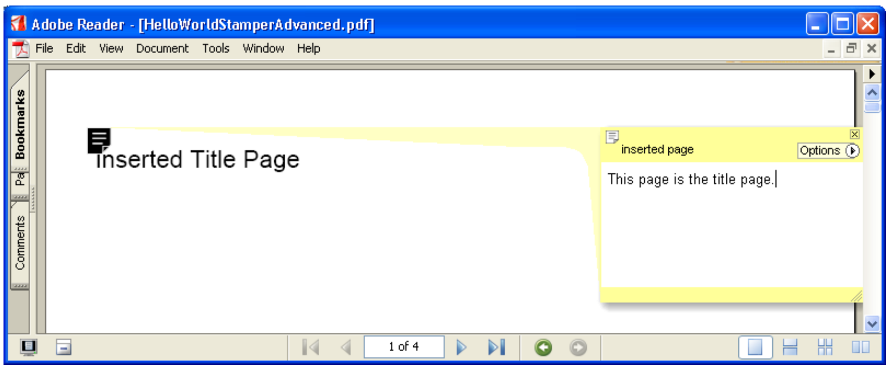
I also threw in some more advanced functionality:

```
/* chapter02/HelloWorldStamperAdvanced.java */
stamper.addAnnotation(
  PdfAnnotation.createText(stamper.getWriter(),
    new Rectangle(30f, 750f, 80f, 800f),
    "inserted page", "This page is the title page.",
    true, null), 1);
```

This adds a comment on the first page. When the comment is closed, you see a page icon; the comment title and text are visible only if you move the mouse pointer over the comment. Figure 2.5 shows the text annotation in its opened state. (Annotations are discussed in chapter 15.)

Notice that the page numbers shift when inserting a new page—not the page numbers that are printed on the page, but the indices used to retrieve the page from the `PdfStamper` object. Make sure you keep track of the actual page count if you're inserting and retrieving pages using one stamper object!

Let's return to the idea of a personalized catalog. You already have two useful pieces of the puzzle: You can stamp a logo on each page, and you can add an


Adobe Reader - [HelloWorldStamperAdvanced.pdf]
File Edit View Document Tools Window Help
Bookmarks
Pages
Comments
1 of 4
Options

Inserted Title Page

inserted page

This page is the title page.

Figure 2.5 A PDF with an open text annotation

extra title page. You can't remove the sections presenting the newest types of dishwasher yet, but it would be nice if you could illustrate the title page with thumbnails of existing pages, such as the title page of the section on DVD players. In other words, you want to copy a complete page and paste a smaller version of it on another page. To achieve this, you need *imported pages*.

2.2.4 Introducing imported pages

If you browse the API of the `PdfReader` class, you'll discover the method `getPageContent(int pagenumber)`, which returns the content stream of that page. You've already seen the content stream of a simple "Hello World" page in listing 2.2. This stream tells you what's inside a page, but it doesn't necessarily return the complete page.

A content stream normally contains references to external objects, images, and fonts. For example, you can find a reference to a font named /F1 in listing 2.2. This font is stored elsewhere in the PDF file. It's possible to retrieve every object that is needed to copy an existing page, but it takes a fair amount of coding and you need to know the Portable Document Format inside out.

That's why it's never advisable to extract a page from `PdfReader` directly. Instead, you should pass the reader object to the manipulation class (`PdfStamper`, `PdfCopy`, or even `PdfWriter`) and ask the writer (not the reader!) for the imported page. A `PdfImportedPage` object is returned. Behind the scenes, all the necessary resources (such as images and fonts) are retrieved. As you'll see in chapter 18, importing pages this way not only saves you a lot of work, but is also less error-prone.

