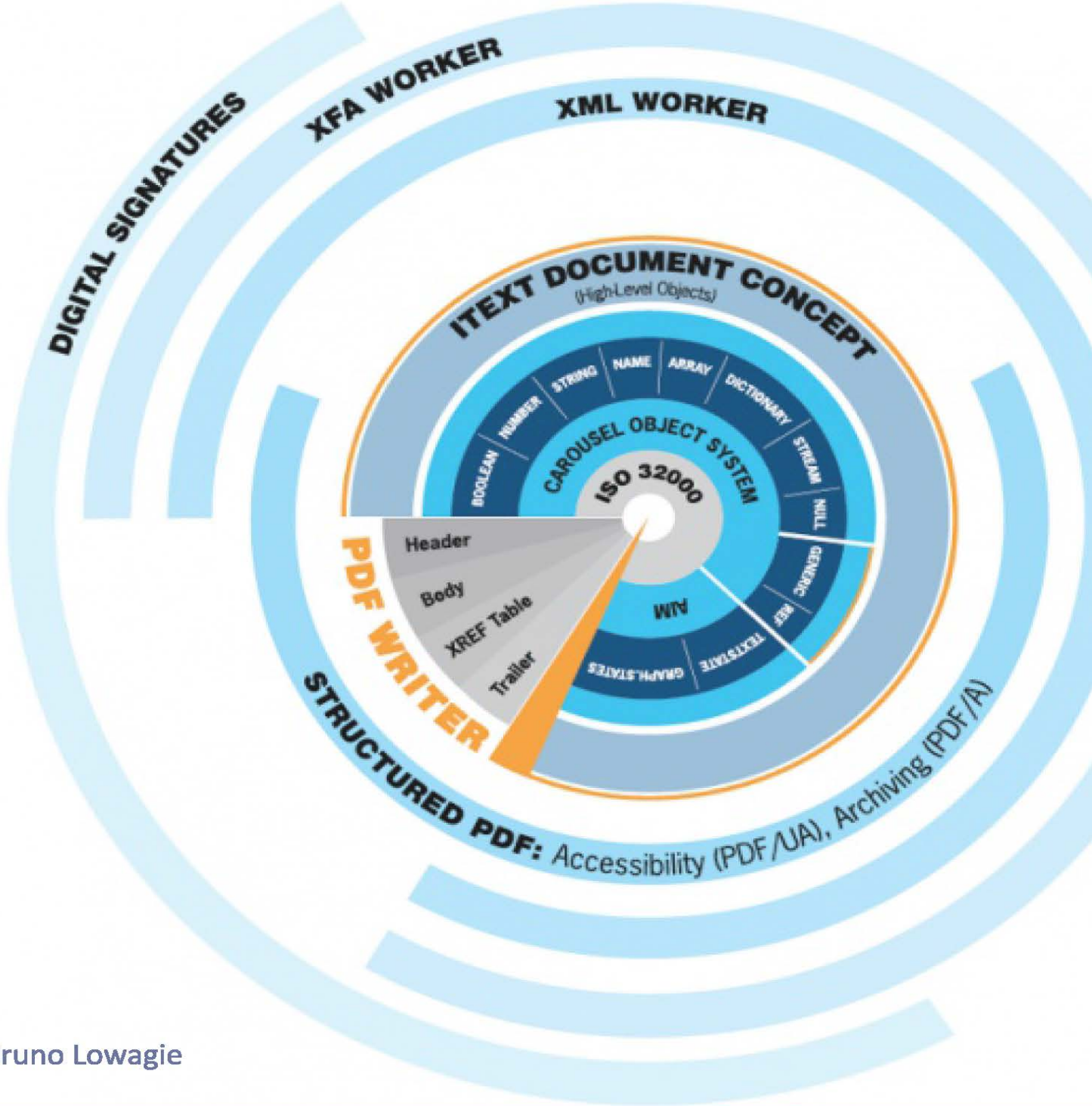
The ABC of PDF with iText

PDF Syntax essentials



by Bruno Lowagie

The ABC of PDF with iText

PDF Syntax essentials

iText Software

This book is for sale at http://leanpub.com/itext_pdfabc

This version was published on 2015-01-06

Tweet This Book!

Please help iText Software by spreading the word about this book on Twitter!

The suggested tweet for this book is:

@iText: I just bought The ABC of PDF with iText

The suggested hashtag for this book is #itext_pdfabc.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#itext_pdfabc

Also By iText Software

The Best iText Questions on StackOverflow

Contents

Introduction

This book is a vademecum for the other iText books entitled "Create your PDFs with iText[1]," "Update your PDFs with iText[2]," and "Sign your PDFs with iText[3]."

In the past, I used to refer to ISO-32000 whenever somebody asked me questions such as *"why can't I use PDF as a format for editing documents"* or whenever somebody wanted to use a feature that wasn't supported out-of-the-box.

I soon realized that answering *"read the specs"* is lethal when the specs consist of more than a thousand pages. In this iText tutorial, I'd like to present a short introduction to the syntax of the Portable Document Format. It's not the definitive guide, but it should be sufficient to help you out when facing a PDF-related problem.

You'll find some simple iText examples in this book, but the heavy lifting will be done in the other iText books.

[1] https://leanpub.com/itext_pdfcreate
[2] https://leanpub.com/itext_pdfupdate
[3] https://leanpub.com/itext_pdfsign

I Part 1: The Carousel Object System

The Portable Document Format (PDF) specification, as released by the International Organization for Standardization (ISO) in the form of a series of related standards (ISO-32000-1 and -2, ISO-19005-1, -2, and -3, ISO-14289-1,...), was originally created by Adobe Systems Inc.

Carousel was the original code name for what later became Acrobat. The name Carousel was already taken by Kodak, so a marketing consultant was asked for an alternative name. These were the names that were proposed:

- *Adobe Traverse*– didn't make it,
- *Adobe Express*– sounded nice, but there was already that thing called Quark Express,
- *Adobe Gates*– was never an option, because there was already somebody with that name at another company,
- *Adobe Rosetta*– couldn't be used, because there was an existing company that went by that name.
- *Adobe Acrobat*– was a name not many people liked, but it was chosen anyway.

Although Acrobat exists for more than 20 years now, the name Carousel is still used to refer to the way a PDF file is composed, and that's what the first part of this book is about.

In this first part, we'll:

- Take a look at the basic PDF objects,
- Find out how these objects are organized inside a file, and
- Learn how to read a file by navigating from object to object.

At the end of this chapter, you'll know how PDF is structured and you'll understand what you see when opening a PDF in a text editor instead of inside a PDF viewer.

1. PDF Objects

There are eight basic types of objects in PDF. They're explained in sections 7.3.2 to 7.3.9 of ISO-32000-1.

1.1 The basic PDF objects

These eight objects are implemented in iText as subclasses of the abstract `PdfObject` class. Table 1.1 lists these types as well as their corresponding objects in iText.

Table 1.1: Overview of the basic PDF objects

PDF Object	iText object	Description
Boolean	`PdfBoolean`	This type is similar to the Boolean type in programming languages and can be `true` or `false`.
Numeric object	`PdfNumber`	There are two types of numeric objects: integer and real. Numbers can be used to define coordinates, font sizes, and so on.
String	`PdfString`	String objects can be written in two ways: as a sequence of literal characters enclosed in parentheses `( )` or as hexadecimal data enclosed in angle brackets `< >`. Beginning with PDF 1.7, the type is further qualified as text string, PDFDocEncoded string, ASCII string, and byte string, depending upon how the string is used in each particular context.
Name	`PdfName`	A name object is an atomic symbol uniquely defined by a sequence of characters. Names can be used as keys for a dictionary, to define an explicit destination type, and so on. You can easily recognize names in a PDF file because they're all introduced with a forward slash: `/`.
Array	`PdfArray`	An array is a one-dimensional collection of objects, arranged sequentially between square brackets. For instance, a rectangle is defined as an array of four numbers: `[0 0 595 842]`.
Dictionary	`PdfDictionary`	A dictionary is an associative table containing pairs of objects known as dictionary entries. The key is always a name; the value can be (a reference to) any other object. The collection of pairs is enclosed by double angle brackets: `<<` and `>>`.
Stream	`PdfStream`	Like a string object, a stream is a sequence of bytes. The main difference is that a PDF consumer reads a string entirely, whereas a stream is best read incrementally. Strings are used for small pieces of data; streams are used for large amounts of data.

Table 1.1: Overview of the basic PDF objects

PDF Object	iText object	Description
		Each stream consists of a dictionary followed by zero or more bytes enclosed between the keywords `stream` (followed by a newline) and `endstream`.
Null object	`PdfNull`	This type is similar to the `null` object in programming languages. Setting the value of a dictionary entry to `null` is equivalent to omitting the entry.

If you look inside iText, you'll find subclasses of these basic PDF implementations created for specific purposes.

- `PdfDate` extends `PdfString` because a date is a special type of string in the Portable Document Format.
- `PdfRectangle` is a special type of `PdfArray`, consisting of four number values: `[llx, lly, urx, ury]` representing the coordinates of the lower-left and upper-right corner of the rectangle.
- `PdfAction`, `PdfFormField`, `PdfOutline` are examples of subclasses of the `PdfDictionary` class.
- `PRStream` is a special implementation of `PdfStream` that needs to be used when extracting a stream from an existing PDF document using `PdfReader`.

When creating or manipulating PDF documents with iText, you'll use high-level objects and convenience methods most of the time. This means you probably won't be confronted with these basic objects very often, but it's interesting to take a look under the hood of iText.

1.2 iText's PdfObject implementations

Let's take a look at some simple code samples for each of the basic types.

1.2.1 PdfBoolean

As there are only two possible values for the `PdfBoolean` object, you can use a static instance instead of creating a new object.

Code sample 1.1: C0101_BooleanObject

```
1  public static void main(String[] args) {
2      showObject(PdfBoolean.PDFTRUE);
3      showObject(PdfBoolean.PDFFALSE);
4  }
5  public static void showObject(PdfBoolean obj) {
6      System.out.println(obj.getClass().getName() + ":");
7      System.out.println("-> boolean? " + obj.isBoolean());
8      System.out.println("-> type: " + obj.type());
9      System.out.println("-> toString: " + obj.toString());
10     System.out.println("-> booleanvalue: " + obj.booleanValue());
11 }
```

In code sample 1.1, we use `PdfBoolean`'s constant values `PDFTRUE` and `PDFFALSE` and we inspect these objects in the `showObject()` method. We get the fully qualified name of the class. We use the `isBoolean()` method that will return `false` for all objects that aren't derived from `PdfBoolean`. And we display the `type()` in the form of an `int` (this value is 1 for `PdfBoolean`).

