Unedited
Draft

iText IN ACTION

Creating and Manipulating PDF

Bruno Lowagie

MANNING

iText in Action

by Bruno Lowagie

Unedited Draft Chapter 1

For more information on this and other Manning titles go to
www.manning.com

1

iText: when and why

If you want to enhance applications with dynamic PDF generation and/or manipulation, you've come to the right place. Throughout this book, you'll learn how to build applications that produce professional, high-quality PDF documents. More specifically, you'll learn how to do the following:

- Serve dynamically generated PDF to a web browser
- Generate documents and reports based on data from an XML file or a database
- Create maps and ebooks, exploiting numerous interactive features available in PDF
- Add bookmarks, page numbers, watermarks, and other features to existing PDF documents
- Split and/or concatenate pages from existing PDF files
- Fill out forms, add digital signatures, and much more

You'll create these documents on the fly, meaning you aren't going to use a desktop application such as Adobe Acrobat. Instead, you'll use an API to produce PDF directly from your own applications, which is necessary when a project has one of the following requirements:

The content needs to be served in a web environment, and PDF is preferred over HTML for better printing quality, for security reasons, or to reduce the file size.

The PDF files can't be produced manually due to the volume (number of pages/documents) or because the content isn't available in advance (it's calculated and/or based on user input).

Documents need to be created in unattended mode (for instance, in a batch process).

The content needs to be customized and/or personalized.

This book is a comprehensive guide to an API that makes all this possible: iText, a free Java-PDF library. For first-time users, this book is indispensable. Although the basic functionality of iText is easy to grasp, this book lowers the learning curve for more advanced functionality.

It's also a must-have for the many developers who are already familiar with iText. With this book, they finally have in one place all the information previously found scattered across the Internet. Even expert developers are likely to discover iText functionality they weren't aware of.

In this chapter, you'll learn how iText was born, and we'll look at some real-world PDF files that were generated using iText.

1.1 The history of iText

In the summer of 1998, the university where I worked[1] was starting up a migration project with the intention of redesigning a series of standalone programs used by the student administration. Up until then, entering the grades of students and calculating their final results at the end of the academic year was done using software that worked only on MS-DOS. Documents produced by this software could be printed on only one type of

[1] ICT Department, Ghent University, Belgium.

printer. This wasn't an ideal way of working, to say the least. Teachers and their administrative staff were using all kinds of systems: Windows, Mac, Linux, Solaris, and so forth. Yet for one of the most delicate aspects of their job—grading students—they were still forced to use plain old DOS. The university decided it was high time to do something about this situation and hired two developers to create a completely web-based solution. One of them was (and still is) my colleague Mario Maccarini. The other one, as you've probably guessed, was me.

Mario and I immediately started writing some Java servlets using Apache JSERV (it was the stone age of J2EE), and we proudly presented our first online lists with students, courses, and grades in the fall of 1998. It was just some ordinary HTML in a browser, but compared to the MS-DOS box, it was a big leap forward. Everybody was enthusiastic, until somebody asked one of the most crucial questions of the project: *what did we, the developers, plan to do about the "document problem"?*

1.1.1 How iText was born

Have you ever tried printing an HTML document in Microsoft Internet Explorer (MSIE), Firefox, or Netscape? If so, you have a good idea of the problem we were facing. Every browser interprets HTML in its own way. A table in MSIE doesn't look completely the same as a table rendered by Firefox. Using Cascading Style Sheets (CSS) can help you fine-tune the end result, but there's another problem: The end-user can disable style sheets, change margins, add page numbers, and so forth. Moreover, just like with Microsoft Word documents, the end user can usually change the content of an HTML document manually, using the application that renders the document. We wanted to avoid this, so we didn't consider Word and HTML to be options. We needed a technology that allowed us to generate unalterable reports with a reliable layout.

I didn't know much about the Portable Document Format back then. I only knew it was supposed to be a read-only format and that you could make printouts look exactly the way you intended to, regardless of the operating system and/or printer. When the document question arose, my answer was impulsive. Without fully realizing the consequences, I told the university committee, "We'll produce PDF!"

Mind you, it was a good answer, and it was well received. PDF is known as a widespread page-description language (PDL), and it's a de facto industry standard. It's portable. It's reliable. It prints really well. Almost everyone has the free Adobe Reader on their system. I assumed all of these fine qualities automatically meant there would be ample free or open source software available to produce PDF.

Apparently I was wrong. I needed an API, a set of classes, preferably written in Java, and preferably open source, but in the winter of 1998, the only free Java-PDF libraries I found on the Internet weren't able to provide the functionality required in our project. Only then did I become aware that I would have to write a PDF library myself if I wanted to keep my promise. During that period, I spent all my free time reading the PDF Reference.

Within seven months of when we were hired, our new intranet application was brought into production at the university where I worked. Its main users were university professors, their proxies, and the administrative staff of the university.

Registered users could log in to a personalized intranet page and do the following:

- Get an overview of all the courses they were responsible for (as a teacher or their proxy)
- Fetch (empty) grading lists in PDF with all the students enrolled for a specific course
- Get an HTML form to submit grades to the server (this could also have been a PDF AcroForm—a form containing a number of fixed areas—or AcroFields, on one or more pages)
- Get a completed version of the grading lists per course

School administrators were also able to

- Compose a curriculum for each individual student
- Generate application forms for students to sign up for specific examination periods
- Calculate every student's grade at the end of the academic year
- Fetch lists with information on the complete year of study for different purposes: deliberation lists, proclamation lists, feedback for the students, and so forth
- Generate official documents such as report cards and transcripts for the students

Every document that needed to be printed was generated in PDF by the newly created library. I designed this set of classes in such a way that it would be usable in other projects, too. I was encouraged to publish the library as a Free and Open Source Software (FOSS) product even before our project went into production. That's how iText was born.

Almost immediately, many fellow developers started to use the library, contributing source code at the same time. Paulo Soares was one of these early adopters. He joined the project in the summer of the year 2000 and is now one of the main developers of new iText features. He also maintains the .NET port iTextSharp.

1.1.2 iText today

Nowadays, iText is used in many online and other services, directly or indirectly. You may have already used iText without being aware of it; a lot of software products ship iText in their distribution. If you've created PDF documents using Macromedia ColdFusion, the file was probably generated by iText. If you're creating reports with one of the most important reporting tools of the moment—JasperReports or Eclipse/BIRT—you'll see that iText is built in as its PDF engine. You could use this book to enhance your own product so that it's capable of producing PDF documents, but the activity on the mailing list tells me it's more likely that you're going to use iText in tailor-made applications similar to the intranet application Mario and I wrote.

