brief contents

3 PDF: why and when

This chapter covers

- What is PDF?
- History of the format
- Different types
- Different versions

In chapter 2, you created some simple and some not-so-simple "Hello World" documents. The not-so-simple documents have an initial demonstration of the power of iText as far as document manipulation is concerned. Before we continue with iText at full force, we'll take one step back and look more closely at the Portable Document Format.

In the first section of this chapter, you'll learn why PDF was invented and how it evolved into a de facto standard. In the second section, you'll see that PDF comes in different flavors, some of which are described in an International Standards Organization (ISO) standard. It's important to understand when to choose which specific type of PDF.

Finally, we'll use a table listing the different versions of the PDF specification to focus on specific features such as compression and encryption. We'll conclude with more "Hello World" examples that show how to compress/decompress and encrypt/decrypt PDF files.

3.1 A document history

Do you remember when people were talking about the paperless office? It was a utopian concept that surfaced in the 1980s, which didn't make it to the end of the century. The brave new technology that was going to eliminate the paper chase had quite the opposite effect—it generated an avalanche of paper.

Although electronic documents didn't bring about utopia, they do have advantages:

- *They're easy to search*—Even if electronic documents don't have an index, there are tools that can make one for you automatically.
- *They're easy to archive*—Just think of the huge amount of cubic meters needed for paper storage and compare that to the number of electronic documents you can save on a mass-storage device.
- *They're easy to exchange*—You can put electronic documents on a web site or e-mail them if you want to share them with others.

Of course, there are also major downsides. The fact that electronic documents are easy to exchange can be a serious disadvantage when it comes to issues of piracy and illegal copies. When it comes to legal issues, a hard copy still holds more credibility than an electronic one. Even more important, there's the irrefutable fact that a printed document is a lot easier to read than text on a computer screen. As it turns out, paper still rules.

New technologies are emerging that may revive the dream of the paperless office. New devices that provide a better reading experience are finding their way to the market. Technologies that add digital signatures to an electronic document are becoming increasingly accepted by companies and governments. Electronic documents are becoming more reliable and more secure. One of the key protagonists in this process, if not the main player, is Adobe Systems Incorporated. In this section, we'll look at the company and its products, and we'll talk about the intellectual property of the PDF specification.

3.1.1 Adobe and documents

Adobe Systems Incorporated was founded in 1982 by John Warnock and Chuck Geschke. Its first products were digital fonts. These days, Adobe Creative Suite (including Photoshop and Illustrator) and Acrobat are the company's flagship products.

It's important to realize that PDF wasn't created out of the blue. The ancestors of PDF still exist and are used in many applications. The best way to understand the difference between PDF and these other specifications is to go back in history and see how it all started.

The ancestors of PDF

In 1985, Adobe introduced the PostScript (PS) Page Description Language (PDL). PS is an interpretive programming language. Its primary goal is to describe the appearance of text, graphical shapes, and sampled images. It also provides a framework for controlling printing devices; for example, specifying the number of copies to be printed, activate duplex printing, and so forth.

Also in 1985, Adobe developed an application for the Apple Macintosh called Adobe Illustrator, a vector-based drawing program with its own format, AI, which was derived from PS. Illustrator was ported to Windows in 1989, so it covered an important market in the graphical industry.

Producing high-quality visual materials was the privilege of specialists for a long time, but with the advent of PostScript and Illustrator, anyone with a computer could accomplish high-end document publishing. By introducing these two technologies, Adobe started the desktop publishing revolution. But the founders of Adobe felt there was something missing.

In 1991, John Warnock wrote the "Camelot paper," in which he said the following:

> The specific problem is that most programs print to a wide range of printers, but there is no universal way to communicate and view this printed information electronically. … What industries badly need is a universal way to communicate documents across a wide variety of machine configurations, operating systems, and communication networks.

As a result of this writing, a new development project was started, and the engineers at Adobe enhanced the PostScript and Illustrator technologies to create a document format and a suite of applications with which to create and visualize documents of this format.

The Portable Document Format

This new document format, originally called Interchange PostScript (IPS), is now known as the Portable Document Format (PDF). Although PostScript (PS) and PDF are related, they're essentially different formats. PDF isn't a programming language like PS; PDF leverages the ability of the PS language to render complex text and graphics and brings this feature to the screen as well as to the printer. As stated in the PDF Reference, "PDF trades reduced flexibility for improved efficiency and predictability."

PDF and PS share the same underlying Adobe imaging model. A PDF document consists of a sequence of pages, with each page including the text, font specifications, margins, layout, graphical elements, and background and text colors. Unlike PS, PDF can contain a lot of document structure, links, and other related information. As opposed to PS, PDF can't tell the printer to use a certain input tray, change the resolution, or use any other hardware-specific feature. One of the key advantages PDF has over PS is page independence. Because PS is a programming language, something in the description of page 1 can affect page 1000, so to view page 1000 you have to interpret all the pages before it. Each page in PDF can be drawn individually.

PDF is called the *Portable* Document Format because a PDF document can be viewed and printed on any platform: UNIX, Macintosh, Windows, Linux, or Palm OS. In theory, a PDF document looks the same on any of these platforms (we'll discuss some exceptions in chapter 8, when we're talking about embedding fonts). In analogy with Java's Write Once, Run Anywhere, you could say PDF is Write Once, Read Anywhere—but in a more reliable way than the catchy Java advertising phrase promises.

Camelot was the original code name for what later became Acrobat. It's important not to confuse PDF, the Page Description Language, with Acrobat, the suite of Adobe products that was developed along with the PDF specification.

3.1.2 The Acrobat family

The Adobe web site describes the Acrobat family as a suite of products that allow you to "create and exchange documents, collect and compare comments, and tailor the security of a file in order to distribute reliable and polished Adobe PDF documents."