All `PdfObject` implementations have a `toString()` method, but only the `PdfBoolean` class has a `booleanValue()` method that allows you to get the value as a primitive Java `boolean` value.

The output of the `showObject` method looks like this:

```
com.itextpdf.text.pdf.PdfBoolean:
-> boolean? true
-> type: 1
-> toString: true
-> booleanvalue: true
com.itextpdf.text.pdf.PdfBoolean:
-> boolean? true
-> type: 1
-> toString: false
-> booleanvalue: false
```

We'll use the `PdfBoolean` object in the tutorial Update your PDFs with iText[1] when we'll update properties of dictionaries to change the behavior of a PDF feature.

1.2.2 PdfNumber

There are many different ways to create a `PdfNumber` object. Although PDF only has two types of numbers (integer and real), you can create a `PdfNumber` object using a `String`, `int`, `long`, `double` or `float`.

This is shown in code sample 1.2.

Code sample 1.2: C0102_NumberObject

```
1  public static void main(String[] args) {
2      showObject(new PdfNumber("1.5"));
3      showObject(new PdfNumber(100));
4      showObject(new PdfNumber(1001));
5      showObject(new PdfNumber(1.5));
6      showObject(new PdfNumber(1.5f));
7  }
8  public static void showObject(PdfNumber obj) {
9      System.out.println(obj.getClass().getName() + ":");
10     System.out.println("-> number? " + obj.isNumber());
11     System.out.println("-> type: " + obj.type());
12     System.out.println("-> bytes: " + new String(obj.getBytes()));
```

[1] https://leanpub.com/itext_pdfupdate

```
13      System.out.println("-> toString: " + obj.toString());
14      System.out.println("-> intValue: " + obj.intValue());
15      System.out.println("-> longValue: " + obj.longValue());
16      System.out.println("-> doubleValue: " + obj.doubleValue());
17      System.out.println("-> floatValue: " + obj.floatValue());
18  }
```

Again we display the fully qualified classname. We check for number objects using the `isNumber()` method. And we get a different value when we asked for the type (more specifically: 2).

The `getBytes()` method returns the bytes that will be stored in the PDF. In the case of numbers, you'll get a similar result using `toString()` method. Although iText works with `float` objects internally, you can get the value of a `PdfNumber` object as a primitive Java `int`, `long`, `double` or `float`.

```
com.itextpdf.text.pdf.PdfNumber:
-> number? true
-> type: 2
-> bytes: 1.5
-> toString: 1.5
-> intValue: 1
-> longValue: 1
-> doubleValue: 1.5
-> floatValue: 1.5
com.itextpdf.text.pdf.PdfNumber:
-> number? true
-> type: 2
-> bytes: 100
-> toString: 100
-> intValue: 100
-> longValue: 100
-> doubleValue: 100.0
-> floatValue: 100.0
```

Observe that you lose the decimal part if you invoke the `intValue()` or `longValue()` method on a real number. Just like with `PdfBoolean`, you'll use `PdfNumber` only if you hack a PDF at the lowest level, changing a property in the syntax of an existing PDF.

1.2.3 PdfString

The `PdfString` class has four constructors:

- An empty constructor in case you want to create an empty `PdfString` object (in practice this constructor is only used in subclasses of `PdfString`),
- A constructor that takes a Java `String` object as its parameter,

- A constructor that takes a Java `String` object as well as the encoding value (`TEXT_PDFDOCENCODING` or `TEXT_UNICODE`) as its parameters,
- A constructor that takes an array of bytes as its parameter in which case the encoding will be `PdfString.NOTHING`. This method is used by iText when reading existing documents into PDF objects.

You can choose to store the PDF string object in hexadecimal format by using the `setHexWriting()` method:

Code sample 1.3: C0103_StringObject

```
1  public static void main(String[] args) {
2      PdfString s1 = new PdfString("Test");
3      PdfString s2 = new PdfString("\u6d4b\u8bd5", PdfString.TEXT_UNICODE);
4      showObject(s1);
5      showObject(s2);
6      s1.setHexWriting(true);
7      showObject(s1);
8      showObject(new PdfDate());
9  }
10 public static void showObject(PdfString obj) {
11     System.out.println(obj.getClass().getName() + ":");
12     System.out.println("-> string? " + obj.isString());
13     System.out.println("-> type: " + obj.type());
14     System.out.println("-> bytes: " + new String(obj.getBytes()));
15     System.out.println("-> toString: " + obj.toString());
16     System.out.println("-> hexWriting: " + obj.isHexWriting());
17     System.out.println("-> encoding: " + obj.getEncoding());
18     System.out.println("-> bytes: " + new String(obj.getOriginalBytes()));
19     System.out.println("-> unicode string: " + obj.toUnicodeString());
20 }
```

In the output of code sample 1.3, we see the fully qualified name of the class. The `isString()` method returns `true`. The type value is 3. In this case, the `toBytes()` method can return a different value than the `toString()` method. The `String` `"\u6d4b\u8bd5"` represents two Chinese characters meaning “test”, but these characters are stored as four bytes.

Hexademical writing is applied at the moment the bytes are written to a PDF `OutputStream`. The encoding values are stored as `String` values, either `"PDF"` for PdfDocEncoding, `"UnicodeBig"` for Unicode, or `""` in case of a pure byte string.

The `getOriginalBytes()` method only makes sense when you get a `PdfString` value from an existing file that was encrypted. It returns the original encrypted value of the string object.

The `toUnicodeString()` method is a safer method than `toString()` to get the PDF string object as a Java `String`.

```
com.itextpdf.text.pdf.PdfString:
-> string? true
-> type: 3
-> bytes: Test
-> toString: Test
-> hexWriting: false
-> encoding: PDF
-> original bytes: Test
-> unicode string: Test
com.itextpdf.text.pdf.PdfString:
-> string? true
-> type: 3
-> bytes: □□mK□□
-> toString: □□
-> hexWriting: false
-> encoding: UnicodeBig
-> original bytes: □□mK□□
-> unicode string: □□
com.itextpdf.text.pdf.PdfString:
-> string? true
-> type: 3
-> bytes: Test
-> toString: Test
-> hexWriting: true
-> encoding: PDF
-> original bytes: Test
-> unicode string: Test
com.itextpdf.text.pdf.PdfDate:
-> string? true
-> type: 3
-> bytes: D:20130430161855+02'00'
-> toString: D:20130430161855+02'00'
-> hexWriting: false
-> encoding: PDF
-> original bytes: D:20130430161855+02'00'
-> unicode string: D:20130430161855+02'00'
```

In this example, we also create a `PdfDate` instance. If you don't pass a parameter, you get the current date and time. You can also pass a Java `Calendar` object if you want to create an object for a specific date. The format of the date conforms to the international Abstract Syntax Notation One (ASN.1) standard defined in ISO/IEC 8824. You recognize the pattern `YYYYMMDDHHmmSSOHH' mm` where `YYYY` is the year, `MM` the month, `DD` the day, `HH` the hour, `mm` the minutes, `SS` the seconds, `OHH` the relationship to Universal Time (UT), and `' mm` the offset from UT in minutes.

1.2.4 PdfName

There are different ways to create a `PdfName` object, but you should only use one. The constructor that takes a single `String` as a parameter guarantees that your name object conforms to ISO-32000-1 and -2.

You probably wonder why we would add constructors that allow people names that don't conform with the PDF specification. With iText, we did a great effort to ensure the creation of documents that comply. Unfortunately, this can't be said about all PDF creation software. We need some `PdfName` constructors that accept any kind of value when reading names in documents that are in violation with the PDF ISO standards.

In many cases, you don't need to create a `PdfName` object yourself. The `PdfName` object contains a large set of constants with predefined names. One of these names is used in code sample 1.4.

Code sample 1.4: C0104_NameObject

```
public static void main(String[] args) {
    showObject(PdfName.CONTENTS);
    showObject(new PdfName("CustomName"));
    showObject(new PdfName("Test #1 100%"));
}
public static void showObject(PdfName obj) {
    System.out.println(obj.getClass().getName() + ":");
    System.out.println("-> name? " + obj.isName());
    System.out.println("-> type: " + obj.type());
    System.out.println("-> bytes: " + new String(obj.getBytes()));
    System.out.println("-> toString: " + obj.toString());
}
```

The `getClass().getName()` part no longer has secrets for you. We use `isName()` to check if the object is really a name. The type is 4. And we can get the value as bytes or as a `String`.

```
com.itextpdf.text.pdf.PdfName:
-> name? true
-> type: 4
-> bytes: /Contents
-> toString: /Contents
com.itextpdf.text.pdf.PdfName:
-> name? true
-> type: 4
-> bytes: /CustomName
-> toString: /CustomName
com.itextpdf.text.pdf.PdfName:
-> name? true
-> type: 4
```

```
-> bytes: /Test#20#231#20100#25
-> toString: /Test#20#231#20100#25
```

Note that names start with a forward slash, also know as a *solidus*. Also take a closer look at the name that was created with the `String` value `"Test #1 100%"`. iText has escaped values such as `' '`, `'#'` and `'%'` because these are forbidden in a PDF name object. ISO-32000-1 and -2 state that a name is a sequence of 8-bit values and iText's interprets this literally. If you pass a string containing multibyte characters (characters with a value greater than 255), iText will only take the lower 8 bits into account. Finally, iText will throw an `IllegalArgumentException` if you try to create a name that is longer than 127 bytes.

1.2.5 PdfArray

The `PdfArray` class has six constructors. You can create a `PdfArray` using an `ArrayList` of `PdfObject` instances, or you can create an empty array and add the `PdfObject` instances one by one (see code sample 1.5). You can also pass a byte array of `float` or `int` values as parameter in which case you create an array consisting of `PdfNumber` objects. Finally you can create an array with a single object if you pass a `PdfObject`, but be carefull: if this object is of type `PdfArray`, you're using the copy constructor.