Here's an example using `PdfStamper`:

```
/* chapter02/HelloWorldStamperImportedPages.java */
PdfReader reader = new PdfReader("HelloWorldRead.pdf");
PdfStamper stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldImportedPages.pdf"));
PdfImportedPage p;
stamper.insertPage(1, PageSize.A4);   <-- Inserts new first page
PdfContentByte cb = stamper.getOverContent(1);
p = stamper.getImportedPage(reader, 2);     <-- Imports second page
cb.addTemplate(p, 0.4f, 0f, 0f, 0.4f, 36f, 450);
p = stamper.getImportedPage(reader, 3);       <-- Imports third page
cb.addTemplate(p, 0.4f, 0f, 0f, 0.4f, 300f, 450);
p = stamper.getImportedPage(reader, 4);        <-- Imports fourth page
cb.addTemplate(p, 0.4f, 0f, 0f, 0.4f, 36f, 100);
```

Instead of inserting a page with text saying "inserted title page," you insert a first page that shows downsized versions of the pages that follow. With the method getImportedPage(), you pass the PdfReader object to a PdfStamper, and you tell the stamper which page you want to import.

The object that is returned is of type PdfImportedPage. It contains a description of the contents of the page; the resources that are referred to from this page are passed to PdfStamper behind the scenes. Note that you can't add new content to a PdfImportedPage object; you can only scale, rotate, and/or translate it while adding it to another page. The example uses the addTemplate method to scale and position the thumbnails. The float values that are passed to this method are elements of a *transformation matrix*. (You'll read all about the transformation matrix in chapter 10.)

There's still a lot to say about PdfStamper. We haven't discussed how you can sign an existing document, change viewer preferences, and so forth, but we'll cover all of that in part 4, "Interactive PDF."

Let's elaborate on these imported pages first.

2.2.5 Using imported pages with PdfWriter

PdfStamper is able to retrieve and (re)use imported pages, but other classes may be better suited for the job. If you're using PdfStamper, it's assumed that you want to manipulate one and only one existing PDF file. But maybe you want to create a document from scratch and use pages from an existing document as new content. If we're talking about generating a document from scratch, we automatically think of PdfWriter.

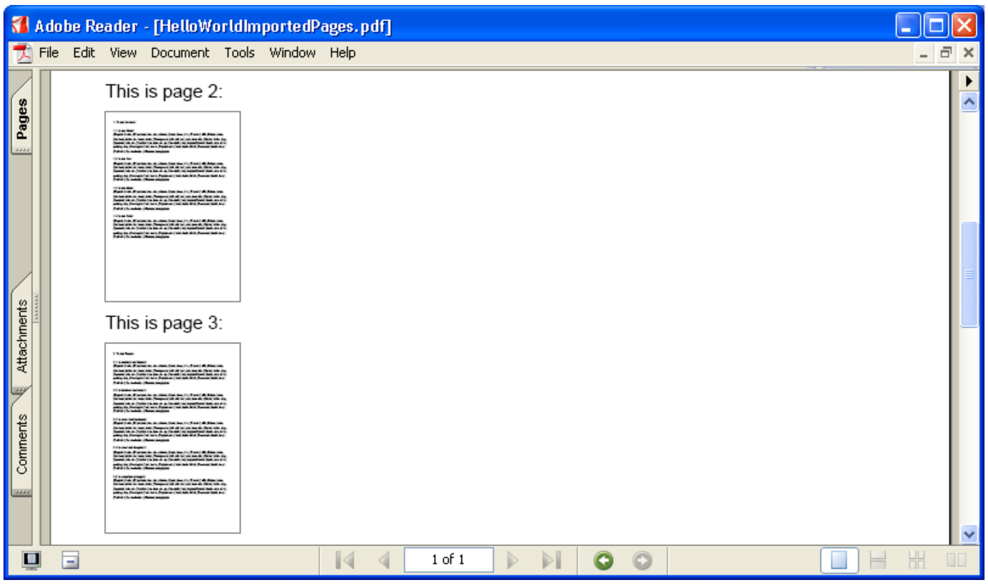
What you did in the PdfStamperImportedPages example can also be done in step ❹ of the PDF creation process we'll discuss in chapter 3. If you wrap the PdfImportedPage in an Image object (as will be discussed in section 5.3.4), it's easy to manipulate the imported page. Figure 2.6 shows how the pages of an existing PDF document are used as thumbnails in a new document.