In this book, I assume you and the end users of the PDF files you're producing have Adobe Reader—a free PDF viewer that works with a plethora of operating systems—installed. You can use it as a standalone product or as a plug-in for your browser. It allows you to view, print, and search PDF files. It doesn't let you create or change PDF files. People often confuse Adobe Reader with Acrobat—for example, thinking that the free reader is capable of saving data entered into any PDF form. (That's only possible with reader-enabled PDFs.)

Non-Adobe alternatives for Adobe Reader are available, such as Preview, Ghostview, and Foxit, but these viewers are less feature-rich than Adobe Reader. Note that Mac OS X uses PDF as the basis of its imaging model and ships Preview as the default application for any PDF. Most of the PDF examples generated in this book will be displayed correctly in the other tools, but not all of the functionalities will work. For example, a PDF form is rendered correctly in Apple's Preview, but Preview doesn't know how to submit forms. (I don't know if they plan to add this functionality.)

Even if you're only planning to develop applications using iText, you may need some other Adobe products. For example, a customer may want to design a PDF that can be used as a resource in her software applications. This resource can act as a template that will be manipulated using iText code (see section 2.2). Note that designing a document usually isn't the task of a developer; it's typically a job for a graphic designer using one of the following Acrobat products:

- *Adobe Acrobat Elements* allows you to view, print, and search PDF files, as well as create PDF files from any application that prints. You can manage specialized content from Microsoft Office and protect documents with passwords, granting or revoking permissions. If you're creating PDF files from Microsoft Word, you can use iText to post process and concatenate these files.
- *Adobe Acrobat Standard* has the same functionality as Adobe Elements, but it can also organize comments from multiple reviewers with sorting and filtering tools; combine application files into a single Adobe PDF document; digitally sign and certify documents; and manage specialized content from Microsoft Outlook, MS Internet Explorer, Access, and Publisher.

- *Adobe Acrobat Professional* adds the following features to those of Adobe Standard: enables anyone with free Adobe Reader software to use highlighter, sticky note, pen, and other commenting tools; and builds intelligent forms with Adobe LiveCycle Designer, which is a separate product (that can be executed from Acrobat). For the moment, iText doesn't fully support forms created with Adobe LiveCycle Designer; only static XFA forms. To be sure your forms can be filled with iText, you can create AcroForms (not XFA forms) with Acrobat Professional (not Designer).
- *Adobe LiveCycle Designer* retains layers and object data in technical drawings and manages specialized content from AutoCAD, Microsoft Visio, and Microsoft Project.
- *Adobe Distiller* lets you turn PostScript into PDF.
- *Acrobat Capture* is a powerful Optical Character Recognition (OCR) tool that teams with your scanner to convert volumes of paper documents into searchable PDF files.

These are all commercial products (proprietary software). If you want to use them, you need to purchase them and pay a license fee. Depending on the tool you need, this can be expensive. You may wonder: If Acrobat tools are expensive, how is it possible that everybody can use iText for free? How were the iText developers able to create their PDF-producing software? Did they have to pay a license fee? No, they didn't, and the following explains why not.

3.1.3 The intellectual property of the PDF specification

Adobe owns the copyright for the PDF specifications, but to promote the use of the Portable Document Format for information interchange among diverse products and applications—including, but not necessarily limited to, Acrobat products—Adobe gives anyone copyright permission to (I quote section 1.5 of the PDF Reference, version 1.6):

- Prepare files whose content conforms to the Portable Document Format
- Write drivers and applications that produce output represented in the Portable Document Format
- Write software that accepts input in the form of the Portable Document Format and displays, prints or otherwise interprets the contents
- Copy Adobe's copyrighted list of data structures and operators, as well as the example code and PostScript language-function definitions in the

written specification, to the extent necessary to use the Portable Document Format for the purposes above

The conditions of such copyright permissions are:

- Authors of software that accepts input in the form of the Portable Document Format must make reasonable efforts to ensure that the software they create respects the access permissions and permissions controls listed in Table 3.20 of this specification (i.e. the PDF Reference), to the extent that they're used in any particular document. These access permissions express the rights that the document's author has granted to the users of the document. It's the responsibility of Portable Document Format consumer software to respect the author's intent.
- Anyone who uses the copyrighted list of data structures and operators, as stated above, must include an appropriate copyright notice.

Again, these permissions and conditions were copied word-for-word from the PDF Reference. If you need advanced PDF features, I highly recommended this manual as a companion for this book. You can purchase a hardcopy or download it for free from the Adobe web site (www.adobe.com).

The general idea is that developers like you and me are free to build tools that view, generate, change, or manipulate PDF files (as long as you don't crack them). And that's exactly what Paulo Soares and I did—we built a tool that let us create and manipulate PDF.

Of course, we didn't implement the complete specification; some version-specific features aren't implemented (yet), and not all the possible types of PDF are supported in iText.

3.2 Types of PDF

PDF is the de facto standard in many different sectors, including the graphic arts industry, prepress companies, and governments. Each of these markets has its own requirements and demands regarding documents, so it's obvious that, although Adobe ensures the integrity of the format through its copyright, many different types of PDF have evolved from the original specifications. Some subsets of the PDF specification were modeled into an ISO standard. Other types of PDF are so new that they aren't supported by (almost) any tools yet.

People who don't know the difference between these types of PDF files risk accepting assignments that might as well be labeled "Mission: Impossible." These

are typically the people posting questions on the mailing list with the word "urgent" in the subject, begging for assistance. Unfortunately, we're unable to help them.

It's important to make sure you and your clients are communicating in the same language when talking about PDF. That's why I made a list with different categories, which are discussed in the following sections. People with other backgrounds could organize their lists differently, but I made my list from an iText developer's point of view.