Code sample 1.5: C0105_ArrayObject

```
1  public static void main(String[] args) {
2      PdfArray array = new PdfArray();
3      array.add(PdfName.FIRST);
4      array.add(new PdfString("Second"));
5      array.add(new PdfNumber(3));
6      array.add(PdfBoolean.PDFFALSE);
7      showObject(array);
8      showObject(new PdfRectangle(595, 842));
9  }
10 public static void showObject(PdfArray obj) {
11     System.out.println(obj.getClass().getName() + ":");
12     System.out.println("-> array? " + obj.isArray());
13     System.out.println("-> type: " + obj.type());
14     System.out.println("-> toString: " + obj.toString());
15     System.out.println("-> size: " + obj.size());
16     System.out.print("-> Values:");
17     for (int i = 0; i < obj.size(); i++) {
18         System.out.print(" ");
19         System.out.print(obj.getPdfObject(i));
20     }
21     System.out.println();
22 }
```

Once more, we see the fully qualified name in the output. The `isArray()` method tests if this class is a `PdfArray`. The value of the array type is 5.

The elements of the array are stored in an `ArrayList`. The `toString()` method of the `PdfArray` class returns the `toString()` output of this `ArrayList`: the values of the separate objects delimited with a comma and enclosed by square brackets. The `getBytes()` method returns `null`.

You can ask a `PdfArray` for its size, and use this size to get the different elements of the array one by one. In this case, we use the `getPdfObject()` method. We'll discover some more methods to retrieve elements from an array in section 1.3.

```
com.itextpdf.text.pdf.PdfArray:
-> array? true
-> type: 5
-> toString: [/First, Second, 3, false]
-> size: 4
-> Values: /First Second 3 false
com.itextpdf.text.pdf.PdfRectangle:
-> array? true
-> type: 5
-> toString: [0, 0, 595, 842]
-> size: 4
-> Values: 0 0 595 842
```

In our example, we created a `PdfRectangle` using only two values `595` and `842`. However, a rectangle needs four values: two for the coordinate of the lower-left corner, two for the coordinate of the upper-right corner. As you can see, iText added two zeros for the coordinate of the lower-left coordinate.

1.2.6 PdfDictionary

There are only two constructors for the `PdfDictionary` class. With the empty constructor, you can create an empty dictionary, and then add entries using the `put()` method. The constructor that accepts a `PdfName` object will create a dictionary with a `/Type` entry and use the name passed as a parameter as its value. This entry identifies the type of object the dictionary describes. In some cases, a `/SubType` entry is used to further identify a specialized subcategory of the general type.

In code sample 1.6, we create a custom dictionary and an action.

Code sample 1.6: C0106_DictionaryObject

```
1 public static void main(String[] args) {
2     PdfDictionary dict = new PdfDictionary(new PdfName("Custom"));
3     dict.put(new PdfName("Entry1"), PdfName.FIRST);
4     dict.put(new PdfName("Entry2"), new PdfString("Second"));
5     dict.put(new PdfName("3rd"), new PdfNumber(3));
6     dict.put(new PdfName("Fourth"), PdfBoolean.PDFFALSE);
7     showObject(dict);
8     showObject(PdfAction.gotoRemotePage("test.pdf", "dest", false, true));
9 }
```

```
10  public static void showObject(PdfDictionary obj) {
11      System.out.println(obj.getClass().getName() + ":");
12      System.out.println("-> dictionary? " + obj.isDictionary());
13      System.out.println("-> type: " + obj.type());
14      System.out.println("-> toString: " + obj.toString());
15      System.out.println("-> size: " + obj.size());
16      for (PdfName key : obj.getKeys()) {
17          System.out.print(" " + key + ": ");
18          System.out.println(obj.get(key));
19      }
20  }
```

The `showObject()` method shows us the fully qualified names. The `isDictionary()` returns `true` and the `type()` method returns 6.

Just like with `PdfArray`, the `getBytes()` method returns `null`. iText stores the objects in a `HashMap`. The `toString()` method of a `PdfDictionary` doesn't reveal anything about the contents of the dictionary, except for its type if present. The type entry is usually optional. For instance: the `PdfAction` dictionary we created in code sample 1.6 doesn't have a `/Type` entry.

We can ask a dictionary for its number of entries using the `size()` method and get each value as a `PdfObject` by its key. As the entries are stored in a `HashMap`, the keys aren't shown in the same order we used to add them to the dictionary. That's not a problem. The order of entries in a dictionary is irrelevant.

```
com.itextpdf.text.pdf.PdfDictionary:
-> dictionary? true
-> type: 6
-> toString: Dictionary of type: /Custom
-> size: 4
 /3rd: 3
 /Entry1: /First
 /Type: /Custom
 /Fourth: false
 /Entry2: Second
com.itextpdf.text.pdf.PdfAction:
-> dictionary? true
-> type: 6
-> toString: Dictionary
-> size: 4
 /D: dest
 /F: test.pdf
 /S: /GoToR
 /NewWindow: true
```

As explained in table 1.1, a PDF dictionary is stored as a series of key value pairs enclosed by `<<` and `>>`. The action created in code sample 1.6 looks like this when viewed in a plain text editor:

```
<</D(dest)/F(test.pdf)/S/GoToR/NewWindow true>>
```

The basic `PdfDictionary` object has plenty of subclasses such as `PdfAction`, `PdfAnnotation`, `PdfCollection`, `PdfGState`, `PdfLayer`, `PdfOutline`, etc. All these subclasses serve a specific purpose and they were created to make it easier for developers to create objects without having to worry too much about the underlying structures.

1.2.7 PdfStream

The `PdfStream` class also extends the `PdfDictionary` object. A stream object always starts with a dictionary object that contains at least a `/Length` entry of which the value corresponds with the number of stream bytes.

For now, we'll only use the constructor that accepts a `byte[]` as parameter. The other constructor involves a `PdfWriter` instance, which is an object we haven't discussed yet. Although that constructor is mainly for internal use —it offers an efficient, memory friendly way to write byte streams of unknown length to a PDF document—, we'll briefly cover this alternative constructor in the Create your PDFs with iText[2] tutorial.

Code sample 1.7: C0107_StreamObject

```
1  public static void main(String[] args) {
2      PdfStream stream = new PdfStream(
3       "Long stream of data stored in a FlateDecode compressed stream object"
4       .getBytes());
5      stream.flateCompress();
6      showObject(stream);
7  }
8  public static void showObject(PdfStream obj) {
9      System.out.println(obj.getClass().getName() + ":");
10     System.out.println("-> stream? " + obj.isStream());
11     System.out.println("-> type: " + obj.type());
12     System.out.println("-> toString: " + obj.toString());
13     System.out.println("-> raw length: " + obj.getRawLength());
14     System.out.println("-> size: " + obj.size());
15     for (PdfName key : obj.getKeys()) {
16         System.out.print(" " + key + ": ");
17         System.out.println(obj.get(key));
18     }
19 }
```

In the lines following the fully qualified name, we see that the `isStream()` method returns `true` and the `type()` method returns 7. The `toString()` method returns nothing more than the word `"Stream"`.

[2] https://leanpub.com/itext_pdfcreate

We can store the long `String` we used in code sample 1.7 "as is" inside the stream. In this case, invoking the `getBytes()` method will return the bytes you used in the constructor.

If a stream is compressed, for instance by using the `flateCompress()` method, the `getBytes()` method will return `null`. In this case, the bytes are stored inside a `ByteArrayOutputStream` and you can write these bytes to an `OutputStream` using the `writeContent()` method. We didn't do that because it doesn't make much sense for humans to read a compressed stream.

The `PdfStream` instance remembers the original length aka the raw length. The length of the compressed stream is stored in the dictionary.

```
com.itextpdf.text.pdf.PdfStream:
-> stream? true
-> type: 7
-> toString: Stream
-> raw length: 68
-> size: 2
 /Filter: /FlateDecode
 /Length: 67
```

In this case, compression didn't make much sense: 68 bytes were compressed into 67 bytes. In theory, you could choose a different compression level. The `PdfStream` class has different constants such as `NO_COMPRESSION` (0), `BEST_SPEED` (1) and `BEST_COMPRESSION` (9). In practice, we'll always use `DEFAULT_COMPRESSION` (-1).

1.2.8 PdfNull

We're using the `PdfNull` class internally in some very specific cases, but there's very little chance you'll ever need to use this class in your own code. For instance: it's better to remove an entry from a dictionary than to set its value to `null`; it saves the PDF consumer processing time when parsing the files you've created.

Code sample 1.8: C0108_NullObject

```
1  public static void main(String[] args) {
2      showObject(PdfNull.PDFNULL);
3  }
4  public static void showObject(PdfNull obj) {
5      System.out.println(obj.getClass().getName() + ":");
6      System.out.println("-> type: " + obj.type());
7      System.out.println("-> bytes: " + new String(obj.getBytes()));
8      System.out.println("-> toString: " + obj.toString());
9  }
```

The output of code sample 1.8 is pretty straight-forward: the fully qualified name of the class, its type (8) and the output of the `getBytes()` and `toString()` methods.

```
com.itextpdf.text.pdf.PdfNull:
-> type: 8
-> bytes: null
-> toString: null
```

These were the eight basic types, numbered from 1 to 8. Two more numbers are reserved for specific `PdfObject` classes: 0 and 10. Let's start with the class that returns 0 when you call the `type()` method.

1.2.9 PdfLiteral

The objects we've discussed so far were literally the first objects that were written when I started writing iText. Since 2000, they've been used to build billions of PDF documents. They form the foundation of iText's object-oriented approach to create PDF documents.

Working in an object-oriented way is best practice and it's great, but for some straight-forward objects, you wish you'd have a short-cut. That's why we created `PdfLiteral`. It's an iText object you won't find in the PDF specification or ISO-32000-1 or -2. It allows you to create any type of object with a minimum of overhead.

For instance: we often need an array that defines a specific matrix, called the identity matrix. It consists of six elements: 1, 0, 0, 1, 0 and 0. Should we really create a `PdfArray` object and add these objects one by one? Wouldn't it be easier if we just created the literal array: `[1 0 0 1 0 0]`?

That's what `PdfLiteral` is about. You create the object passing a `String` or a `byte[]`; you can even pass the object type to the constructor.