In e-commerce applications, you replace students with customers, courses with products, and grades with prices. Energy companies use iText to generate invoices with tables showing customers how much gas, electricity, or water they consumed. The iText library is popular in e-government projects because iText can be used to add a digital signature to a PDF document using an *eID*—a smartcard issued by some governments that can be used for proof of identity. The financial sector uses iText to provide clients with reports about investments, or to produce and process loan application forms. Manufacturers can use iText to compose lists of the parts, subassemblies, and raw materials used to make a product (the Bill of Materials) complete with barcodes that allow automating the manufacturing process. I've seen blueprints and city maps that were created with iText. NASA uses iText in a tool that produces PDF documents showing global longitude-latitude images or pole-to-pole latitude-vertical images of the earth. Google Calendar uses iText to produce calendar sheets.

In short, whatever your project, iText can save you a lot of work and time, helping you to create new PDF documents and/or manipulate existing PDF files.

Ease of use and flexibility

First-time iText users will find lots of examples on the Internet explaining how to create a simple PDF document using iText. On the Java Boutique site is an article by Benoy Jose titled "PDF Generation Made Easy" (http://javaboutique.internet.com/tutorials/iText/). This title reflects the initial idea of iText—that you shouldn't have to be a PDF specialist to be able to generate PDF documents. iText's small set of basic building blocks allows you to create a proof of concept in no time.

Some in the community are occasionally heard to say that working with iText can be demanding, as might be expected of even a well-designed software tool when you're dealing with complicated issues. However, this book is structured so that even iText's complexities are presented painlessly. Don Fluckinger, a freelance writer who has been covering Acrobat and PDF technologies for PDFZone since 2000, writes that iText is "a robust little software tool for generating PDFs on the fly that isn't for the technically faint of heart." I must admit that iText code can get complex as soon as you want maximum flexibility when creating a customized PDF document. Don recommends iText "if you feel like rolling up your sleeves, popping open the hood, and getting to work." That's exactly what we're going to do in this book: We're going to go further than the articles you can find on the Internet and in the online tutorial. This book will give you an in-depth overview of what is possible with iText.

A developer who successfully integrated iText into his software writes, "You're able to produce an extremely size-optimized PDF on-the-fly without sacrificing any feature of the desired output." That's the spirit of the true iText user.

iText licensing

Although iText is free (you're allowed to use iText in open or closed source software, in standalone or web-based applications, for free or proprietary services, and in commercial or nonprofit projects), this doesn't mean you're free to do anything you want with the library; you have to respect the copyright and the Mozilla Public License (MPL) that protects iText. The first versions of iText were published under the Library (or Lesser) GNU Public License (LGPL), but once iText got interesting for some major players in the Information and Communications Technology (ICT) business, there was increasing pressure to move to another license.

Many company lawyers had issues with some of the quirky details in the LGPL, so we chose the MPL with LGPL as an alternative license, for backward compatibility. Basically, the MPL says that you have to inform your customers that you're using the FOSS library iText (by Bruno Lowagie and Paulo Soares), and you have to tell them where they can find the library's source code. Additionally, if you change the library, you should make your enhancements and bug fixes available to the community. This leads to a win-win situation: You win if you get your fixes in the official release, because you reduce upgrade-related problems. The iText community wins because it can benefit from your enhancements. This is the short explanation. For the long version, see the full text of the MPL that is available on the iText site (http://www.lowagie.com/iText/MPL-1.1.txt) and packaged with the source code.

1.1.3 Beyond Java

This book focuses on PDF manipulation with iText seen from a Java developer's point of view, but that doesn't mean you can't use iText in another environment. Companies make choices, and when it comes to building enterprise software, it seems to come down to a choice between two technologies: J2EE or .NET. That's why the .NET ports are religiously synchronized at the release and Concurrent Versioning System (CVS) level.

iText.NET and iTextSharp

There are two important .NET ports: *iText.NET* is a J# port by Kazuya Ujihara; and *iTextSharp* is a C# port originally written by Gerald Henson, but which has been taken over by Paulo Soares, the most active developer of iText in the past five years. Paulo has been "converted" from Java to .NET recently and keeps iTextSharp synchronized with the original Java version.

iText and pdftk

The PDF Toolkit (pdftk) by Sid Stewart is "a command-line tool for doing everyday things with PDF documents," as defined on the AccessPDF web site (www.accesspdf.com). pdftk is also a good example of how iText can be used in a C++ program by building a native library using the GNU compiler for Javagcj. If your program needs some of the PDF-manipulation functionality found in a C++ environment, you should try this toolkit.

iText and ColdFusion

The iText.jar file is shipped with Macromedia's server product ColdFusion. This means it's possible to use iText in your ColdFusion applications for generating PDF documents on the fly. By acquiring Macromedia, Adobe now has an affordable server product that is able to produce PDFs.

Using iText in PHP, Python

There are PHP or Python ports, but in a PHP application, you can use a PHP/Java bridge for PHP-Java integration. If you search the Internet, you'll find some iText examples written in Jython, the Java implementation of Python.

You won't find any VB, C#, J#, CF, PHP, or Jython examples in this book, but it should be fairly easy to adapt the Java examples so that you can use them in your specific development environment. Most of the mechanisms that are explained in this book are independent of the programming language. Let's return to Java and find out how to download and test iText.

1.2 iText: first contact

Setting up an environment in which to run and test the examples in a book can be cumbersome, especially if you need to install additional services or servers. To reduce the complexity, most examples in this book were conceived as small standalone applications.

All examples were written in Java, so you'll need a Java environment (JDK 1.4 or higher is preferred) and the appropriate Java Archives (jars). Each example writes a short explanation to the `System.out`, telling you what it does. It also lists the necessary resources and the jars needed in the `CLASSPATH` (a variable that tells the Java Compiler and JVM where to find all necessary Java class-files and archives).

iText.jar is an executable jar. If you open it in a Java Runtime Environment (JRE), the iText toolbox opens. This is a GUI application that lets you do some simple PDF experiments without having to write a single line of code.

But first things first: Let's find out how to compile and execute the code samples.

1.2.1 Running the examples in the book

You can download a Zip file containing all the examples in this book from http://itext.ugent.be/itext-in-action/. Unzip this file in the directory of your choice, but be sure to name it something you can easily remember. After unzipping the file, you should have a subdirectory called /examples. The examples are organized in packages by chapter.

The code snippets in this book all start with a comment line, indicating where to find the corresponding example by giving you a subdirectory called <*your_dir*>/examples/ and the name of the Java source file. If an example needs some extra resources (such as an image or an XML file), you'll find them in a subdirectory: <*your_dir*>/examples/chapter<*chapter_number*>/resources.

Whenever extra fonts are needed (TTF, OTF, or TTCfiles, for example), they should be in the directory C:/Windows/Fonts. You'll need to adapt this hardcoded path in the example if you're working on a Mac, Linux, or Unix OS, or if the fonts are stored elsewhere on your Windows system.

NOTE

Never use hardcoded paths in your production code. I wanted the examples to be simple, so I didn't use code to load properties files or fetch information from a Java Naming and Directory Interface (JNDI) repository. You should use a more robust solution to refer to fonts or any other resource once you start writing your own code.