3.2.1 Traditional PDF

This isn't an official term, but I use the word *traditional* when I want to refer to the kind of PDF that is intended to be a finished product with unchangeable content and a print-ready layout. The way it looks on the screen is the way it will look when it's printed, in contrast with other formats such as RTF or HTML. The printed output of an RTF or HTML (and even a Microsoft Word) file depends on the application that is used to render it.

Traditional PDF is a read-only paginated document format that can contain all kinds of multimedia, links, bookmarks, and so forth; but it doesn't know anything about text structure. For example, traditional PDF doesn't understand the concept of a table; you can render a table in a PDF file, but you can't retrieve the data that was organized in this tabular structure from the PDF to reuse it in another application. As far as the PDF file is concerned, the table consists of some characters drawn on a canvas, along with some lines. The concept of rows and columns is lost on PDF. You'd need specialized OCR software to retrieve the original content.

In short, creating traditional PDF is a one-way process.

3.2.2 Tagged PDF

Sometimes traditional PDF isn't sufficient for your needs. You may want to produce PDF files that can adapt themselves to the device they will be used on, or you may want to repurpose the PDF file if, for example, end users will read the document on the smaller screen of their Palm Pilot. If you need to make the document accessible for the visually impaired, the PDF file should contain the logical reading order (which isn't always the case with traditional PDF). Images should be given alternate descriptions. Also, if you need to be able to recognize document structures such as paragraphs and tables, you'll need *tagged PDF*.

Tagged PDF is a stylized use of PDF; it defines a set of standard structure types and attributes that allow page content to be extracted and reused for other

purposes. Page content is represented so that the characters, words, and text order can be determined reliably. There's a basic layout model and a set of standard structure elements and attributes. Limited support for tagged PDF has been added to iText only recently (see appendix F).

3.2.3 Linearized PDF

A *linearized PDF* file is organized in a special way to enable efficient incremental access, thus enhancing the viewing performance. Its primary goal is to display the first page as quickly as possible. When data for a page is delivered over a slow channel, the page content is displayed incrementally as it arrives. Linearized PDF isn't supported by iText, but iText can read linearized PDFs just fine—an important distinction.

3.2.4 PDFs preserving native editing capabilities

I mentioned briefly that Adobe Illustrator was one of the ancestors of PDF. In Adobe Illustrator, you have the option to save files as a PDF file. If you open such a file in Illustrator, you can continue editing, just like with the native AI format. Note that these PDF files aren't suited for general, online distribution: they're larger than the traditional PDFs because they contain a lot of application-specific data. It's a matter of taste, but I wouldn't recommend using PDF as an editing format. It's not what PDF was designed for. Instead, keep the source of the document in another format and convert to PDF when needed.

3.2.5 PDF types that became an ISO standard

There are many ways to create a valid PDF file. This freedom is an advantage, but it can be a disadvantage too. Not all valid PDF files are usable in every context. To tackle this problem, different ISO standards were created.

PDF/X

In particular, the prepress sector felt the need to restrict the freedom offered by the Portable Document Format. A consortium of prepress companies got together and released specifications for *PDF/X* (the X stands for *eXchange*). PDF/X is a set of ISO standards (ISO 15930-1, -2, and -3) describing well-defined subsets of the PDF specification that promise predictable and consistent PDF files. The main goal of PDF/X-1a is to support *blind exchange* of PDF documents. Blind exchange means you can deliver PDF documents to a print service provider with hardly any technical discussion. PDF/X-3 is a superset of PDF/X-1a. The primary difference is that a PDF/X-3 file can also contain color managed data. PDF/X-2 is a superset of

PDF/X-3. It was designed for exchanges where there is more discussion between the supplier and receiver of the PDF.

Each standard has its own specific requirements and constraints, but in general, you can say that functionality that will probably break PDF/X conformance includes encryption, the use of fonts that aren't embedded, RGB colors, layers, image masks, transparency, and some blend modes. The two most useful PDF/X standards are supported by iText: PDF/X-1a:2001 and PDF/X-3:2002.

PDF/A and XMP

PDF/A is another ISO specification: ISO 19005-1:2005, "Document management—Electronic document file format for long-term preservation—Part 1: Use of PDF 1.4 (PDF/A-1)." The standard was approved in September 2005. The initiative for PDF/A was started by the Association for Information and Image Management (AIIM) and the Association for Suppliers of Printing, Publishing and Converting Technologies (NPES).

The *A* in PDF/A stands for *archiving*; there are many electronic formats (ASCII, TIFF, PDF, XML) and technologies (databases, repositories) to choose from for archiving. The proprietary nature of many of these formats is one of the biggest disadvantages: They can't be guaranteed to continue for the long term. For example, if you try to open a 10-year-old Microsoft Word file in the most recent version of Word, you can't expect it to look like it looked 10 years ago in the version that was used to create it.

As opposed to most word-processing formats, PDF represents not only the data contained in the document but also the exact form the document takes. The file can be viewed without the originating application. All the revisions of the PDF specification are backward-compatible. For example, if your viewer can read and print a PDF with version 1.6, it can also read a PDF with version 1.2. Moreover, the information about the file format is always in the public domain. Anyone, at any time, using any hardware or software, can create programs to access PDF documents.

This makes PDF an interesting candidate as a format for archiving. PDF/A goes a step further: It's a subset of PDF-1.4 intended to be suitable for long-term preservation of page-oriented documents. Just like PDF/X, PDF/A imposes some constraints: In order to meet level-B conformance, all fonts must be embedded; encryption isn't allowed; audio and video content are forbidden, as are JavaScript and executable file launches; and so forth. Level-A conformance also means the PDF has to be tagged (see the discussion of tagged PDF earlier in this chapter).