In this example, you wrap the imported page inside an image ❶, scale it to 15 percent of its original size ❷, draw a gray box that is three units thick around it ❸, and add it to the page ❹:

```
/* chapter02/HelloWorldImportedPages.java */
PdfReader reader = new PdfReader("HelloWorldToImport.pdf");
PdfWriter writer = PdfWriter.getInstance(document,
  new FileOutputStream("HelloWorldImportedPages.pdf"));
document.open();
System.out.println("Tampered? " + reader.isTampered());
document.add(new Paragraph("This is page 1:"));
PdfImportedPage page = writer.getImportedPage(reader, 1);
Image image = Image.getInstance(page);    <-❶
image.scalePercent(15f);   <-❷
image.setBorder(Rectangle.BOX);
image.setBorderWidth(3f);
image.setBorderColor(new GrayColor(0.5f));      3
document.add(image);   <-4
System.out.println("Tampered? " + reader.isTampered());
document.close();
```

This functionality can be handy if you want to invite customers to order the complete product catalog. You can make a flyer with the description of the content of the catalog along with some thumbnails showing the most interesting and attractive pages.

Note the `System.out` lines: I added them to show that importing pages with `PdfWriter` doesn't tamper with the reader object. A reader object used by `PdfWriter` isn't exclusively tied to the writer as was the case with `PdfStamper`. This may sound unimportant, but once you get to know iText well, you'll understand that you can improve your applications drastically by choosing the right object for the right job. Throughout this book, I'll present different ways to achieve the same result. If performance is an issue, you should try different solutions, benchmark them in your specific working environment, and use the best solution in your production software.


Adobe Reader - [HelloWorldImportedPages.pdf]
File Edit View Document Tools Window Help
Pages
Attachments
Comments
1 of 1

This is page 2:

This is page 3:

Figure 2.6 Imported pages as thumbnails

The next example is a little more complex. It places the four pages of an existing document on one page of a new document, so that the document can be folded into a booklet. If you have a four-page brochure presenting your products, you can use this code to print the four pages on one page in such a way that the page can be folded to fit inside an envelope:

```
/* chapter02/HelloWorldWriter.java */
PdfReader reader = new PdfReader("HelloWorldToImport.pdf");
PdfWriter writer = PdfWriter.getInstance(document,
 new FileOutputStream("HelloWorldFolded.pdf"));
document.open();
PdfContentByte cb = writer.getDirectContent();
PdfImportedPage page;
page = writer.getImportedPage(reader, 1);
cb.addTemplate(page, -0.5f, 0f, 0f, -0.5f,
  PageSize.A4.width() / 2, PageSize.A4.height());       ❶
page = writer.getImportedPage(reader, 2);
cb.addTemplate(page, 0.5f, 0f, 0f, 0.5f, 0f, 0f);       ❷
page = writer.getImportedPage(reader, 3);
cb.addTemplate(page, 0.5f, 0f, 0f, 0.5f,
  PageSize.A4.width() / 2f, 0f);                        ❸
page = writer.getImportedPage(reader, 4);
cb.addTemplate(page, -0.5f, 0f, 0f, -0.5f,
  PageSize.A4.width(), PageSize.A4.height());           ❹
document.close();
```

The height and width of the first imported page ❶ are divided by 2; the page is turned upside down and added at the upper-left side of the new page. The second page ❷ is also scaled; it's added at the lower-left side of the new page. Page three ❸ is scaled and added next to page 2, at the lower-right side of the new page. The fourth page ❹ is scaled, rotated, and added next to page 1 at the upper-right side of the page.

TOOLBOX `com.lowagie.tools.plugins.NUp` *(Manipulate)* The N-up tool allows you to create a new PDF document based on an existing one. Each page of the new document contains N pages of the existing document, with N equal to 2, 4, 8, 16, 32, or 64.