You'll also need to download a file containing all the Java archives that are needed to run the examples. The Zip file with the examples comes with a build.xml file that expects these jars to be present in the directory called <*your_dir*>/bin. If you're used to working with ANT—the standard tool used to build and execute Java code—you'll immediately feel comfortable with it.

The `action` target allows you to compile and execute each example like this:

```
$ ant -Dchapter=01 -Dexample=HelloWorld action
```

Although this is the official way to run `ant`, with the target at the end of the command, I find it more practical to switch the order of parameters and target like this:

```
$ ant action -Dchapter=01 -Dexample=HelloWorld
```

It saves you a few keystrokes to use the Up arrow to repeat and the Backspace key to change a command previously called in your shell (such as DOS or bash). This particular command compiles and executes a "Hello, World" example. The source code can be found in the directory <*your_dir*>/examples/chapter01/HelloWorld.java. This Java source file is compiled to <*your_dir*>/bin/classes/chapter01/HelloWorld.class, and the file HelloWorld.pdf appears in <*your_dir*>/examples/chapter01/results as soon as the compiled code is executed.

After a while, you'll have generated lots of files—compiled Java classes, PDF documents, and so forth. You can remove all these files at once by using the `clean` target for the `ant` command.

Once you succeed in running these examples, integrating iText into your own application should be a piece of cake. Just add the iText.jar to your `CLASSPATH`, and start coding. If you're new to Java development, and you have trouble finding where to put the jar or where to change the `CLASSPATH` in a web application, please consult your application server's manual.

If you're not ready to compile and execute these examples yet, you can turn to the iText toolbox first. This toolbox offers some ready-to-use tools that don't require any knowledge of Java or PDF; you only need a JRE.

1.2.2 Experimenting with the iText toolbox

Originally, iText was developed as a developer's library, meaning that it wasn't aimed at an end-user market. Developers could integrate iText into their Java web applications or standalone Java programs, but the library itself didn't have a user interface.

When the first PDF manipulation classes were added to iText, some simple command-line applications for splitting, encrypting, and concatenating PDF files were provided as examples in the iText tutorial. Later, these sample applications were moved to a `com.lowagie.tools` package.

Mailing-list questions made it clear that not many people were using command-line tools, probably because they aren't user friendly. So, a small GUI called the iText toolbox was developed. The toolbox has now become a means to test part of the iText functionality without having to write any source code.

You can open the toolbox by executing the iText jar file:

```
java -jar iText.jar
```

In figure 1.1, some plug-ins are opened in an internal window of the toolbox.

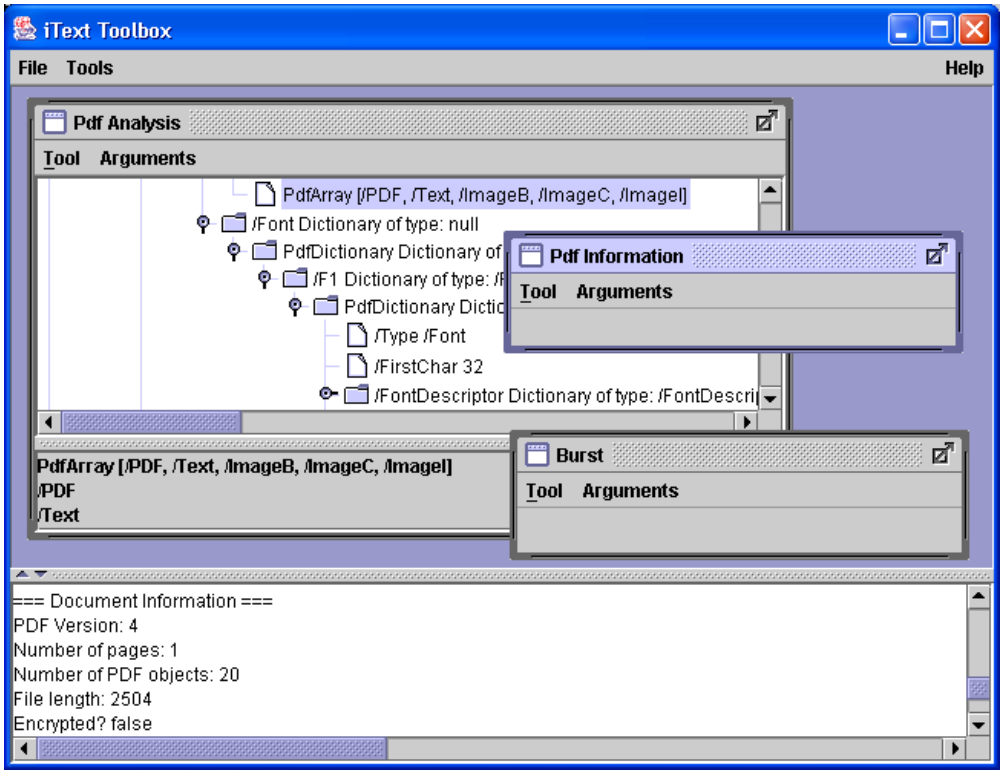


Figure 1.1 The iText toolbox

The toolbox contains three menu items:

- *File*—The File > Close command closes the toolbox.
- *Tools*—A selection of plug-ins is loaded from the package `com.lowagie.tools.plugins` when you open the toolbox. These plug-ins are organized in different categories under the Tools menu.
- Help—Choosing Help > About directs you to a web page describing the tools, and Help > Version shows the list of tools that were loaded and their versions.

NOTE

By going to the URL http://www.lowagie.com/iText/itext.jnlp, you can use the Java Network Launching Protocol (JNLP) to download and start the jar as a Java Web Start (JWS) application. The application should start automatically. Notice that you'll get a security warning because I signed the jar with a self-signed certificate.

Most of the plug-ins are self-explanatory. In the chapters that follow, we'll dig into the mechanics of some of these tools. Whenever there's a toolbox tool that illustrates some specific functionality, I'll insert a note about it like this:

NOTE

iText toolbox: `com.lowagie.tools.plugins.Burst` (category: Manipulate)

The verb *to burst* has different meanings. One of its meanings is "to divide paper; to separate continuous stationery such as computer printout into individual sheets." In the context of electronic paper, to *burst* a PDF means splitting it into single pages.

For instance, using the Burst plug-in on a three-page file named HelloWorld.pdf generates three separate files—HelloWorld_1.pdf, HelloWorld_2.pdf, and HelloWorld_3.pdf—each containing a single page of the original document, to which the number after the underscore corresponds.

Each plug-in can be used in three different ways:

From an internal window in the toolbox—You can fill in the parameters for the tool (source file, destination file, and so on) by choosing Arguments in the internal window's menu. By clicking Tool, you can ask the tool for its Usage, consult the Arguments, and Execute the tool. Other (optional) menu items are Execute > Open and Execute > Printdialog. There's always a Close item to close the window.