Of course, archiving isn't just about storing documents somewhere in some format. You also have to be able to search and find the documents.

Self-documentation of every archived file is important. This is where XML and, more specifically, Adobe's Extensible Metadata Platform (XMP) come into the picture. XMP is a standard format for the creation, processing, and interchange of metadata, not limited to the PDF format. Applications that don't understand PDF, JPG, PNG, or GIF syntax but are able to extract and read XMP can retrieve the metadata from files in either of these formats.

PDF/E

Another ISO standard that will emerge soon is PDF/E. You can follow the progress of this standard on the AIIM site (www.aiim.org/), where the PDF/E committee defines their scope as being "responsible for specifying PDF tags for creating, viewing, and printing documents used in engineering workflows."

The PDF/E standard doesn't exist yet, so it's evident that PDF/E isn't supported yet in iText.

3.2.6 PDF forms, FDF, and XFDF

A PDF document can contain an interactive form, sometimes referred to as an AcroForm. An *AcroForm* is a collection of fields. These fields can be used to gather information interactively from the user. They can also act as placeholders with fixed coordinates that can be filled with variable content.

In the first situation, the PDF file can be served on a web site, as if it were an HTML page with a single form. If the user clicks the Submit button, the data entered can be submitted to the web server in different formats (depending on how the submit action was defined in the AcroForm):

- *As an HTML query string*—`key1=value1&key2=value2&...` or HTML multi-part form data.
- *In the Forms Data Format (FDF)*—An FDF file contains the data of the form and a reference to the PDF file with the AcroForm. When an FDF file is opened in Adobe Reader, the original PDF is fetched, and the fields are filled with the data in the FDF.
- *In XFDF*—This is the XML-based alternative to FDF.
- *As PDF*—In this case, a complete filled-in PDF file is sent to the server (note that this is not possible if you only have Adobe Reader).

In this book, you'll also use PDFs with an AcroForm as a kind of template. You'll fill the fields with data coming from a database, XML, FDF, or XFDF. One special type of form field is the digital signature.

3.2.7 XFA and XDP

Forms that are made with Acrobat 7.0 (more specifically, with Adobe's LiveCycle Designer, which comes with Acrobat 7.0 Professional but not with the Standard version) are completely different from AcroForms. They're based on the XML Forms Architecture (XFA). The XML Data Package (XDP) provides a mechanism for packaging units of PDF content as XML. XFA resources are described as XDP packages inside the PDF. In this case, you still have a PDF file, but the form is described in XML. Forms like this aren't discussed in this book. You can read more about XFA in the XFA Specification on the Adobe web site (www.adobe.com). There is only basic XFA support.

The XML Data Package is more than just XFA. XDP is intended to be an XML-based companion to PDF. An XDP file is an XML file that encodes a PDF file in XML. An XDP file consists of five parts, many of which are optional:

- *The XML form data*—The user data encoded according to an arbitrary XML schema chosen by the designer of the form.
- *The XML form template*—Contains all the form intelligence. Maps the XML form data to PDF form fields. Holds the business logic to validate fields, calculates results, and so forth.
- *XML configuration information*—A global reference for database and web service connections.
- *Other XML information*—Metadata, schemas, and digital signatures.
- *The PDF file*—Embeds the PDF as base64 encoded.

PDF and XDP are equivalent and interchangeable representations of the same underlying electronic form. PDF offers advantages for large documents, when file size is important, or when forms contain images. XDP is interesting when forms have to fit in an XML workflow and data needs to be manipulated by software that isn't PDF-aware. For the time being, there are no plans to support XDP files in iText.

3.2.8 Rules of thumb

I'll refer to the different types of PDF files regularly in parts 2, 3, and 4 of this book. It's not essential that you remember all of them, as long as you keep the following points in mind:

- Traditional PDF is a one-way process.
- Don't abuse the phrase *PDF template*. No one will know whether you're referring to a traditional PDF file that can be *stamped*, tagged PDF files that can be *repurposed,* or a PDF form that can be *filled in*.

- If you're talking about a PDF form, always specify whether you're referring to an *AcroForm* or an *XFA* form.
- PDF is a de facto standard; PDF/X, PDF/A, and (soon) PDF/E are ISO standards.

Now that you have an idea of the types of PDF that are supported, let's look at the different PDF versions and discuss some iText-specific issues.

3.3 PDF version history

In chapter 2, you learned how to change the PDF version of the documents that are generated with iText. Table 2.1 listed the different versions and the year the specifications of these versions were published; in table 3.1 you'll find a nonrestrictive list of new features that were added in each PDF version.

Table 3.1 New features in different PDF versions

PDF version	Year	Acrobat version	New features
PDF-1.0	1993	Acrobat 1	- Ability to render complex text and graphics to the screen as well as to the printer
PDF-1.1	1994	Acrobat 2	- Ability to create a password-protected PDF - External links - Device-independent color
PDF-1.2	1996	Acrobat 3	- Flate (zip/gzip) compression - Interactive, fill-in forms - Chinese, Japanese, Korean (CJK) support
PDF-1.3	1999	Acrobat 4	- File attachments, - Digital signatures, - Logical page numbering
PDF-1.4	2001	Acrobat 5	- 128-bit encryption - Transparency - Tagged PDF
PDF-1.5	2003	Acrobat 6	- Additional compression and encryption options - Optional content groups - Enhanced support for embedding and playback of multimedia
PDF-1.6	2004	Acrobat 7	- Customizable UserUnit value - Support for Advanced Encryption Standard (AES) - Page-scaling option for printing

For a complete list, see the PDF Reference Manual. Each version of the Reference has a section in its introductory chapter detailing the latest version's new features.