Code sample 1.9: C0109_LiteralObject

```
1  public static void main(String[] args) {
2      showObject(PdfFormXObject.MATRIX);
3      showObject(new PdfLiteral(
4          PdfObject.DICTIONARY, "<</Type/Custom/Contents [1 2 3]>>"));
5  }
6  public static void showObject(PdfObject obj) {
7      System.out.println(obj.getClass().getName() + ":");
8      System.out.println("-> type: " + obj.type());
9      System.out.println("-> bytes: " + new String(obj.getBytes()));
10     System.out.println("-> toString: " + obj.toString());
11 }
```

The `MATRIX` constant used in code sample 1.9 was created like this: `new PdfLiteral("[1 0 0 1 0 0]");` when we write this object to a PDF, it is treated in exactly the same way as if we'd had created a `PdfArray`, except that its type is 0 because `PdfLiteral` doesn't parse the `String` to check the type.

We also create a custom dictionary, telling the object its type is `PdfObject.DICTIONARY`. This doesn't have any impact on the fully qualified name. As the `String` passed to the constructor isn't being parsed, you can't ask the dictionary for its size nor get the key set of the entries.

The content is stored *literally*, as indicated in the name of the class: `PdfLiteral`.

```
com.itextpdf.text.pdf.PdfLiteral:
-> type: 0
-> bytes: [1 0 0 1 0 0]
-> toString: [1 0 0 1 0 0]
com.itextpdf.text.pdf.PdfLiteral:
-> type: 6
-> bytes: <</Type/Custom/Contents [1 2 3]>>
-> toString: <</Type/Custom/Contents [1 2 3]>>
```

It goes without saying that you should be very careful when using this object. As iText doesn't parse the content to see if its syntax is valid, you'll have to make sure you don't make any mistakes. We use this object internally as a short-cut, or when we encounter content that can't be recognized as being one of the basic types whilst reading an existing PDF file.

1.3 The difference between direct and indirect objects

To explain what the iText `PdfObject` with value `10` is about, we need to introduce the concept of indirect objects. So far, we've been working with direct objects. For instance: you create a dictionary and you add an entry that consists of a PDF name and a PDF string. The result looks like this:

```
<</Name (Bruno Lowagie)>>
```

The string value with my name is a *direct object*, but I could also create a PDF string and label it:

```
1 0 obj
(Bruno Lowagie)
endobj
```

This is an *indirect object* and we can refer to it from other objects, for instance like this:

```
<</Name 1 0 R>>
```

This dictionary is equivalent to the dictionary that used a direct object for the string. The `1 0 R` in the latter dictionary is called an *indirect reference*, and its iText implementation is called `PdfIndirectReference`. The type value is `10` and you can check if a `PdfObject` is in fact an indirect reference using the `isIndirect()` method.

A stream object may never be used as a direct object. For example, if the value of an entry in a dictionary is a stream, that value always has to be an indirect reference to an indirect object containing a stream. A stream dictionary can never be an indirect object. It always has to be a direct object.

An indirect reference can refer to an object of any type. We'll find out how to obtain the actual object referred to by an indirect reference in chapter 3.

1.4 Summary

In this chapter, we've had an overview of the building blocks of a PDF file:

- boolean,
- number,
- string,
- name,
- array,
- dictionary,
- stream, and
- null

Building blocks can be organized as numbered indirect objects that reference each other.

It's difficult to introduce code samples explaining how direct and indirect objects interact, without seeing the larger picture. So without further ado, let's take a look at the file structure of a PDF document.

2. PDF File Structure

Figure 2.1 shows a simple, single-page PDF document with the text “Hello World” opened in Adobe Reader.

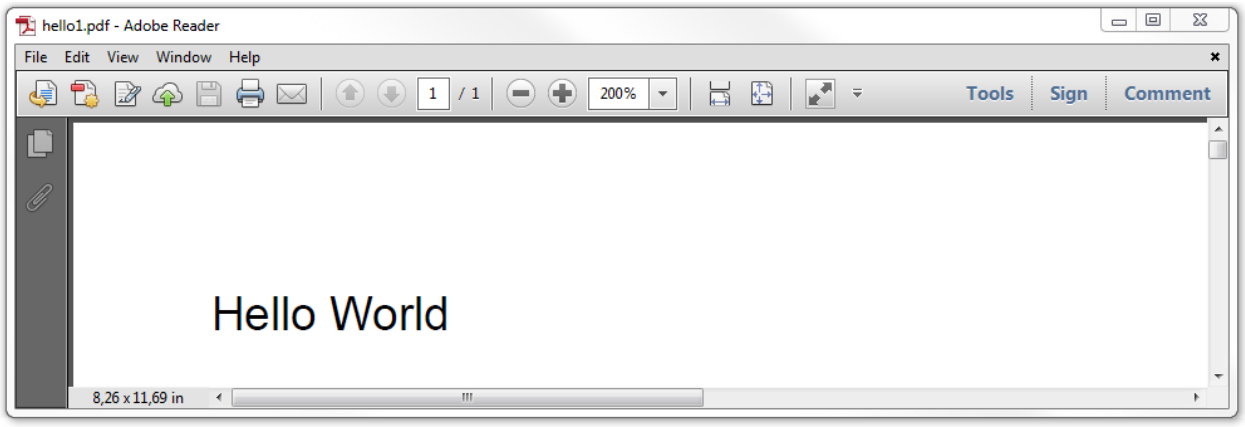


Figure 2.1: Hello World

Now let’s open the file in a text editor and examine its internal structure.

2.1 The internal structure of a PDF file

When we open the “Hello World” document in a plain text editor instead of in a PDF viewer, we soon discover that a PDF file consists of a sequence of indirect objects as described in the previous chapter.

Table 2.1 shows how to find the four different parts that define the “Hello World” document listed in code sample 2.1:

Table 2.1: Overview of the parts of a PDF file

Part	Name	Line numbers
1	The Header	Lines 1-2
2	The Body	Lines 3-24
3	The Cross-reference Table	Lines 25-33
4	The Trailer	Lines 34-40

Note that I’ve replaced a binary content stream by the words `*binary stuff*`. Lines that were too long to fit on the page were split; a \ character marks where the line was split.

Code sample 2.1: A PDF file inside-out

```
%PDF-1.4
%âãÏÓ
2 0 obj
<</Length 64/Filter/FlateDecode>>stream
*binary stuff*
endstream
endobj
4 0 obj
<</Parent 3 0 R/Contents 2 0 R/Type/Page/Resources<</ProcSet [/PDF /Text /ImageB /ImageC /\
ImageI]/Font<</F1 1 0 R>>>>/MediaBox[0 0 595 842]>>
endobj
1 0 obj
<</BaseFont/Helvetica/Type/Font/Encoding/WinAnsiEncoding/Subtype/Type1>>
endobj
3 0 obj
<</Type/Pages/Count 1/Kids[4 0 R]>>
endobj
5 0 obj
<</Type/Catalog/Pages 3 0 R>>
endobj
6 0 obj
<</Producer(iText® 5.4.2 ©2000-2012 1T3XT BVBA \(AGPL-version\))/ModDate(D:20130502165150+\
02'00')/CreationDate(D:20130502165150+02'00')>>
endobj
xref
0 7
0000000000 65535 f
0000000302 00000 n
0000000015 00000 n
0000000390 00000 n
0000000145 00000 n
0000000441 00000 n
0000000486 00000 n
trailer
<</Root 5 0 R/ID [<91bee3a87061eb2834fb6a3258bf817e><91bee3a87061eb2834fb6a3258bf817e>]/In\
fo 6 0 R/Size 7>>
%iText-5.4.2
startxref
639
%%EOF
```

Let's examine the four parts that are present in code sample 2.1 one by one.

2.1.1 The Header

Every PDF file starts with `%PDF-`. If it doesn't, a PDF consumer will throw an error and refuse to open the file because it isn't recognized as a valid PDF file. For instance: iText will throw an `InvalidPdfException` with the message *"PDF header signature not found."*

iText supports the most recent PDF specifications, but uses version 1.4 by default. That's why our "Hello World" example (that was created using iText) starts with `%PDF-1.4`.

Beginning with PDF 1.4, the PDF version can also be stored elsewhere in the PDF. More specifically in the *root* object of the document, aka the *catalog*. This implies that a file with header `%PDF-1.4` can be seen as a PDF 1.7 file if it's defined that way in the document root. This allows the version to be changed in an incremental update without changing the original header.

The second line in the header needs to be present if the PDF file contains binary data (which is usually the case). It consists of a percent sign, followed by at least four binary characters. That is: characters whose codes are 128 or greater. This ensures proper behavior of the file transfer applications that inspect data near the beginning of a file to determine whether to treat the file's contents as a text file, or as a binary file.

Line 1 and 2 start with a percent sign (%). Any occurence of this sign outside a string or stream introduces a comment. Such a comment consists of all characters after the percent sign up to (but not including) the End-of-Line marker. Except for the header lines discussed in this section and the End-of-File marker `%%EOF`, comments are ignored by PDF readers because they have no semantical meaning,

The Body of the document starts on the third line.

2.1.2 The Body

We recognize six indirect objects between line 3 and 24 in code sample 2.1. They aren't ordered sequentially:

1. Object 2 is a stream,
2. Object 4 is a dictionary of type `/Page`,
3. Object 1 is a dictionary of type `/Font`,
4. Object 3 is a dictionary of type `/Pages`,
5. Object 5 is a dictionary of type `/Catalog`, and
6. Object 6 is a dictionary for which no type was defined.

A PDF producer is free to add these objects in any order it desires. A PDF consumer will use the cross-reference table to find each object.

2.1.3 The Cross-reference Table

The cross-reference table starts with the keyword `xref` and contains information that allows access to the indirect objects in the body. For reasons of performance, a PDF consumer doesn't read the entire file.

Imagine a document with 10,000 pages. If you only want to see the last page, a PDF viewer doesn't need to read the content of the 9,999 previous pages. It can use the cross-reference table to retrieve only those objects needed as a resource for the requested page.

The keyword `xref` is followed by a sequence of lines that either consist of two numbers, or of exactly 20 bytes. In code sample 2.1, the cross-reference table starts with `0 7`. This means the next line is about object 0 in a series of seven consecutive objects: 0, 1, 2, 3, 4, 5, and 6.