As a command-line tool—For instance, if you want to burst a PDF file from the command line, you can call the plug-in like this:

```
$ java -cp ./iText.jar com.lowagie.tools.plugins.Burst HelloWorld.pdf
```

Calling the plug-in without any arguments will show you the *Usage* information.

From another Java application—Construct a `String` array with the arguments and call the main method of the plug-in:

```
/* chapter01/HelloWorldBurst.java */
String[] arg = {"HelloWorldRead.pdf"};
com.lowagie.tools.plugins.Burst.main(arg);
```

We'll create some more HelloWorld PDF files in the next chapter to get acquainted with iText. First, let's look at the more interesting examples this book has in store. Let me tell you a story that could have happened to you.

1.3 An almost-true story

I'm not really a programmer. I studied civil architectural engineering, and I started my professional career in the Geographical Information Systems (GIS.) division of Tractebel Information Systems, in Brussels, Belgium, which is now owned by the international industrial and services group Suez. While I was looking for an application that could run continuously throughout this book, I started drawing the map of a fictional city called Foobar. On this map, I added a university campus. That way, I combined my GIS background with my current professional situation. I thought of a story that would make an employee of the fictive Technological University of Foobar (TUF) the heroine. Her name is Laura, and she will be your guide throughout the longer examples in this book.

The following subsections tell the beginning of Laura's story, but their main purpose is to give you a preview of the iText features that will be explained in parts 2, 3, and 4. Starting with chapter 2, you'll find lots of small, almost atomic source code examples that explain how to do something; later, some longer real-world examples will show you how it all works together. The screenshots in this section represent the output of these longer examples.

1.3.1 Some Foobar fiction

Laura is preparing to attend yet another staff meeting. According to her business card, she's a software architect for the central administration at TUF. When asked for her job title, Laura prefers to call herself a Java developer, plain and simple.

TUF is a small university located in the city of Foobar. Apart from the central administration, it consists of only two departments: the Department of Science and the Department of Engineering. There has been a constant rivalry between the departments, one of the catalysts being the introduction of computer science as a new study discipline. That was over 20 years ago. At that time, the board of the university decided to follow in the footsteps of King Solomon and divided the discipline over both departments. Undergraduates had to enroll in the Department of Science, whereas graduate students enrolled in the Department of Engineering.

It was a great idea in theory, but in practice, it was a burden. Making decisions concerning the educational program of the complete field of study was no longer a sinecure. Hidden agendas and internal differences between the departments often got in the way of good management. Informatics students suffered from this pragmatic division, too—their colleagues from other scientific disciplines didn't consider them to be "real" scientists in the first years of their studies, and during their graduate years, their peers didn't regard them as being "engineer material."

Laura was aware of the feeling, but she was always careful never to be dragged into a discussion about it. For a long time, the university played with the idea of redesigning all the software applications supporting the core business processes of the central administration. Finally, a decision was made, and a committee was formed with authorities from both departments. Laura, of course, was also invited. She feared the worst and decided to keep quiet while the debates between scientists and engineers heated up. At one point, she forgot where she was and began to daydream.

1.3.2 A document daydream

Computer sciences, software engineering, Information and Communication Technology (ICT)—all of these disciplines have their differences, but is dividing really the best way to conquer the hearts of students? Laura had given this question a lot of thought. *Suppose I were given the opportunity to start a new department,"* she said to herself, *"a department that combined all the courses and education in the field of computer science and engineering. What would I need?*

She decided to start with the following:

Promotional flyers for the new department

A guide containing study programs (tables)

A course catalog (columns)

In part 2 of this book, all the elements needed to bring these assignments to completion will be explained step by step throughout four chapters. At the end of most chapters, you'll work with Laura to create the documents she's dreaming of.

Making a flyer

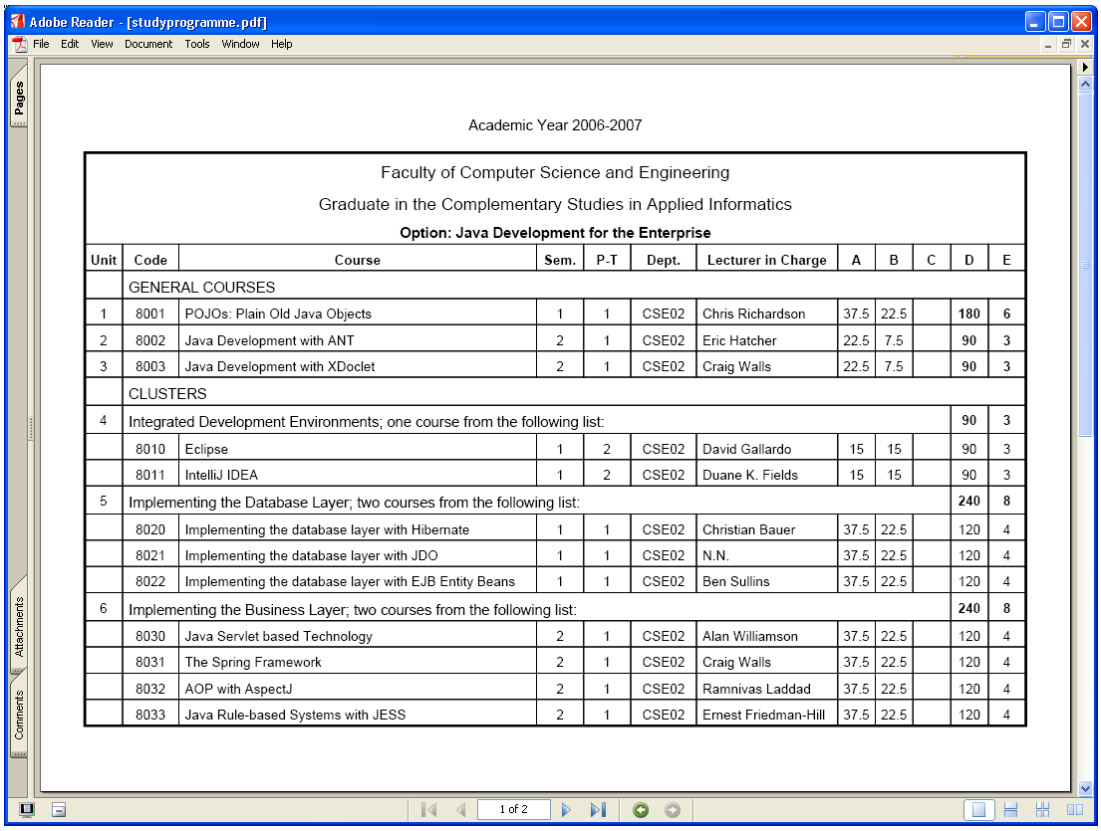
As Laura's new colleagues, the first thing we'll do is create a flyer with the university's logo, a paragraph welcoming new students, lists of programs offered by the department, and links to the university's web site. See figure 1.2 for an example.