There is one major downside when you're adding a page imported using `PdfWriter` (or with `PdfStamper.getImportedPage`) to a document. All interactive features (annotations, bookmarks, fields, and so forth) are lost in the process. If you want to import pages in order to concatenate several PDF files into one, this is a big disadvantage. That's where `PdfCopy` comes into the picture.

2.2.6 Manipulating existing PDF files with PdfCopy

You used PdfStamper to manipulate one and only one existing PDF file. PdfCopy is the class you need if you want to combine a selection of pages from one or multiple existing PDFs. This is the next puzzle piece you can use to create a personalized catalog.

You can distinguish different approaches. If you have different small catalogs per product line, you can concatenate the PDF files that are of importance to your customers into one catalog. If you have one big catalog with all the products, you can make a selection of specific pages and page ranges.

Concatenating PDF files

In the next example, you'll concatenate three pages from three different PDF documents into one new document. The first document contains a plain page with a paragraph; the second, a page with a text annotation; and the third, a page with an anchor. All of these features are preserved in the resulting three-page document:

```
/* chapter02/HelloWorldCopy.java */
PdfReader reader = new PdfReader("Hello1.pdf");
Document document = new Document(reader.getPageSizeWithRotation(1));
PdfCopy copy = new PdfCopy(document,
  new FileOutputStream("HelloWorldPdfCopy123.pdf"));
document.open();
System.out.println("Tampered? " + reader.isTampered());
copy.addPage(copy.getImportedPage(reader, 1));
reader = new PdfReader("Hello2.pdf");
copy.addPage(copy.getImportedPage(reader, 1));
reader = new PdfReader("Hello3.pdf");
copy.addPage(copy.getImportedPage(reader, 1));
System.out.println("Tampered? " + reader.isTampered());
document.close();
```

Again, you work with a getImportedPage() method, but this time you add the imported page to the manipulation class with the method addPage(). You don't scale or position the page; it's added as is. PdfCopy is a subclass of PdfWriter; the use of both classes is similar, but it's important to realize that PdfCopy can't be used to change the content of a PDF file. This time, you can't grab a PdfContentByte object; PdfCopy doesn't allow new content on a page. If you need to concatenate and stamp different PDF files (as you'll do with the personalized catalog), you must create the resulting PDF in multiple passes (see section 2.3).

When you run the example, you'll see that importing a page with PdfCopy doesn't tamper with PdfReader. You can reuse the reader object for different instances of `PdfCopy`—for example, if you need to add the same title page to a series of existing PDF files.

Selected pages

There are two ways to select pages from an existing PDF file. You can use `PdfCopy` to import the pages you need and add only those pages to the new document with the method `addPage()`, but there's a more elegant way to achieve this. You can use the method `selectPages()` on `PdfReader` to narrow the selection even before you start reading and copying.

The next code snippet uses this method to select the odd pages from the existing PDF file:

```
/* chapter02/HelloWorldSelectPages.java */
PdfReader reader = new PdfReader("HelloMultiplePages.pdf");
reader.selectPages("o");
int pages = reader.getNumberOfPages();
Document document = new Document();
PdfCopy copy = new PdfCopy(document,
  new FileOutputStream("HelloWorldSelectPagesOdd.pdf"));
document.open();
for (int i = 0; i < pages; ) {
  ++i;
  copy.addPage(copy.getImportedPage(reader, i));
}
document.close();
```

The general syntax for the range that is used in the `selectPages()` method looks like this: `[!][o][odd][e][even]start-end`. You can have multiple ranges separated by commas. The `!` modifier removes pages from what is already selected. The range changes are incremental—numbers are added or deleted as the range appears. The start or the end can be omitted. If you omit both, you need at least `o` (`odd`; selects all odd pages) or `e` (`even`; selects all even pages).