There can be gaps in a cross-reference table. For instance, an additional line could be `10 3` followed by three lines about objects 10, 11, and 12.

The lines with exactly 20 bytes consist of three parts separated by a space character:

1. a 10-digit number representing the byte offset,
2. a 5-digit number indicates the generation of the object,
3. a keyword, either `n` if the object is *in use*, or `f` if the object is *free*.

Each of these lines ends with a 2-byte End-of-Line sequence.

The first entry in the cross-reference table representing object 0 at position 0 is always a free object with the highest possible generation number: 65,535. In code sample 2.1, it is followed by 6 objects that are in use: object 1 starts at byte position 302, object 2 at position 15, and so on.

Since PDF 1.5, there's another, more compact way to create a cross-reference table, but let's first take a look at the final part of the PDF file in code sample 2.1, the trailer.

2.1.4 The Trailer

The trailer starts with the keyword `trailer`, followed by the *trailer dictionary*. The trailer dictionary in line 35-36 of code sample 2.1 consists of four entries:

- The `/ID` entry is a file identifier consisting of an array of two byte sequences. It's only required for encrypted documents, but it's good practice to have them because some workflows depend on each document to be uniquely identified (this implies that no two files use the same identifier). For documents created from scratch, the two parts of the identifier should be identical.
- The `/Size` entry shows the total number of entries in the file's cross-reference table, in this case 7.
- The `/Root` entry refers to object 5. This is a dictionary of type `/Catalog`. This root object contains references to other objects defining the content. The Catalog dictionary is the starting point for PDF consumers that want to read the contents of a document.

- The `/Info` entry refers to object 6. This is the info dictionary. This dictionary can contain metadata such as the title of the document, its author, some keywords, the creation date, etc. This object will be deprecated in favor of XMP metadata in the next PDF version (PDF 2.0 defined in ISO-32000-2).

Other possible entries in the trailer dictionary are the `/Encrypt` key, which is required if the document is encrypted, and the `/Prev` key, which is present if the file has more than one cross-reference section. This will occur in the case of PDFs that are updated in append mode as will be explained in section 2.2.1.

Every PDF file ends with three lines consisting of the keyword `startxref`, a byte position, and the keyword `%%EOF`. In the case of code sample 2.1, the byte position points to the location of the `xref` keyword of the most recent cross-reference table.

Let's take a look at some variations on this file structure.

2.2 Variations on the file structure

Depending on the document requirements of your project, you'll expect a slightly different structure:

- When a document is updated and the bytes of the previous revision need to remain intact,
- When a document is postprocessed to allow fast web access, or
- When file size is important and therefore full compression is recommended.

Let's take a look at the possible impact of these requirements on the file structure.

2.2.1 PDFs with more than one cross-reference table

There are different ways to update the contents of a PDF document. One could take the objects of an existing PDF, apply some changes by adding and removing objects, and creating a new structure where the existing objects are reordered and renumbered. That's the default behavior of iText's `PdfStamper` class.

In some cases, this behavior isn't acceptable. If you want to add an extra signature to a document that was already signed, changing the structure of the existing document will break the original signature. You'll have to preserve the bytes of the original document and add new objects, a new cross-reference table and a new trailer. The same goes for *Reader enabled* files, which are files signed using Adobe's private key, adding specific usage rights to the file.

Code sample 2.2 shows three extra parts that can be added to code sample 2.1 (after line 40): an extra body, an extra cross-reference table and an extra trailer. This is only a simple example of a possible update to an existing PDF document; no extra visible content was added. We'll see a more complex example in the tutorial Sign your PDFs with iText[1].

[1] https://leanpub.com/itext_pdfsign

Code sample 2.2: A PDF file inside-out (part 2)

```
6 0 obj
<</Producer(iText® 5.4.2 ©2000-2012 1T3XT BVBA \(AGPL-version\))/ModDate(D:20130502165150+\
02'00')/CreationDate(D:20130502165150+02'00')>>
endobj
xref
0 1
0000000000 65535 f
6 1
0000000938 00000 n
trailer
<</Root 5 0 R/Prev 639/ID [<91bee3a87061eb2834fb6a3258bf817e><84c1b02d932693e4927235c277cc\
489e>]/Info 6 0 R/Size 7>>
%iText-5.4.2
startxref
1091
%%EOF
```

When we look at the new cross-reference table, we see that object 0 is again a free object, whereas object 6 is now updated.

Object 6 is reused and therefore the generation number doesn't need to be incremented. It remains `00000`. In practice, the generation number is only incremented if the status of an object changes from `n` to `f`.

Observe that the `/Prev` key in the trailer dictionary refers to the byte position where the previous cross-reference starts.

The first element of the `/ID` array generally remains the same for a given document. This helps Enterprise Content Management (ECM) systems to detect different versions of the same document. They shouldn't rely on it, though, as not all PDF processors support this feature. For instance: iText's `PdfStamper` will respect the first element of the ID array; `PdfCopy` typically won't because there's usually more than one document involved when using `PdfCopy`, in which case it doesn't make sense to prefer the identifier of one document over the identifier of another.

The file parts shown in code sample 2.2 are an incremental update. All changes are appended to the end of the file, leaving its original contents intact. One document can have many incremental updates.

The principle of having multiple cross-reference streams is also used in the context of linearization.

2.2.2 Linearized PDFs

A linearized PDF file is organized in a special way to enable efficient incremental access. Linearized PDF is sometimes referred to as PDF for "fast web view." Its primary goal is to enhance the viewing performance

whilst downloading a PDF file over a streaming communications channel such as the internet. When data for a page is delivered over the channel, you'd like to have the page content displayed incrementally as it arrives.

With the essential cross-reference at the end of the file, this isn't possible unless the file is linearized. All the content in the PDF file needs to be reorganized so that the first page can be displayed as quickly as possible without the need to read all of the rest of the file, or to start reading with the final cross-reference file at the very end of the file.

Such a reorganization of the PDF objects, creating a cross-reference for each page, can only be done after the PDF file is completed and after all resources are known. iText can read linearized PDFs, but it can't create a linearized PDF, nor can you (currently) linearize an existing PDF using iText.

2.2.3 PDFs with compressed object and cross-reference streams

Starting with PDF 1.5, the cross reference table can be stored as an indirect object in the body, more specifically as a stream object allowing big cross-reference tables to be compressed. Additionally, the file size can be reduced by putting different objects into one compressed object stream.

Code sample 2.3 has the same appearance as code sample 2.1 when opened in a PDF viewer, but the internal file structure is quite different:

Code sample 2.3: Compressed PDF file structure

```
%PDF-1.5
%âãÏÓ
2 0 obj
<</Length 64/Filter/FlateDecode>>stream
*binary stuff*
endstream
endobj
6 0 obj
<</Type/Catalog/Pages 3 0 R>>
endobj
7 0 obj
<</Producer(iText® 5.4.2 ©2000-2012 1T3XT BVBA \(AGPL-version\))/ModDate(D:20130502165150+\
02'00')/CreationDate(D:20130502165150+02'00')>>
endobj
5 0 obj
<</Type/ObjStm/N 3/Length 191/First 16/Filter/FlateDecode>>stream
*binary stuff*
endstream
endobj
8 0 obj
<</Type/XRef/W[1 2 2]/Root 6 0 R/Index[0 9]/ID [<21cb45acb652a807ba62d55f7b29d8be><21cb45a\
cb652a807ba62d55f7b29d8be>]/Length 41/Info 7 0 R/Size 9/Filter/FlateDecode>>stream
*binary stuff*
endstream
endobj
```

```
26 %iText-5.4.2
27 startxref
28 626
29 %%EOF
```

Note that the header now says `%PDF-1.5`. When I created this file, I've opted for full compression before opening the `Document` instance, and iText has automatically changed the version to 1.5.

The `startxref` value on line 28 no longer refers to the byte position of an `xref` keyword, but to the byte position of the stream object containing the cross-reference stream.

The stream dictionary of a cross-reference stream has a `/Length` and a `/Filter` entry just like all other streams, but also requires some extra entries as listed in table 2.2.

Table 2.2: Entries specific to a cross-reference stream dictionary

Key	Type	Value
Type	name	Required; always `/XRef`.
W	array	Required; an array of integers representing the size of the fields in a single cross reference entry.
Root	dictionary	Required; refers to the catalog dictionary; equivalent to the `/Root` entry in the trailer dictionary.
Index	array	An array containing a pair of integers for each subsection in the cross-reference table. The first integer shall be the first object number in the subsection; the second integer shall be the number of entries in the subsection.
ID	array	An array containing a pair of IDs equivalent to the `/ID` entry in the trailer dictionary.
Info	dictionary	An info dictionary, equivalent to the `/Info` entry in the trailer dictionary (deprecated in PDF 2.0).
Size	integer	Required; equivalent to the `/Size` entry in the trailer dictionary.
Prev	integer	Equivalent of the `/Prev` key in the trailer dictionary. Refers to the byte offset of the beginning of the previous cross-reference stream (if such a stream is present).

If we look at code sample 2.3, we see that the `/Size` of the cross-reference table is 9, and all entries are organized in one subsection `[0 9]`, which means the 9 entries are numbered from 0 to 8. The value of the `W` key, in our case `[1 2 2]`, tells us how to distinguish the different cross-reference entries in the stream, as well as the different parts of one entry.

Let's examine the stream by converting each byte to a hexadecimal number and by adding some extra white space so that we recognize the `[1 2 2]` pattern as defined in the `W` key:

```
00 0000 ffff
02 0005 0001
01 000f 0000
02 0005 0002
02 0005 0000
01 0157 0000
01 0091 0000
01 00be 0000
00 0000 ffff
```

We see 9 entries, representing objects 0 to 8. The first byte can be one out of three possible values:

- If the first byte is `00`, the entry refers to a free entry. We see that object 0 is free (as was to be expected), as well as object 8, which is the object that stores the cross-reference stream itself.
- If the first byte is `01`, the entry refers to an object that is present in the body as an uncompressed indirect object. This is the case for objects 2, 5, 6, and 7. The second part of the entry defines the byte offset of these objects: 15 (`000f`), 343 (`0157`), 145 (`0091`) and 190 (`00be`). The third part is the generation number.
- If the first byte is `02`, the entry refers to a compressed object. This is the case with objects 1, 3, and 4. The second part gives you the number of the object stream in which the object is stored (in this case object 5). The third part is the index of the object within the object stream.