A number of the features listed in table 3.1 were additions to the existing PDF specification (for example, support for 128-bit encryption and support for transparency), whereas other features led to an almost completely different type of PDF (for example, tagged PDF).

When you create a new document using iText, the default version is 1.4. In chapter 2, you used the method `setPdfVersion()` to create a PDF document in another version, but it's important to realize that this method changes only a single character in the PDF header (see section 2.1.3); iText doesn't check the compatibility of every feature you're using in your code.

In this section, we'll look at specific examples that will help you understand the implications of this limitation. You'll learn what happens if you change the user unit, a feature that was introduced in version 1.6; and you'll learn more about the compression and encryption of PDF documents, two important topics that figure in different rows of table 3.1.

3.3.1 Changing the user unit

When we discussed the first step of the iText PDF-creation process, we talked about the maximum and minimum size of a page. If you decide to create a PDF document with a version that is different from the default, you have to be careful not to create a PDF that isn't valid.

For example, if you change the PDF version to 1.3, iText won't check the page size. It's your responsibility not to insert pages that are smaller than 72 by 72 units or bigger than 3,240 by 3,240 units.

Since version 1.4, pages can have a minimum size of 3 by 3 units and a maximum of 14,400 by 14,400 units. This corresponds with a minimum page size of approximately 0.04 by 0.04 in and a maximum of 200 by 200 in, because 1 in equals 72 pt. That's true for PDF-1.4 and -1.5; but table 3.1 indicates that you can change the user unit, starting with version 1.6. The minimum value of the user unit is 1 pt (this is the default; 1 unit = 1/72 in); in PDF 1.6 it can be changed to a maximum of 75,000 pt (1 unit = 1042 in).

Let's give it a try and create a "Hello World" document with a page of 15,000,000 by 15,000,000 inches (14,400 ❶ x 75,000 ❷ x 1/72).

```
/* chapter03/HelloWorldMaximum.java */
Document document = new Document(new Rectangle(14400, 14400));   <-❶
PdfWriter writer = PdfWriter.getInstance(document,
```

```
    new FileOutputStream("HelloWorldMaximum.pdf"));
writer.setPdfVersion(PdfWriter.VERSION_1_6);
writer.setUserunit(75000f);    <-- 2
document.open();
```

Note that this document measures 381 by 381 kilometers! You'll only be able to view it correctly in Adobe Reader 7.0 or later. If you open HelloWorldMaximum.pdf in an earlier version of Acrobat Reader, you'll get a warning similar to the one Adobe Reader 6.0 is giving in figure 3.1.

Adobe Reader 6.0 can't display the page correctly because it doesn't understand the meaning of a user unit of 75,000 pt.

End users get the warning shown in figure 3.1 every time you serve them a PDF that has a higher version than the one supported by their version of Adobe Reader. This happens even if the PDF doesn't contain new functionality that can't be shown in that specific viewer application. For example, Acrobat Reader 3.0 gives a similar warning if you try to open the "Hello World" file you created in chapter 2. Once you click the OK button, the document displays correctly. That's because listing 2.1 doesn't produce any PDF syntax that isn't compatible with PDF version 1.2.

Requiring the end user to click OK can be annoying. Table 3.1 can help you decide when it's necessary to change the PDF version. If you plan to use the optional content group functionality (OCG; see chapter 12), you have to change the version of your PDF file to 1.5 or 1.6 before opening the document. Note that iText can't change the version number automatically. The PDF version number is

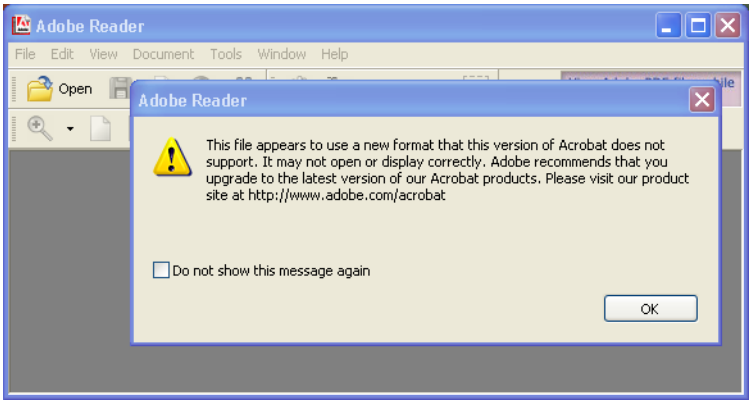

Figure 3.1 Warning when opening a PDF document with a version higher than the version of the viewer

written to the output stream in the second step of the PDF creation process; iText notices the use of OCG functionality only in the fourth step.

Changing the user unit, on the other hand, is done *before* the second step. In this case, you could have omitted the line with `setPdfVersion()`. Setting the version is done implicitly in the method `setUserUnit()`. The same happens when you use `setFullCompression()`. A glance at table 3.1 shows that flate/zip compression was introduced in PDF 1.2, but additional full compression functionality wasn't added until version 1.5.

Let's look at some examples that demonstrate the difference between uncompressed, compressed, and fully compressed files.

3.3.2 PDF content and compression

Figure 3.1 showed the warning you get when you opened your initial "Hello World" file in Acrobat Reader 3.0. In spite of this warning, Acrobat Reader was able to display the document correctly. This isn't the case if you try to open the file with Acrobat Reader 2.0. Instead of a warning, you get an error message (see figure 3.2).

The document you've generated isn't damaged; you know it opens without any problem in more recent versions of Adobe Reader. After you click OK, Acrobat Reader 2.0 gives you another message box, saying *This file contains information not understood by the viewer. Suppress further errors?*

That's a better error message. Acrobat Reader 2.0 is only supposed to support PDF version 1.1 or earlier. By default, iText compresses the content streams of each page. Acrobat Reader versions prior to 3.0 can't show compressed streams; that's what causes the error.