TOOLBOX *com.lowagie.tools.plugins.SelectedPages (Manipulate)* If you need to quickly create a new document from a selection of pages from an existing PDF file, you don't need to adapt the example that demonstrates the `selectPages` method. You can go to the iText Toolbox and use the SelectedPages plug-in instead.

Note that if you reuse a reader object from which you've removed pages, the pages remain removed; that's why you have to create a new `PdfReader` for every new selection in the example. The next code snippet selects pages 1, 2, 3, 7, and 9 (I excluded page 8):

```
/* chapter02/HelloWorldSelectPages.java */
reader = new PdfReader("HelloWorldMultiplePages.pdf");
reader.selectPages("1-3, 7-9, !8");
```

This `PdfReader` functionality can also be used in the context of a `PdfStamper` application.

```
/* chapter02/HelloWorldSelectedPages */
reader = new PdfReader("HelloMultiplePages.pdf");
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloSelectedEven.pdf"));
reader.selectPages("e");
stamper.close();
reader = new PdfReader("HelloMultiplePages.pdf");
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloSelected12379.pdf"));
reader.selectPages("1-3, 7-9, !8");
stamper.close();
```

Again, I'm presenting different ways to solve the same problem. It's up to you to experiment and choose the object that is best suited for your specific needs. For example, if you need to combine a selection of pages from different product catalogs, you'll probably prefer using `PdfCopy` over `PdfStamper`. In some cases, you'll even need another class: `PdfCopyFields`.

2.2.7 Concatenating forms with PdfCopyFields

Be careful with the next example: It shows you how *not* to combine PDF files with forms. It doesn't differ much from the example HelloWorldCopy, except that the pages you import now contain *form fields*:

```
/* chapter02/HelloWorldCopyForm.java */
PdfReader reader = new PdfReader("HelloWorldForm1.pdf");
Document document =
  new Document(reader.getPageSizeWithRotation(1));
PdfCopy writer = new PdfCopy(document,
  new FileOutputStream("HelloWorldCopyForm.pdf"));
document.open();
writer.addPage(writer.getImportedPage(reader, 1));
reader = new PdfReader("HelloWorldForm2.pdf");
writer.addPage(writer.getImportedPage(reader, 1));
reader = new PdfReader("HelloWorldForm3.pdf");
writer.addPage(writer.getImportedPage(reader, 1));
document.close();
```

When you open the resulting file HelloWorldCopyForm.pdf, you immediately see that something didn't work out the way you expected. HelloWorldForm1.pdf and

HelloWorldForm2.pdf each have a form containing one text field that has the same name: field1. After concatenating the files with `PdfCopy`, one of these fields got lost in the process.

That's just one of the problems you could potentially experience when copying forms using `PdfCopy`. `PdfCopy` only deals with the form in the first document. The form fields of the other documents are copied but not added to the initial form. The resulting PDF looks good in most cases, but as soon as you start to work with it, it will fail. This is an example of how you *shouldn't* concatenate forms.

To avoid problems when concatenating forms, you should use the class `PdfCopyFields`. This is the safest way to concatenate documents that have forms; but as you probably know, everything comes with a price—unlike `PdfCopy`, `PdfCopyFields` keeps all the documents in memory so the final form can be updated correctly. Make sure you have enough memory available:

```
/* chapter02/HelloWorldCopyFields.java */
PdfCopyFields copy =
  new PdfCopyFields(new FileOutputStream("HelloWorldCopyFields.pdf"));
copy.addDocument(new PdfReader("HelloWorldForm1.pdf"));
copy.addDocument(new PdfReader("HelloWorldForm2.pdf"));
copy.addDocument(new PdfReader("HelloWorldForm3.pdf"));
copy.close();
```

If you look at HelloWorldCopyFields.pdf, you now see that field1 is present on the first and the second page (with `PdfCopy`, it was missing on the second page). If you change one of these fields, the other fields with the same name are changed automatically, which is expected behavior.

I've been stressing the importance of choosing the right manipulation class for the right job. Now that you've worked with the different reader and writer classes for PDF manipulation available in iText, it's a good time for an overview.