Figure 1.2 A PDF document containing some basic text elements, such as paragraphs, lists, anchors, and images

You can consult section 4.3 if you need to generate a flyer with paragraphs, lists, and anchors. If you need images, you'll also need to read section 5.3. These sections explain how to write source code that allows you to create an exact copy the PDF in figure 1.2.

Composing a study guide

Once students have seen our flyer, they may be interested in studying at the Department of Computer Science and Engineering. If they contact the university for more information, we should be able to send them a study guide. One part of the study guide should contain tables representing the study programs. Figure 1.3 shows the first page of the program for students who want to earn a graduate degree in complementary studies in applied informatics.



Academic Year 2006-2007

Faculty of Computer Science and Engineering

Graduate in the Complementary Studies in Applied Informatics

Option: Java Development for the Enterprise

Unit	Code	Course	Sem.	P-T	Dept.	Lecturer in Charge	A	B	C	D	E
	GENERAL COURSES										
1	8001	POJOs: Plain Old Java Objects	1	1	CSE02	Chris Richardson	37.5	22.5		**180**	**6**
2	8002	Java Development with ANT	2	1	CSE02	Eric Hatcher	22.5	7.5		**90**	**3**
3	8003	Java Development with XDoclet	2	1	CSE02	Craig Walls	22.5	7.5		**90**	**3**
	CLUSTERS										
4	Integrated Development Environments; one course from the following list:									**90**	**3**
	8010	Eclipse	1	2	CSE02	David Gallardo	15	15		90	3
	8011	IntelliJ IDEA	1	2	CSE02	Duane K. Fields	15	15		90	3
5	Implementing the Database Layer; two courses from the following list:									**240**	**8**
	8020	Implementing the database layer with Hibernate	1	1	CSE02	Christian Bauer	37.5	22.5		120	4
	8021	Implementing the database layer with JDO	1	1	CSE02	N.N.	37.5	22.5		120	4
	8022	Implementing the database layer with EJB Entity Beans	1	1	CSE02	Ben Sullins	37.5	22.5		120	4
6	Implementing the Business Layer; two courses from the following list:									**240**	**8**
	8030	Java Servlet based Technology	2	1	CSE02	Alan Williamson	37.5	22.5		120	4
	8031	The Spring Framework	2	1	CSE02	Craig Walls	37.5	22.5		120	4
	8032	AOP with AspectJ	2	1	CSE02	Ramnivas Laddad	37.5	22.5		120	4
	8033	Java Rule-based Systems with JESS	2	1	CSE02	Ernest Friedman-Hill	37.5	22.5		120	4

Figure 1.3 A PDF document containing basic text elements, organized in tables

The second part of the study guide should describe the courses that are mentioned in the study program. Figure 1.4 shows how we could organize this information in columns with tables and images.

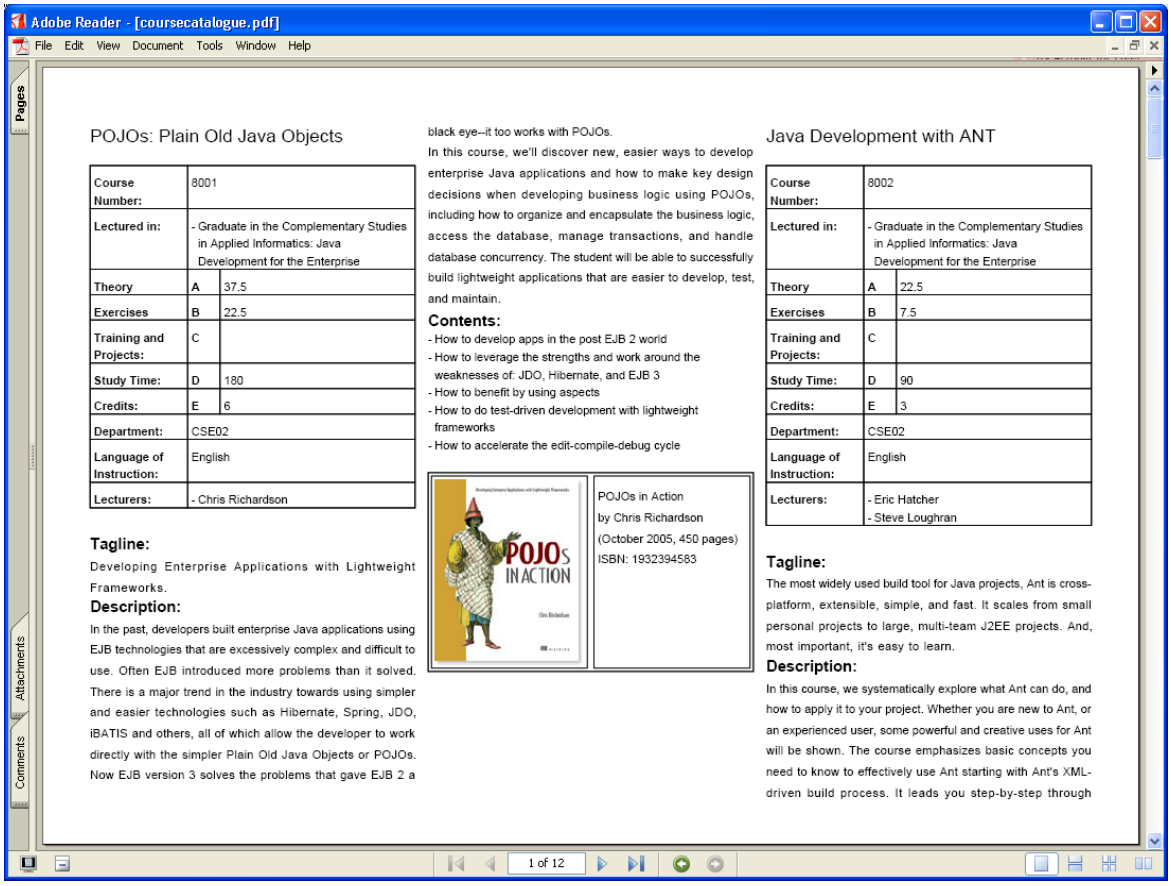


Figure 1.4 A PDF document containing basic text elements, organized in columns

Chances are, you've been working on projects that deal with similar information. Maybe you've been asked to publish content coming from a database or an XML repository in the form of some neat-looking PDF reports.

If that is the case, you may want to read chapters 6 and 7 and discover how to shape your data into tabular or columnar text elements. The code that was used to create figure 1.3 and figure 1.4 is discussed in sections 6.3 and 7.5.

1.3.3 Welcoming the student

The university will welcome students from all over the world, so it's important that we provide them with an information package with some information written in different languages. We'll also have to give them a map of the city so that they're able to find their way to the campus. The six chapters of part 3 deal with PDF text and graphics, which we'll need to produce documents using different fonts and writing systems, and a map of the city of Foobar.

Whereas part 2 discusses mainly iText-specific functionality, part 3 goes to the core of iText and focuses on the internal structure of a PDF page.