Objects 1, 3, and 4 are stored in object 5. This object is an object stream, and its stream dictionary requires some extra keys as listed in table 2.3.

Table 2.3: Entries specific to an object stream dictionary

Key	Type	Value
Type	name	Required; always `/ObjStm`.
N	integer	Required; the number of indirect objects stored in the stream.
First	integer	Required; the byte offset in the decoded stream of the first compressed object
Extends	stream	A reference to another object stream, of which the current object shall be considered an extension.

The `N` value of the stream dictionary in code sample 2.3 tells us that there are three indirect objects stored in the object stream. The entries in the cross-reference stream tell us that these objects are numbered and ordered as 4, 1, and 3. The `First` value tells us that object 4 starts at byte position 16.

We'll find three pairs of integers, followed by three objects starting at byte position 16 when we uncompress the object stream stored in object 5. I've added some extra newlines to the uncompressed stream so that we can distinguish the different parts:

```
4 0
1 142
3 215
<</Parent 3 0 R/Contents 2 0 R/Type/Page/Resources<</ProcSet [/PDF /Text /ImageB /ImageC /\
ImageI]/Font<</F1 1 0 R>>>>/MediaBox[0 0 595 842]>>
<</BaseFont/Helvetica/Type/Font/Encoding/WinAnsiEncoding/Subtype/Type1>>
<</Type/Pages/Count 1/Kids[4 0 R]>>
```

The three pairs of integers consist of the numbers of the objects (4, 1, and 3), followed by their offset relative to the first object stored in the stream. We recognize a dictionary of type `/Page` (object 4), a dictionary of type `/Font` (object 1), and a dictionary of type `/Pages` (object 3).

You can never store the following objects in an object stream:

- stream objects,
- objects with a generation number different from zero,
- a document's encryption dictionary,
- an object representing the value of the `/Length` entry in an object stream dictionary,
- the document catalog dictionary,
- the linearization dictionary, and
- page objects of a linearized file.

Now that we know how a cross-reference is organized and how indirect objects are stored either in the body or inside a stream, we can retrieve all the relevant PDF objects stored in a PDF file.

2.3 Summary

In this chapter, we've examined the four parts of a PDF file: the header, the body, the cross-reference table and the trailer. We've learned that some PDFs have incremental updates, that the cross-reference table can be compressed into an object, and that objects can be stored inside an object stream. We can now start exploring the file structure of every PDF file that can be found in the wild.

While looking under the hood of some simple PDF documents, we've encountered objects such as the Catalog dictionary, Pages dictionaries, Page dictionaries, and so on. It's high time we discover how these objects relate to each other and how they form a document.

3. PDF Document Structure

In chapter 1, we've learned about the different types of objects available in the Portable Document Format, and we discovered that one object can refer to another using an indirect reference. In chapter 2, we've learned how the objects are stored in a file, as well as where to find indirect objects based on their object number. In this chapter, we're going to combine this knowledge to find out how these objects are structured into a hierarchy that defines a document.

3.1 Viewing a document as a tree structure using RUPS

The seemingly linear sequence of PDF objects we see when we open a PDF file in a text editor, isn't as linear as one might think at first sight.

Figure 3.1 shows the "Hello World" document we examined in code sample 2.1, opened in iText RUPS.

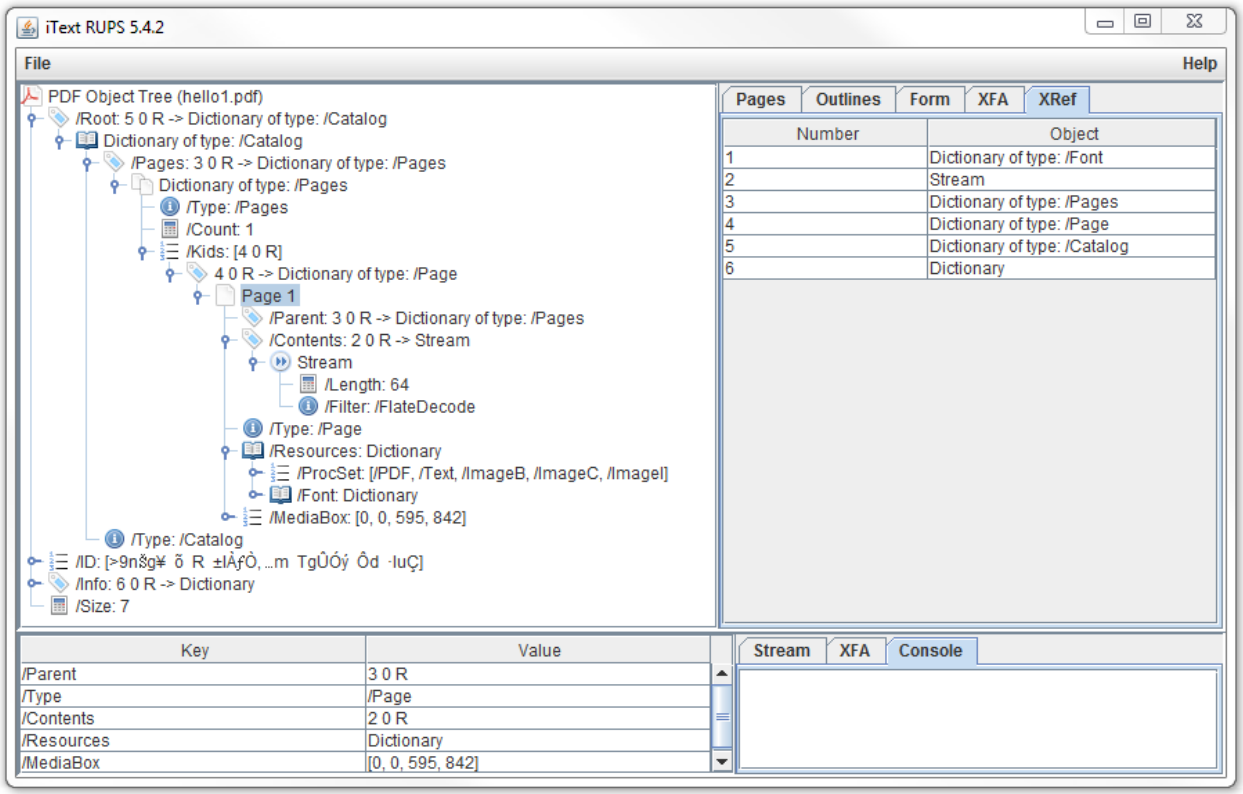


Figure 3.1: Hello World opened in iText RUPS

RUPS offers a Graphical User Interface that allows you to look inside a PDF. It's written in Java and compiled to a Windows executable. You can download the source code and the binary from SourceForge[1].

[1] http://sourceforge.net/projects/itextrups/

To the left, you recognize the entries of the trailer dictionary (see section 2.1.4). These entries are visualized in a Tree-view panel as the branches of a tree. The most prominent branch is the `/Root` dictionary. In figure 3.1, we've opened the `/Pages` dictionary, and we've unfolded the leaves of the `/Page` dictionary representing "Page 1" of the document.

To the right, there's a panel with different tabs. We see the **XRef** tab, listing the entries of the cross-reference table. It contains all the objects we discussed in section 2.1.3, organized in a table with rows numbered from 1 to 6. Clicking a row opens the corresponding object in the Tree-view panel. We'll take a look at the other tabs later on.

At the bottom, we can find info about the object that was selected. In this case, RUPS shows a tabular structure listing the keys and values of the `/Page` dictionary that was opened in the tree view panel.

To the right, we see another panel with different tabs. The **Console** tab shows whatever output is written to the `System.out` or `System.err` while using RUPS. Here's where you'll find the stack trace when you try reading a file that can't be parsed by iText because it contains invalid PDF syntax. We'll have a closer look at the **Stream** panel in part 2 and at the **XFA** panel in part 3 of this book.

Figure 3.2 shows the "Hello World" document we examined in code sample 2.3.