2.2.8 Summary of the manipulation classes

When dealing with existing PDF documents, you can turn to table 2.2 to determine which manipulation class or classes can be used to perform the different aspects of your assignment.

You'll soon discover that choosing one class that solves all problems isn't possible. You'll have to combine different classes, and the most efficient way to do this is by creating a PDF in multiple passes.

Table 2.2 An overview of PDF manipulation classes

iText class	Usage
`PdfReader`	Read PDF files. In most cases, you have to pass an instance of this class to one of the PDF manipulation classes.
`PdfStamper`	Manipulate the content of an existing PDF document. For example, you can add page numbers, fill form fields, or sign an existing PDF file.
`PdfEncryptor`	Uses `PdfStamper` to encrypt an existing PDF file in a user-friendly way (see chapter 3).
`PdfWriter`	Generate PDF documents from scratch; import pages from other PDF documents. The major downside: All interactive features (annotations, bookmarks, fields, and so forth) of the imported page are lost in the process.
`PdfCopy`	Concatenate a selection of pages from one or multiple existing PDF forms. Major disadvantages: `PdfCopy` doesn't allow new content, and combining multiple forms into one is problematic.
`PdfCopyFields`	Put the fields of the different forms into one new form. Can be used to avoid the problems encountered with form fields when using `PdfCopy`, but remember that memory use can be an issue.

2.3 Creating PDF in multiple passes

You finally have all the pieces of the puzzle when it comes to manipulating existing PDF files, but now you have to start putting the puzzle together. For example, you know how to fill in one standard letter using one user record, but how do you combine all the letters into a single file so you can send it to a printing office?

One solution would be to use `PdfStamper` to fill in the fields of one PDF template form. `PdfStamper` can't add multiple forms that are filled in with different data to the same document. You could keep the stamped PDF in memory temporarily and do the concatenation with `PdfCopy`. As you remember, `PdfCopy` wasn't able to add new data to a document, so you need both classes: `PdfCopy` and `PdfStamper`.

Once you've chosen which manipulation class to use for which aspect of your assignment, you have to determine the best order to perform the manipulation. Will you stamp the existing PDFs first, and then copy? Or is it better to do it the other way around?

2.3.1 Stamp first, then copy

Let's say you have a standard letter in PDF (with a form) that says the following:

```
Dear ...
I just wanted to say Hello.
```

In place of the ellipsis, you want a name from your customer database, and you want to create a single document that has all the different versions of this letter, one per addressee. In this case, the first step is to stamp and flatten the original document. You don't need the individual files, so you keep the result in memory (in a `ByteArrayOutputStream`):

```
/* chapter02/HelloWorldStampCopy.java */
RandomAccessFileOrArray letter =
  new RandomAccessFileOrArray("HelloLetter.pdf");
reader = new PdfReader(letter, null);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
stamper = new PdfStamper(reader, baos);
form = stamper.getAcroFields();
form.setField("field", "World,");
stamper.setFormFlattening(true);
stamper.close();
```

Now you read the stamped and flattened file from memory and copy it:

```
/* chapter02/HelloWorldStampCopy.java */
reader = new PdfReader(baos.toByteArray());
Document document =
  new Document(reader.getPageSizeWithRotation(1));
PdfCopy writer = new PdfCopy(document,
  new FileOutputStream("HelloWorldStampCopy.pdf"));
document.open();
writer.addPage(writer.getImportedPage(reader, 1));
```

You can repeat this process as many times as you want:

```
/* chapter02/HelloWorldStampCopy.java */
reader = new PdfReader(letter, null);
baos = new ByteArrayOutputStream();
stamper = new PdfStamper(reader, baos);
form = stamper.getAcroFields();
form.setField("field", "People,");
stamper.setFormFlattening(true);
stamper.close();
reader = new PdfReader(baos.toByteArray());
writer.addPage(writer.getImportedPage(reader, 1));
```

This is just a simple example. You'll probably want to write some loops to handle all the copies with the same code and to copy all pages of the original document (instead of just the first one), but that shouldn't be a problem. Also, if file size and performance are an issue, it may be wiser to work with PdfWriter and page events as discussed in chapter 14.