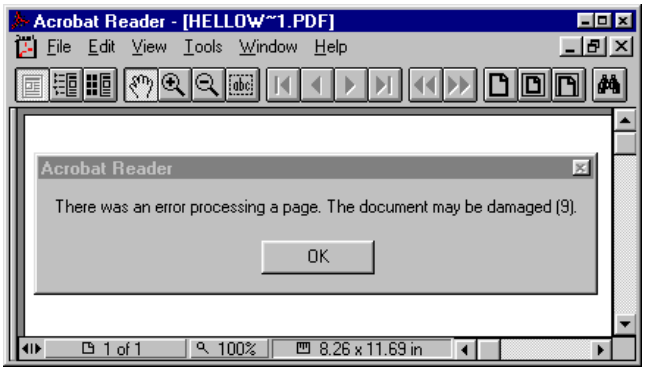

Figure 3.2
Error message prompted when opening HelloWorld.pdf in Acrobat Reader 2.

FAQ *What is the default compression when creating PDF files with iText?* Since PDF-1.2, flate/deflate compression has been the default compression used by Acrobat. This is an algorithm based on Huffman encoding and LZ77 compression, one of the first versions of Lempel-Ziv-Welch (LZW). It's also the compression iText uses by default.

If you refer again to table 2.1, you'll notice that the iText constant values for PDF-1.0 and -1.1 are missing. This was intentional; it's assumed that you aren't interested in generating a PDF file using a specification that is more than 10 years old.

Nevertheless, you can tweak iText to generate a valid 1.0 or 1.1 PDF file. The PDF header that is written to the output stream upon opening the document is stored in a `HEADER` variable. The `setPdfVersion()` method replaces one character in this `String`. You could tweak iText to generate a PDF-1.1 by calling `setPdfVersion()` and passing the char 1 as a parameter. Additionally, you'd have to turn off the default compression. Note that this example is shown for pedagogic reasons only; I don't recommend that you change the compression variable. It's a static value, so if you set compression to `false`, you do this for the entire JVM (and thus for all the PDFs you're generating in the same process). Doing so may lead to unwanted side effects:

```
/* chapter03/HelloWorldUncompressed.java */
Document.compress = false;
writer.setPdfVersion('1');
```

You can open this particular HelloWorldUncompressed.pdf file in Acrobat Reader 2.0 without getting the error message shown in figure 3.2. Mind my choice of words: You can open this *particular* file in Reader 2.0. I already explained that using `setPdfVersion()` doesn't necessarily result in files that are compliant with that version.

You've just made a PDF that was uncompressed. Why not make one that is fully compressed for a change? *Full compression* means that not only page streams are compressed, but some other objects as well, such as the cross-reference table. This is only possible since PDF-1.5:

```
/* chapter03/HelloWorldFullyCompressed.java */
writer.setFullCompression();
```

You don't set the version in this example; iText changes it to 1.5 automatically.

Existing PDF documents and compression

Suppose you have a large repository of old PDF files that aren't fully compressed. With `PdfStamper`, you can upgrade the version of these PDF files by constructing the `PdfStamper` with a version character as an extra parameter. You can then apply full compression with the method `setFullCompression()`:

```
/* chapter03/HelloWorldFullyCompressed.java */
reader = new PdfReader("HelloWorldCompressed.pdf");
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldFullCompression.pdf"),
  PdfWriter.VERSION_1_5);
stamper.setFullCompression();
stamper.close();
```

Isn't that easy? If you compare the sizes of the files, you'll see that the original file is 4211 bytes, and the one with full compression is only 3179 bytes. Just for fun, you can also decompress the file, which results in a file that is 5561 bytes long:

```
/* chapter03/HelloWorldCompression.java */
reader = new PdfReader("HelloWorldCompressed.pdf");
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldDecompressed.pdf"), '1');
Document.compress = false;
int total = reader.getNumberOfPages() + 1;
for (int i = 1; i < total; i++) {
  reader.setPageContent(i, reader.getPageContent(i));
}
stamper.close();
```

I used a trick to decompress the pages. You can get the uncompressed content stream of a page (see listing 2.2) directly from the reader with `getPageContent()`; this can be interesting if you want to debug a PDF file at the lowest level. You can set the content back with `setPageContent()`. (Note that you should have some experience with PDF before you start experimenting with these methods; you'll read more about them in chapter 18.)

Let's wrap up this chapter by covering one more topic that's mentioned several times in table 3.1: encryption.

3.3.3 Encryption

The FAQs of many tools that produce PDF documents recommend iText as a tool for post-processing PDF files. For example, Apache Formatting Objects Processor (FOP) can be used to convert XML to PDF, but it doesn't encrypt the resulting file; the FOP developers recommend using iText as a post-processor for FOP-generated PDF documents.

In the next example, you'll encrypt an existing PDF document in two different ways, and you'll learn how to decrypt an encrypted PDF file (provided that you have the needed credentials).