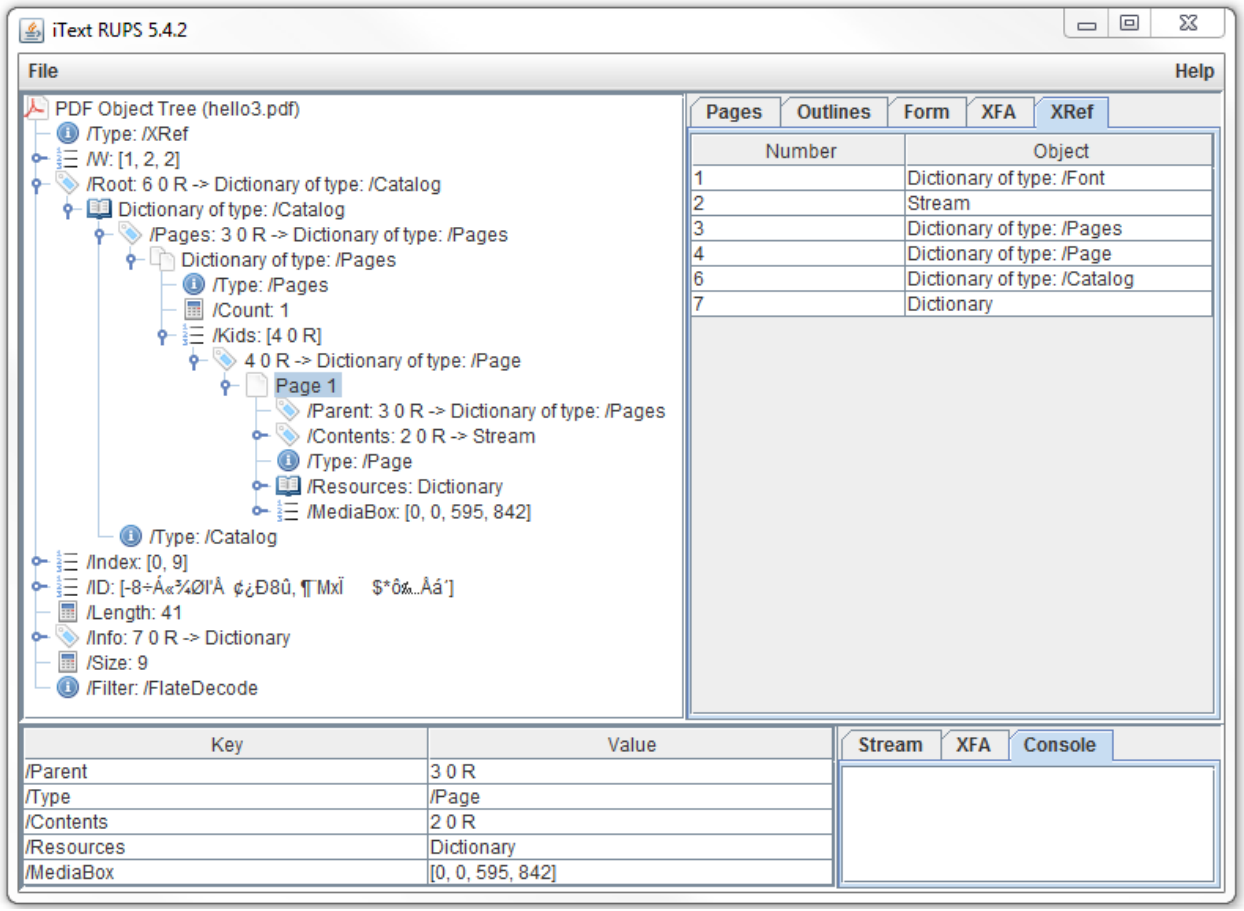


Figure 3.2: Compressed Hello World opened in iText RUPS

When you open a file with a compressed cross-reference stream, RUPS shows the `/XRef` dictionary instead of the trailer dictionary (because there is no trailer dictionary).

The **XRef** table on the right is also slightly different. Based on what we know from section 2.2.3 about this "Hello World" file, we notice that two objects are missing:

- *object 5* — a compressed object stream. Instead of showing the original stream, RUPS shows the objects that were compressed into this stream: 4, 1 and 3.
- *object 8* — the compressed cross-reference stream. This stream isn't shown either; instead its content is interpreted and visualized in the **XRef** tab.

When you open a document that was incrementally updated in RUPS, you'll only see the most recent objects. RUPS doesn't show any unused objects.

The history behind RUPS

I wrote RUPS out of frustration, at a time iText wasn't generating any revenue. When I needed to debug a PDF file, I used to open that PDF in a text editor. I then had to search through that text file looking for specific object numbers and references. When I needed to examine streams, I used the iText toolbox, a predecessor of RUPS, to decompress the binary data.

All of this was very time-consuming and almost unaffordable as long as I didn't get paid for debugging other people's documents. So I've spent the Christmas holidays of 2007 writing a GUI to "Read and Update PDF Syntax" aka "RUPS". Rups is the Dutch word for caterpillar, and I imagined the GUI as a tool to penetrate into the heart of a document, the way a caterpillar eats its way through the leaves of a plant.

My initial idea was to also allow people to change objects at their core and by doing so, to update their PDFs manually. We've only recently started implementing functionality that allows updating keys in dictionaries and applying other minor changes. Such functionality makes it very easy for people who aren't fluent in PDF to cause serious damage to a PDF file. We still aren't sure if it's a good idea to allow this kind of PDF updating.

Now that we have a means to look at the document structure using a tool with a GUI, let's find out how we can obtain the different objects that compose a PDF document programmatically, using code.

3.2 Obtaining objects from a PDF using PdfReader

When you open a document with RUPS, RUPS uses iText's `PdfReader` class under the hood. This class allows you to inspect a PDF file at the lowest level. Code sample 3.1 shows how we can create such a `PdfReader` instance and fetch different objects.

Code sample 3.1: C0301_TrailerInfo

```
1  public static void main(String[] args) throws IOException {
2      PdfReader reader =
3          new PdfReader("src/main/resources/primes.pdf");
4      PdfDictionary trailer = reader.getTrailer();
5      showEntries(trailer);
6      PdfNumber size = (PdfNumber)trailer.get(PdfName.SIZE);
7      showObject(size);
8      size = trailer.getAsNumber(PdfName.SIZE);
9      showObject(size);
10     PdfArray ids = trailer.getAsArray(PdfName.ID);
11     PdfString id1 = ids.getAsString(0);
12     showObject(id1);
13     PdfString id2 = ids.getAsString(1);
14     showObject(id2);
15     PdfObject object = trailer.get(PdfName.INFO);
16     showObject(object);
17     showObject(trailer.getAsDict(PdfName.INFO));
18     PdfIndirectReference ref = trailer.getAsIndirectObject(PdfName.INFO);
19     showObject(ref);
20     object = reader.getPdfObject(ref.getNumber());
21     showObject(object);
22     object = PdfReader.getPdfObject(trailer.get(PdfName.INFO));
23     showObject(object);
24     reader.close();
25 }
26 public static void showEntries(PdfDictionary dict) {
27     for (PdfName key : dict.getKeys()) {
28         System.out.print(key + ": ");
29         System.out.println(dict.get(key));
30     }
31 }
32 public static void showObject(PdfObject obj) {
33     System.out.println(obj.getClass().getName() + ":");
34     System.out.println("-> type: " + obj.type());
35     System.out.println("-> toString: " + obj.toString());
36 }
```

In this code sample, we create a `PdfReader` object that is able to read and interpret the PDF syntax stored in the file `primes.pdf`. This `reader` object will allow us to obtain any indirect object as an iText PDF object from the body of the PDF document. But let's start by fetching the trailer dictionary.

In line 4, we get the trailer dictionary using the `getTrailer()` method. We take a look at its entries the same way we looked a the entries of other dictionaries in section 1.2.6.

The `showEntries()` method produces the following output:

```
/Root: 762 0 R
/ID: [8Ã¯2õg©¬Ô , 8Ã¯2õg©¬Ô ]
/Size: 764
/Info: 763 0 R
```

In line 6 of code sample 3.1, we use the same `get()` as in the `showEntries()` method to obtain the value of the `/Size` entry. As we expect a number, we cast the `PdfObject` to a `PdfNumber` instance. We'll get a `ClassCastException` if the value of the entry is of a different type. The same exception will be thrown if the entry is missing in the dictionary, in which case the `get()` method will return `null`.

One way to avoid `ClassCastException` problems, is to get the value as a `PdfObject` instance first and to check whether or not it's `null`. If it's not, we can check the type before casting the `PdfObject` to one of its subclasses. An alternative to this convoluted method sequence would be to use one of the `getAsX()` methods listed in table 3.1.

Table 3.1: Overview of the getters available in PdfArray and PdfDictionary

Method name	Return type
`get()` / `getPdfObject()`	a `PdfObject` instance (could even be an indirect reference). The `get()` method is to be used for entries in a `PdfDictionary`; the `getPdfObject()` for elements in a `PdfArray`.
`getDirectObject()`	a `PdfObject` instance. Indirect references will be resolved. In case the value of an entry is referenced, `PdfReader` will go and fetch the `PdfObject` using that reference. You'll get a direct object, or `null` if the object can't be found.
`getAsBoolean()`	a `PdfBoolean` instance.
`getAsNumber()`	a `PdfNumber` instance.
`getAsString()`	a `PdfString` instance.
`getAsName()`	a `PdfName` instance.
`getAsArray()`	a `PdfArray` instance.
`getAsDict()`	a `PdfDictionary` instance.
`getAsStream()`	a `PdfStream` instance, that can be cast to a `PRStream` object.
`getAsIndirectObject()`	a `PdfIndirectReference` instance, that can be cast to a `PRIndirectReference` object.

These methods either return a specific subclass of `PdfObject`, or they return `null` if the object was of a different type or missing. In line 8 of code sample 3.1, we get a `PdfNumber` by using `trailer.getAsNumber(PdfName.SIZE);`

Suppose that we had used the `getAsString()` method instead of the `getAsNumber()` method. This would have returned `null` because the size isn't expressed as a `PdfString` value. This behavior is useful in case you don't know the type of the value for a specific entry in advance. For instance, when we'll talk about *named destinations* in section 35.2.1.1, we'll see that a named destination can be defined using either a `PdfString` or a `PdfName`. We could use the `getAsName()` method as well as the `getAsString()` method and check which method doesn't return `null` to determine which flavor of named destination we're dealing with.

When invoked on a `PdfDictionary`, the methods listed in table 3.1 require a `PdfName` —the key— as parameter; when invoked on a `PdfArray`, they require an `int` —the index. In line 10 of code sample 3.1, we get the `/ID` entry as a `PdfArray`, and we get the two elements of the array using the `getAsString()` method and the indexes `0` and `1`.

In line 15, we ask for the `/Info` entry, but the info dictionary isn't stored in the trailer dictionary as a direct object. The entry in the trailer dictionary refers to an indirect object with number `763`. If we want the actual dictionary, we need to use the `getAsDict()` method. This method will look at the object number of the indirect reference and fetch the corresponding indirect object from the `PdfReader` instance.

Take a look at the output of the `showObject()` methods in line 16 and line 17 to see the difference:

```
com.itextpdf.text.pdf.PRIndirectReference:
-> type: 10
-> toString: 763 0 R
com.itextpdf.text.pdf.PdfDictionary:
-> type: 6
-> toString: Dictionary
```

The `get()` method returns the reference, the `getAsDict()` method returns the actual object by fetching the content of object 763. Note that the reference instance is of type `PRIndirectReference`.

The `PdfStream` and `PdfIndirectReference` objects have `PRStream` and `PRIndirectReference` subclasses. The prefix `PR` refers to `PdfReader` and the object instances contain more information than the object instances we've discussed in chapter 1. For instance: if you want to extract the content of a stream, you'll need the `PRStream` instance instead of the `PdfStream` object.

On line 18, we try a slightly different approach. First, we get the indirect reference value of the `/Info` dictionary using the `getAsIndirectReferenceObject()` method. Then we get the actual object from the `PdfReader` by using the reference number. `PdfReader`'s `getPdfObject()` method can give you every object stored in the body of a PDF file by its number. `PdfReader` will fetch the byte position of the indirect object from the cross-reference table and parse the object found at that specific byte offset.

As an alternative, you can also use `PdfReader`'s static `getPdfObject()` method that accepts a `PdfObject` instance as parameter. If this parameter is an indirect reference, the reference will be resolved. If it's a direct object, that object will be returned as-is.