2.3.2 Copy first, then stamp

We've dealt with having one form that was filled in multiple times using different data. Now we'll look at the best way to proceed when you want to combine different forms into one and then stamp the result. For example, suppose you have several different loan application forms—one for people who own a house and one for people who don't; one for people who own their own company and one for people who work for someone else. You want to be able to concatenate the forms in a personalized way depending on the applicant's individual situation, so that you have all the necessary data (and nothing more) in one big form.

In this case, it's probably better to start with the concatenation of the different forms:

```
/* chapter02/HelloWorldCopyStamp.java */
ByteArrayOutputStream baos = new ByteArrayOutputStream();
PdfCopyFields copy = new PdfCopyFields(baos);
copy.addDocument(new PdfReader("HelloWorldLetter1.pdf"));
copy.addDocument(new PdfReader("HelloWorldLetter2.pdf"));
copy.close();
```

HelloWorldLetter1.pdf has a form containing field1. HelloWorldLetter2.pdf has a form with field2. The resulting PDF (kept in memory) has one form containing both fields. You can stamp these fields like this:

```
/* chapter02/HelloWorldCopyStamp.java */
reader = new PdfReader(baos.toByteArray());
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldCopyStamp.pdf"));
form = stamper.getAcroFields();
form.setField("field1", "World");
form.setField("field2", "People");
stamper.setFormFlattening(true);
stamper.close();
```

Of course, it could happen that you want to combine different forms having the same field names for entities that are different in reality. For example, suppose you have a form that contains the fields name and income and that allows one person to declare his monthly revenues. When dealing with a couple, you need to know the income of both partners, so you want to combine two versions of the income form: one version with fields named name_husband and income_husband, and another with fields named name_wife and income_wife. In this case, you must rename these fields before you copy them.

2.3.3 Stamp, copy, stamp

Let's keep it simple and experiment with the original letter. You'll stamp it and use the method `renameField()` to change the name of the field:

```
/* chapter02/HelloWorldStampCopyStamp.java */
RandomAccessFileOrArray letter =
  new RandomAccessFileOrArray("HelloLetter.pdf");
reader = new PdfReader(letter, null);
ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
stamper = new PdfStamper(reader, baos1);
form = stamper.getAcroFields();
form.renameField("field", "field1");
stamper.close();
reader = new PdfReader("HelloLetter.pdf");
ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
stamper = new PdfStamper(reader, baos2);
form = stamper.getAcroFields();
form.renameField("field", "field2");
stamper.close();
```

Then, repeat what you did in section 2.3.2 (applying some small changes):

```
/* chapter02/HelloWorldStampCopyStamp.java */
ByteArrayOutputStream baos = new ByteArrayOutputStream();
PdfCopyFields copy = new PdfCopyFields(baos);
copy.addDocument(new PdfReader(baos1.toByteArray()));
copy.addDocument(new PdfReader(baos2.toByteArray()));
copy.close();
```

Finally, stamp the fields you've just renamed:

```
/* chapter02/HelloWorldStampCopyStamp.java */
reader = new PdfReader(baos.toByteArray());
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldStampCopyStamp.pdf"));
form = stamper.getAcroFields();
form.setField("field1", "World");
form.setField("field2", "People");
stamper.setFormFlattening(true);
stamper.partialFormFlattening("field2");
stamper.close();
```

Notice this line: `stamper.partialFormFlattening("field2");`.

Although you've set flattening to `true`, the resulting PDF still has a form with editable fields. Only the fields you marked with the method `partialFormFlattening()` are flattened. This is useful if the forms are part of a workflow, being filled in by different instances. For example, suppose some parts of a loan-application form are to be filled in by the couple applying for the loan, whereas other parts are to be filled in by the company granting the loan. The form can