Producing documents in different languages

In the ICT world, developers have adopted the English language as the de facto standard for human communication. That's why I'm writing this book in English, although my mother tongue is Dutch. At some point, however, you may be asked to create documents with non-English text. You probably won't have a problem displaying text in French, even with all those little accents and cedillas; those characters can be found

in the standard *latin-1* encoding. But to display some special characters that are common in languages such as Polish or Turkish, you have to use another encoding . The same goes for Greek and Russian, languages that have completely different alphabets than English.

It gets harder when you need to display text in an Asian alphabet, because such alphabets use many different symbols or ideograms organized into many different character sets. Another issue arises: In general, Asian languages can be written from left to right, but it's also common to write text in vertical columns read from top to bottom and right to left. Producing electronic documents using such a writing system can be complex using standard software. The same goes for Semitic languages, such as Arabic and Hebrew , which have scripts that are written from right to left.

This is the problem Laura is facing. Foobar is a small city in a small country. In order to be a successful university, TUF invites students from all over the world. Laura isn't multilingual, but she has found a web site with the translation of the word *peace* in a few hundred languages. To prove that we can generate a welcoming document in different languages, we'll help Laura display these words of peace.

Figure 1.5 shows a document with a message of peace in English, Arabic, and Hebrew, respectively. Even if you can't read Arabic or Hebrew, you can see these languages are written from right to left by looking at the position of the exclamation point and the comma. The order of the numbers and Latin characters in the abbreviation for Internet Internationalization (I18N) is preserved.

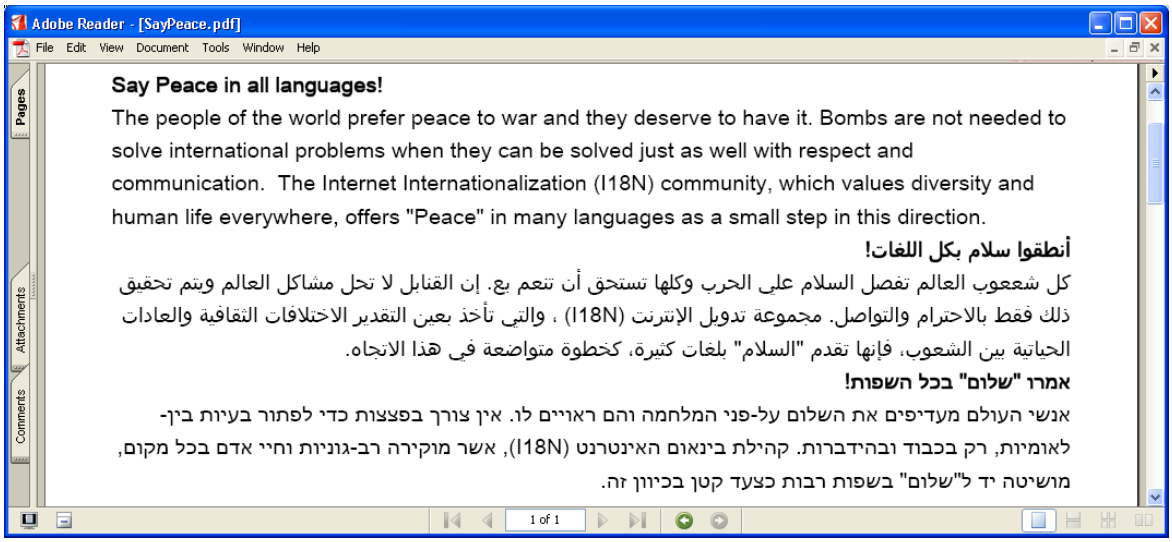


Figure 1.5 A PDF document demonstrating different writing systems

If you need support for special character sets, encodings, or writing systems, you'll find chapters 8 and 9 indispensable.

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.

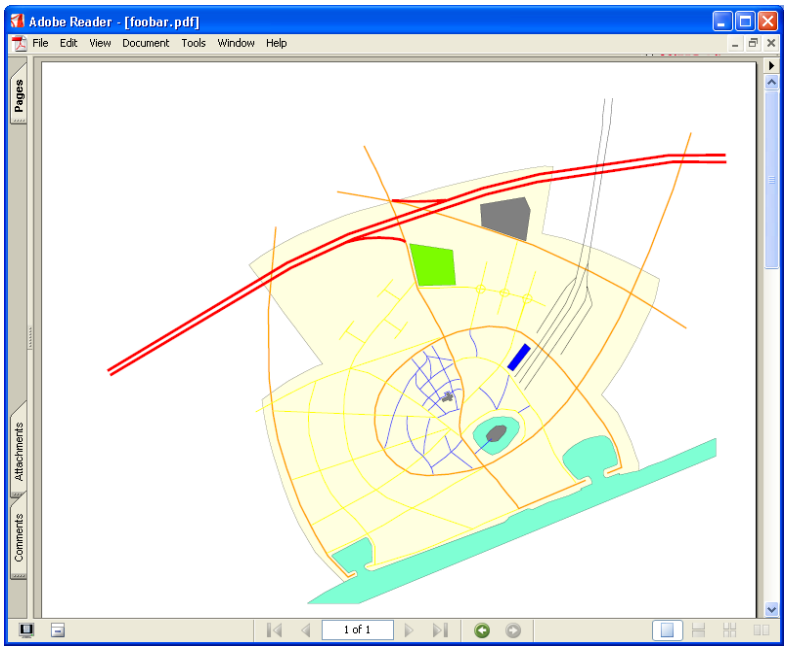


Figure 1.6Using iText to draw graphics such as lines and shapes

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).

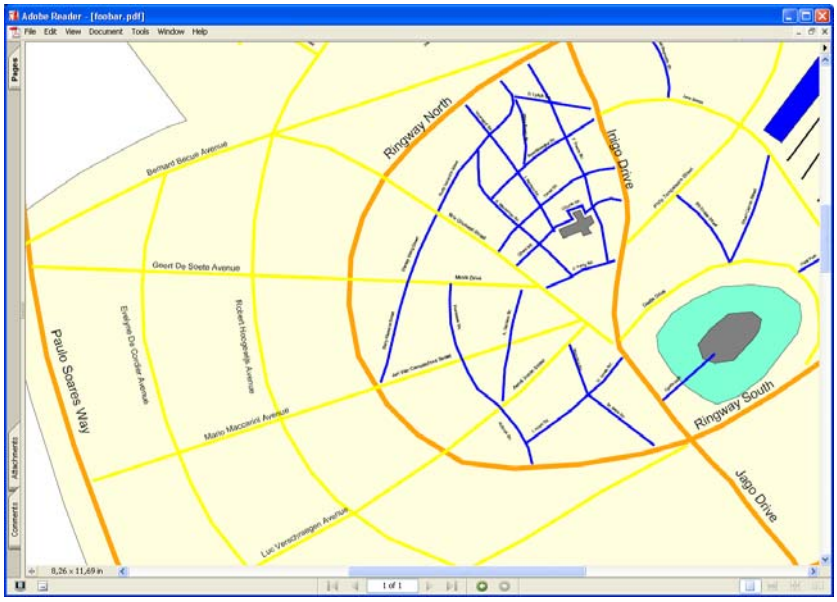


Figure 1.7 Using iText to draw text at absolute positions

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

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.