Encrypting existing PDF documents

To encrypt an existing PDF document, you can create a `PdfReader` object, construct a `PdfStamper` object with it, set the encryption parameters, and close the stamper:

```
/* chapter03/HelloWorldEncryptDecrypt.java */
reader = new PdfReader("HelloWorldNotEncrypted.pdf");
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldEncrypted1.pdf"));
stamper.setEncryption(
  "Hello".getBytes(), "World".getBytes(),    <-1
  PdfWriter.AllowPrinting | PdfWriter.AllowCopy,    <-2
  PdfWriter.STRENGTH40BITS);    <-3
stamper.close();
```

This looks simple, but you can do all this in a one-liner using the `PdfEncryptor` class:

```
/* chapter03/HelloWorldEncryptDecrypt.java */
PdfEncryptor.encrypt(new PdfReader("HelloWorldNotEncrypted.pdf"),
  new FileOutputStream("HelloWorldEncrypted2.pdf"),
  "Hello".getBytes(), "World".getBytes(),    <-1
  PdfWriter.AllowDegradedPrinting,    <-2
  PdfWriter.STRENGTH128BITS);    <-3
```

Note that the encrypt methods in `PdfEncryptor` use `PdfStamper` behind the scenes. The end result is exactly the same as if you used the same arguments with `PdfStamper`. In both cases, you need to pass two passwords ❶, an or-ed sequence of permissions ❷, and the strength of the encryption ❸. Let's look more closely at these parameters.

PDF passwords

The PDF standard security handler allows *access permissions* and up to two passwords to be specified for a document: a *user password* (sometimes referred to as the *open password*) and an *owner password* (sometimes referred to as the *permissions password*). Encryption applies to all strings and streams used in the PDF objects, but not to other types such as integers and boolean values needed to define the document's structure rather than its content.

In the examples, the user must enter the password "Hello" in order to open the files HelloWorldEncrypted1.pdf and HelloWorldEncrypted2.pdf. The PDF

file is locked for everyone who doesn't know the password. If you want to read the PDF file in order to change the permissions (and possibly decrypt it), you need the owner password. Remember that the owner password (in this case, "World") will also let you open the PDF file.

The maximum password length is 32 characters: You can enter longer passwords, but only the first 32 characters will be taken into account. One or both of the passwords can be null. If you don't specify a user password, all users will be able to open the document without being prompted for a password, but the permissions and restrictions (if any) will remain in place. This protection is merely psychological. The encryption key is derived from the user password, so omitting this password doesn't provide real security: The content is encrypted as described in the PDF Reference. You could write a program to decrypt such a file, but that would be illegal.

It's even easier to decrypt a file if no owner password was specified; again, you can read the PDF Reference to learn how to change the permissions of the file. If you want decent protection for your document, choose 128-bit key length and always set both passwords, using different strings and all 32 characters for each one. If you choose a password shorter than 32 characters, it will be padded with default padding (as described in the PDF Reference).

Passwords such as "Hello" and "World" are good for simple examples because they make it easy for you to test (reducing the possibility that you can't open the document due to a slip of the keyboard); but in a production environment, you should use passwords that are more complex. Remember that anyone with one of the passwords will be able to remove all the permissions from the file. If users have the owner/permissions password, they can do this legally. If they have the user/open password, they can use rogue software to decrypt the content and create an unprotected copy.

Speaking of protection, let's sum up the permissions that can be applied to a PDF document.

Overview of the permissions

Encryption is often used to enforce restrictions. The permissions that can be granted or restricted depend on the *strength* of the encryption; there's 40-bit encryption and 128-bit encryption. A quick glance at table 3.1 tells you that 128-bit encryption became possible only in PDF-1.4. In iText, you can use `PdfWriter.STRENGTH40BITS` or `PdfWriter.STRENGTH128BITS` as a parameter to pass to the `setEncryption()` or `encrypt()` method.

Permissions are or-ed like this: `PdfWriter.AllowPrinting | PdfWriter.AllowCopy`.

TOOLBOX *`com.lowagie.tools.plugins.Encrypt` (Encrypt)* With this tool, you can encrypt an unencrypted PDF document as you did in the examples. Notice that if you're using this tool from the command line, the permissions argument is a series of 0 and 1 `String` values.

Table 3.2 provides an overview of all the possible values. If you're using 40-bit encryption, every permission that has the remark "128 bit" is granted automatically. If you want to revoke these permissions, you need to use 128-bit encryption. As you can see, 128-bit encryption offers more fine-grained permission levels.

Table 3.2 Overview of the permission parameters

Static final in iText	Description of permission	Remark
`PdfWriter.AllowPrinting`	Printing the document.	
`PdfWriter.AllowDegradedPrinting`	Printing the document, but not with the quality offered by `PdfWriter.AllowPrinting`.	128 bit
`PdfWriter.AllowModifyContents`	Modifying the contents—for example, changing the content of a page, or inserting or removing a page.	
`PdfWriter.AllowAssembly`	Inserting, removing, and rotating pages and adding bookmarks is allowed. The content of a page can't be changed (unless the permission `PdfWriter.AllowModifyContents` is granted too).	128 bit
`Pdfwriter.AllowCopy`	Copying or otherwise extracting text and graphics from the document, including assistive technologies such as screen readers or other accessibility devices.	
`PdfWriter.AllowScreenReaders`	Extracting text and graphics for use by accessibility devices.	128 bit
`PdfWriter.AllowModifyAnnotations`	Adding or modifying text annotations and interactive form fields.	
`PdfWriter.AllowFillIn`	Filling form fields; adding or modifying annotations only if `PdfWriter.AllowModifyAnnotations` is granted too.	128 bit

FAQ *How do you revoke permission to save or copy a PDF file?* It isn't possible to restrict someone from saving or copying a PDF file. You can't disable the Save (or Save As) option in Adobe Reader. And even if you could, people would always be able to retrieve and copy the file with another tool. This isn't an iText issue—it goes beyond standard PDF security.

If you really need this kind of protection, you must look for a Digital Rights Management (DRM) solution. DRM tools give you fine-grained control over the document. There are different DRM software vendors, but these tools are rather expensive.