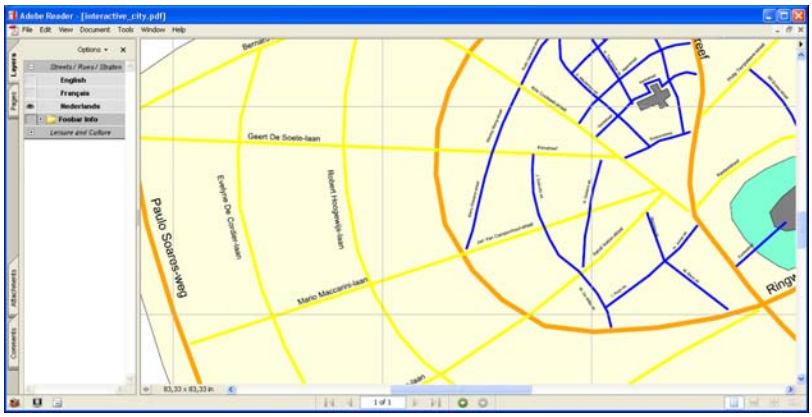


Figure 1.8 A PDF document demonstrating the use of optional content groups.

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.

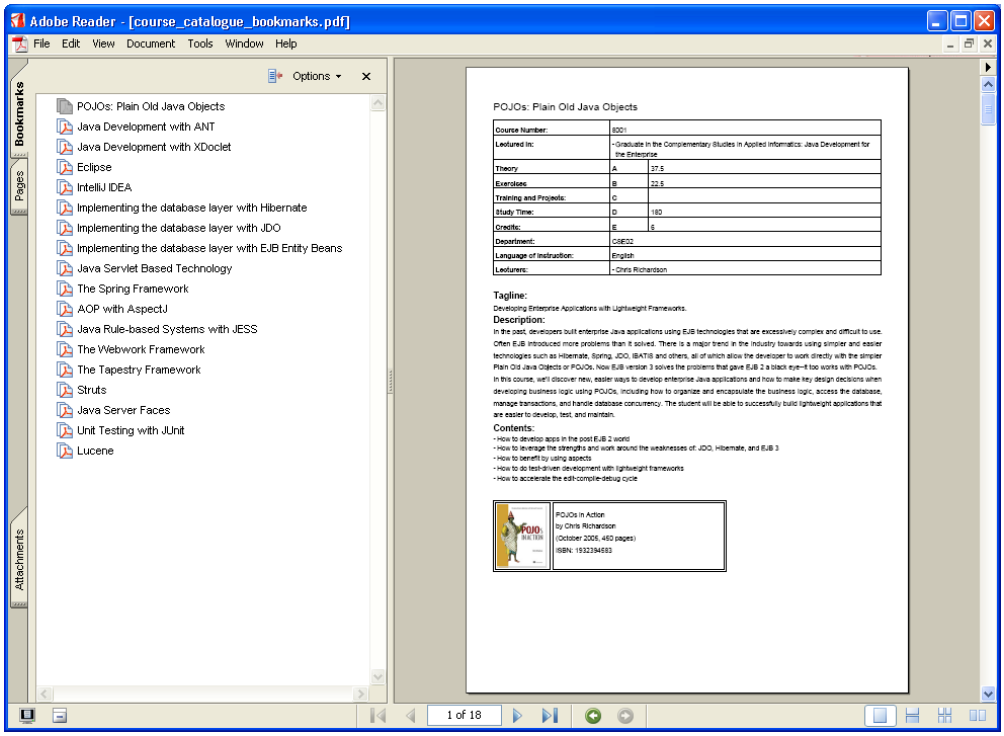


Figure 1.9 A PDF document demonstrating some interactive features.

Adding watermarks and page numbers

Figure 1.10 shows pages 10 and 11 of the course catalog. The course number has been added as a header, and every file has the university's logo as its watermark.

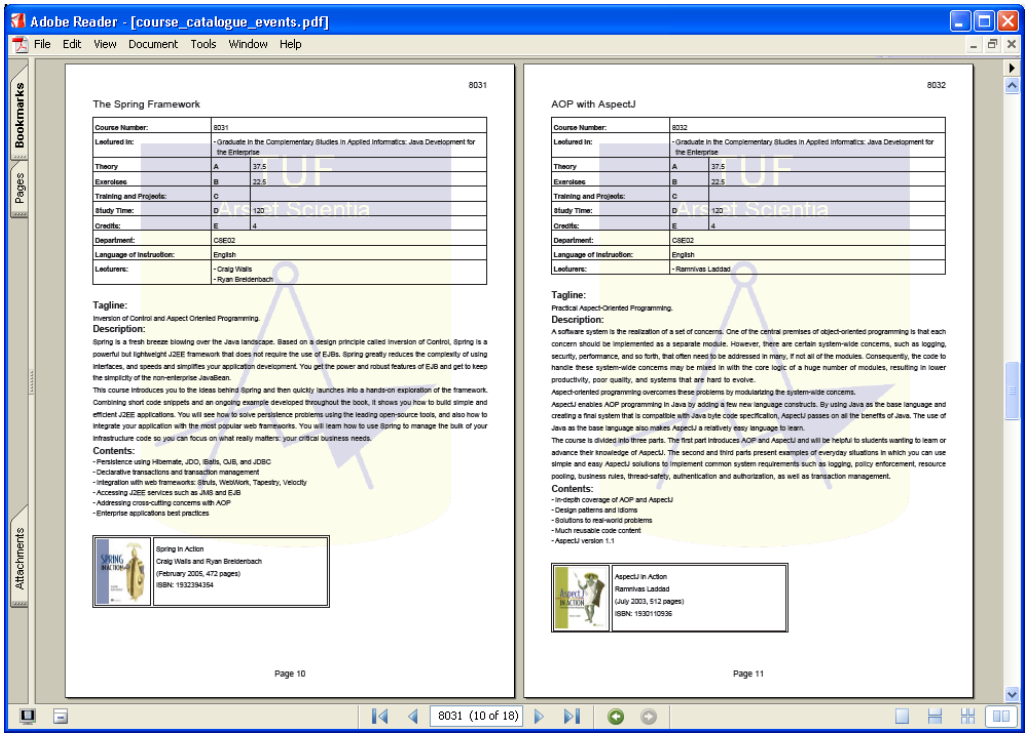


Figure 1.10 Using page events, among others, to add page numbers and watermarks

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.

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.