If you have an existing file that is encrypted, you can get its permissions with the `getPermissions()` method of `PdfReader`. This method returns a value that is rather cryptic. You can get a verbose overview of the permissions using `getPermissionsVerbose()`, a static method in `PdfEncryptor`:

```
/* chapter03/HelloWorldEncryptDecrypt.java */
System.out.println("Encrypted? " + reader.isEncrypted());
if (reader.isEncrypted()) {
System.out.println("Permissions: " +
  PdfEncryptor.getPermissionsVerbose(reader.getPermissions()));
  System.out.println("128 bit? " + reader.is128Key());
}
```

We have discussed all the parameters needed for encryption. You've used them to encrypt an existing PDF document. In the next example, you'll use these parameters to create a PDF document from scratch.

Encrypting a PDF document generated from scratch

The `PdfWriter` class has a `setEncryption()` method that takes the same parameters as the `PdfStamper` method with the same name. If you go back to the reference example in chapter 2, it's sufficient to add one extra line after the second step:

```
/* chapter03/HelloWorldEncrypted.java */
PdfWriter writer
  = PdfWriter.getInstance(document, new
   FileOutputStream("HelloWorldEncrypted.pdf"));
writer.setEncryption(PdfWriter.STRENGTH128BITS,    <-- 3
  "Hello", "World",    <-- 1
  PdfWriter.AllowCopy | PdfWriter.AllowPrinting);    <-- 2
```

Note that the order of the parameters is slightly different.

You've been encrypting PDF files, both existing and new, but if you want to read an encrypted PDF file with `PdfReader`, you need a constructor that takes a password as parameter.

Decrypting an existing PDF file

If you try reading an encrypted PDF file with `PdfReader`, an exception will be thrown if you don't provide the owner password. If you *do* know the owner password, decrypting a PDF file with iText is simple. Create the reader object with the constructor that takes the password as parameter ❶, construct the stamper object ❷ and close it immediately afterward ❸:

```
/* chapter03/HelloWorldEncryptDecrypt.java */
reader = new PdfReader("HelloWorldEncrypted1.pdf", "World".getBytes()); ❶
stamper = new PdfStamper(reader,
  new FileOutputStream("HelloWorldDecrypted.pdf")); ❷
stamper.close();  ❸
```

You've just created an unencrypted version of an encrypted PDF file.

TOOLBOX *`com.lowagie.tools.plugins.Decrypt` (Encrypt)* With this tool, you can decrypt an encrypted PDF document as you did in the example.

Note that changing the compression and/or encryption of a PDF file is easy when using iText. It's sufficient to change some settings. If you want to know more about the compression and/or encryption algorithms that are used behind the scenes, please consult the PDF Reference.

We have dealt with three version-specific features that are mentioned in table 3.1. I won't go into detail about the differences between the versions prior to PDF-1.4, but whenever we encounter functionality that was added after version 1.4 (the default version used by iText), I'll mention this in the text. That way, you'll know if and when it's necessary to change the PDF version in your source code.

3.4 Summary

This chapter started with a general overview of the Portable Document Format. We talked about the origins and the initial purpose of PDF. PDF has become a de facto standard, but you've seen that along the way different types of PDF and different real ISO standards have emerged. We have discussed how to deal with different PDF versions when using iText. The concepts of user unit, compression, and encryption were introduced in a series of simple examples. This concludes the first part of this book.

In the second part, you'll create traditional PDF documents using iText's basic building blocks. There will be no need to change the PDF version. All the files will

be generated in the default version: PDF 1.4. In part 3, we'll encounter some more advanced functionality. You'll still be producing traditional PDF files, but you'll need to change the version once you start working with optional content groups. Part 4 will deal with interactive PDF, including some very recent PDF functionality. You'll also work with other types of PDF: PDF documents with AcroForms and FDF and XFDF files.

If you haven't done so already, now is the time to roll up your sleeves and start doing some real work!

JAVA/PDF

iText IN ACTION

Creating and Manipulating PDF

Bruno Lowagie

Say you need a tool to add dynamic or interactive features to a PDF file and you decide to search on Google for "Java PDF." What do you think you'd find? Why, at the top of the page you'd find "iText," of course. A leading tool for programmatic creation and manipulation of PDF documents, iText is an open source Java library developed and maintained by Bruno Lowagie, the author of this book, with the help of many contributors.

While at the entry level iText is easy to learn, developers find they soon need its more advanced features. Written by the master himself, *iText in Action* now offers an introduction and a practical guide to the subject—you will gain a sound understanding of the Portable Document Format and how to do interesting and useful things with PDF using iText.

iText in Action introduces iText and lowers the learning curve to its advanced features. Its numerous, valuable examples unlock many of the secrets hidden in Adobe's *PDF Reference*. The examples are in Java but they can be easily adapted to .NET using one of iText's .NET ports: iTextSharp (C#) or iText.NET (J#).

What's Inside

How to
- Serve PDF to a browser
- Generate dynamic documents from XML files or databases
- Use PDF's many interactive features
- Add bookmarks, page numbers, watermarks, etc.
- Split, concatenate, and manipulate PDF pages
- Automate filling out of PDF forms
- Add digital signatures to a PDF file
- And much more

Bruno Lowagie is the original developer and one of the current maintainers of iText. He works for Ghent University and lives in Ghent, Belgium with his wife and two sons.

"Thorough and complete … will be a long running, valuable resource for iText and PDF."
—Alan Dennis, Software Architect, MyFamily.com

"One of the best technical books I have ever read! Great work!"
—Oliver Zeigermann
Technical Trainer, CoreMedia AG

"I wholeheartedly recommend it."
—Doug James, eReporting Team Lead, Benefitfocus.com, Inc.

"Impressive! It provides depth without all the noise."
—Justin Lee
President, Antwerkz Inc.

"Valuable to any developer using PDF."
—Stuart Caborn
Consultant, Thoughtworks

www.manning.com/lowagie

ISBN 1-932394-79-6

$49.99 US/$64.99 Canada