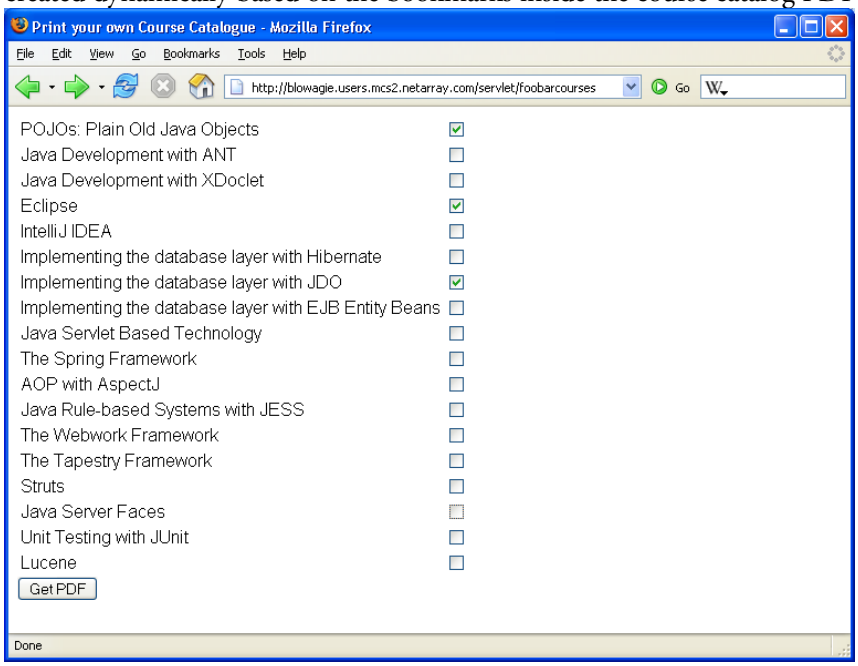


Figure 1.11 An HTML form listing the different courses in the course catalog

Student 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 page labels.

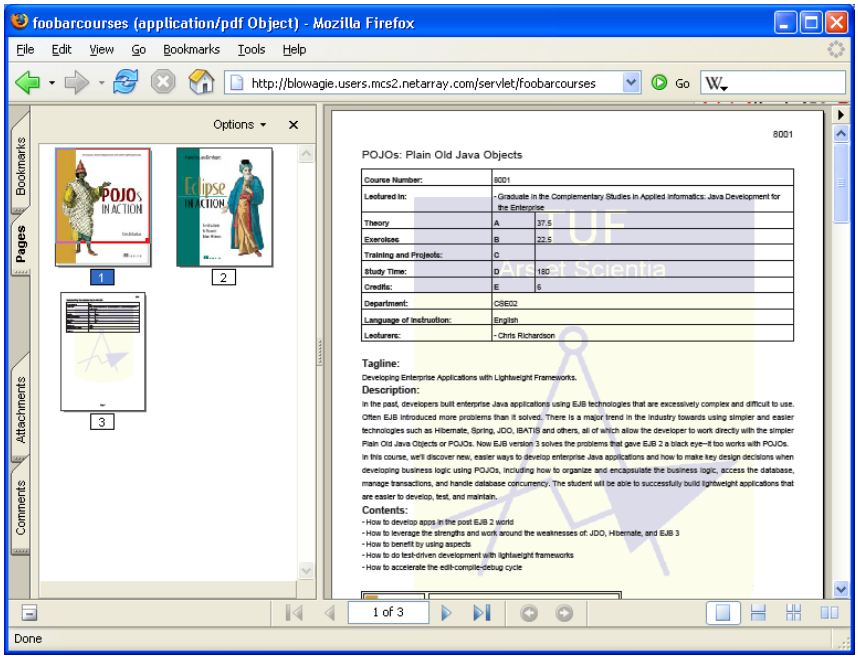


Figure 1.12A PDF served by a web application containing a personalized course catalog

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 figure 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.

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.

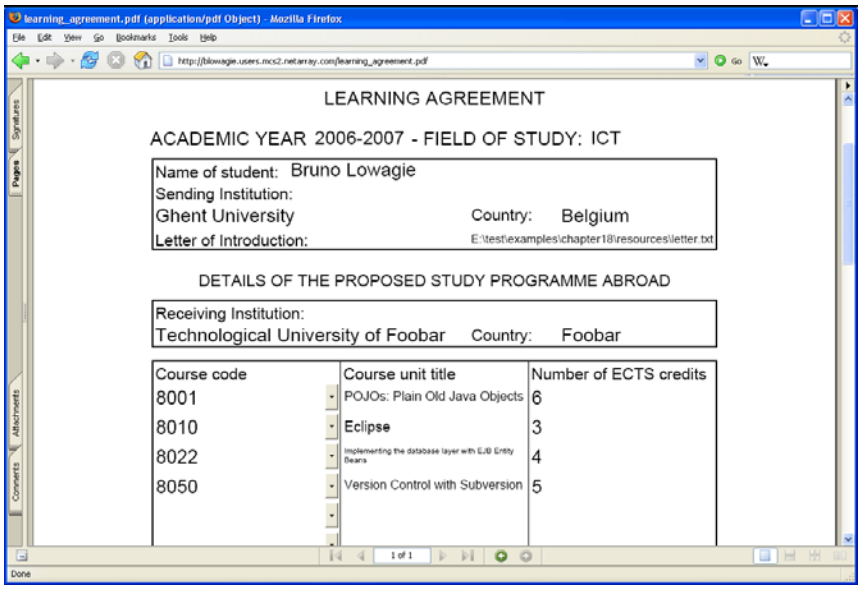


Figure 1.13 A PDF form in a browser

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.

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.

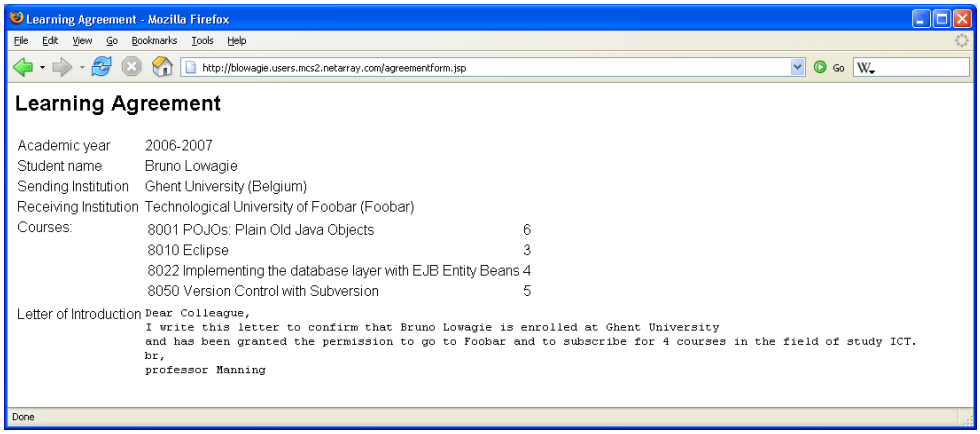


Figure 1.14Displaying the data that was submitted using a PDF AcroForm

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' 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?