Now that we've played with different objects obtained from a `PdfReader` instance, let's explore the document structure using code. Looking at what RUPS shows us, the `/Root` dictionary aka the Document Catalog dictionary is where we should start. This dictionary has two required entries. One is the `/Type` which must be `/Catalog`. The other is the `/Pages` entry which refers to the root of the page tree.

We'll look at the optional entries in a moment, but let's begin by looking at the page tree.

3.3 Examining the page tree

Every page in a PDF document is defined using a `/Page` dictionary. These dictionaries are stored in a structure known as the page tree. Each `/Page` dictionary is the child of a page tree node, which is a dictionary of type

/Pages. One could work with a single page tree node, the one that is referred to from the catalog, but that would be bad practice. The performance of PDF consumers can be optimised by contructing a balanced tree.

If you create a PDF using iText, you won't have more than 10 /Page leaves or /Pages branches attached to every /Pages node. By design, a new intermediary page tree node is introduced by iText every 10 pages.

Before we start coding, let's take a look at figure 3.3. It shows part of the page tree of the primes.pdf document using RUPS, starting with the root node.

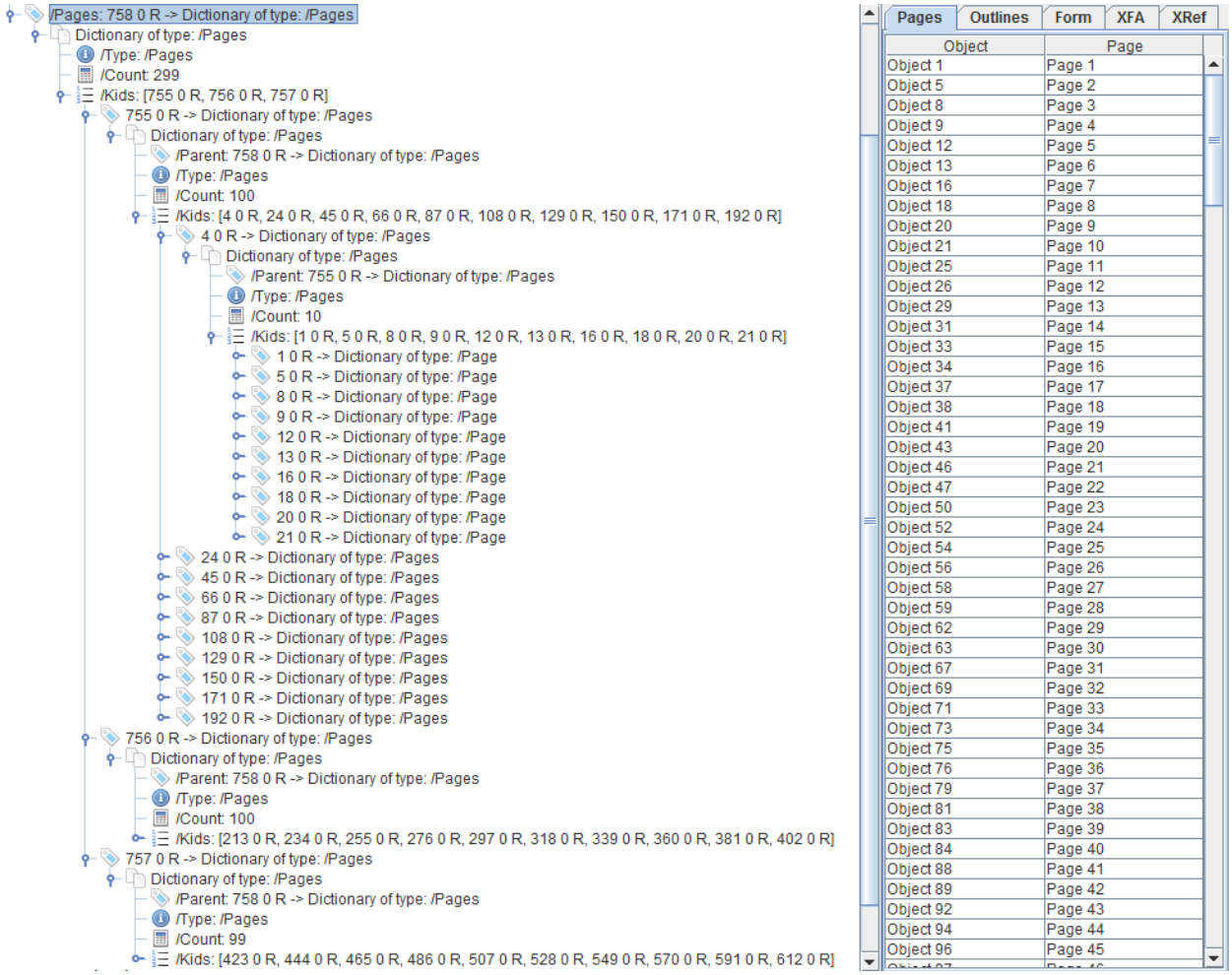


Figure 3.3: The page tree of primes.pdf opened in RUPS

The/Count entry of a page tree node shows the total number of leaf nodes attached directly or indirectly to this branch. The root of the page tree (object 758) shows that the document has 299 pages. The /Kids entry is an array with references to three other page tree nodes (objects 755, 756 and 757). The 299 leaves are nicely distributed over these three branches: 100, 100 and 99 pages. Each branch or leaf requires a /Parent entry referring to its parent; for the root node, the /Parent entry is forbidden.

When we expand the first page tree node, we discover that this tree node has ten branches. The first of these ten page tree nodes (object 4) has ten leaves, each leaf being a dictionary of type `/Page`. If you look to the panel at the right, you see that we've selected the **Pages** tab. This tab shows a table in which every row represents a page in the document. In the first column, you'll find the object number of a `/Page` dictionary; in the second column, you'll find its page number.

3.3.1 Page Labels

A page object on itself doesn't know anything about its page number. The page number of a page is calculated based on the occurrence of the page dictionary in the page tree. In figure 3.3, RUPS has examined the page tree, and attributed numbers going from 1 to 299.

If the Catalog has a `/PageLabels` entry, viewers can present a different numbering, for instance using latin numbering, such as *i, ii, iii, iv*, etc... It's important to understand that page labels and even page numbers are completely independent from the number that may or may not be visible on the actual page. Both the page number and its label only serve as an extra info when browsing the document in a viewer. You won't see any of these page labels on the printed document.

Figure 3.4 shows an example of a PDF file with a `/PageLabels` entry.

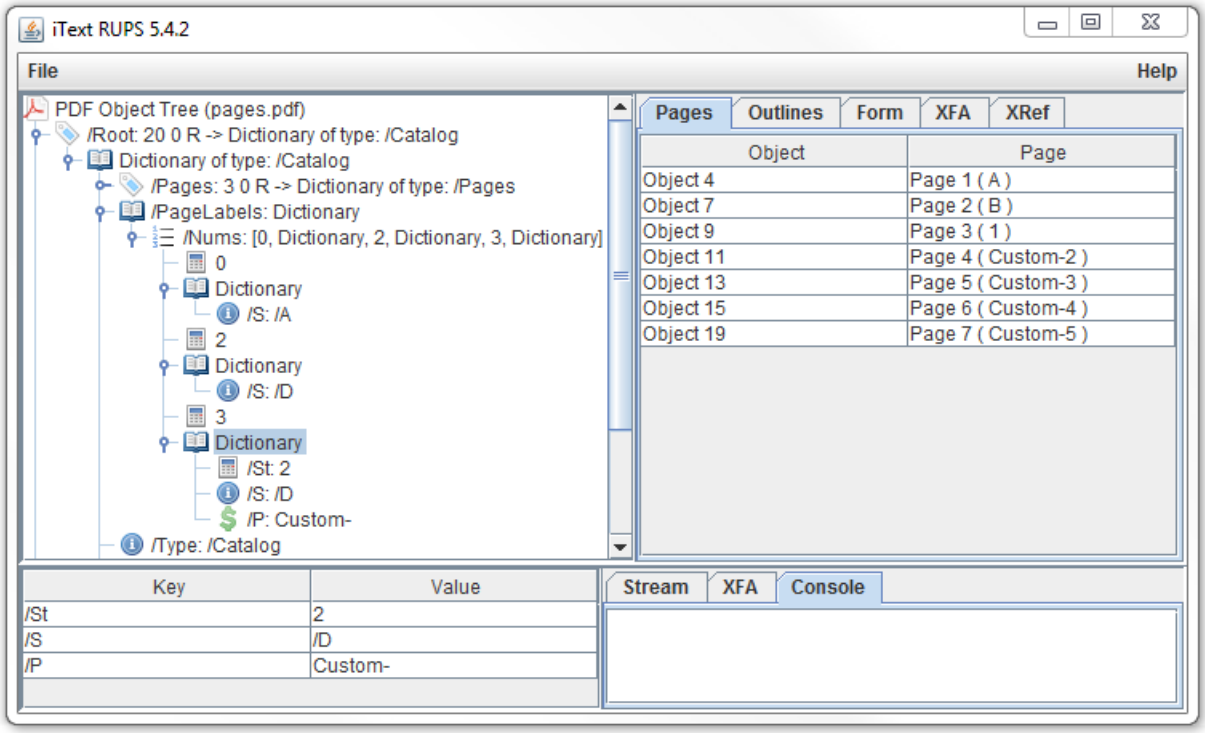


Figure 3.4: Using page labels

The value of the `/PageLabels` entry is a number tree.

What is a number tree?

A number tree serves a similar purpose as a dictionary, associating keys and values, but the keys are numbers, they are ordered, and a structure similar to the page tree (involving branches and leaves) can be used. The leaves are stored in an array that looks like this `[key1 value1 key2 value2 ... keyN valueN]` where the keys are numbers sorted in numerical order and the values are either references to a string, array, dictionary or stream, or direct objects in case of null, boolean, number or name values. See also section 3.5.1 for the definition of a name tree.

In the case of a number tree defining page labels, you always need a `0` key for the first page. The value of each entry will be a page label dictionary. Table 3.2 lists the possible entries of such a dictionary.

Table 3.2: Entries in a page label dictionary

Key	Type	Value
Type	name	Optional value: `/PageLabel`
S	name	The numbering style: - `/D` for decimal, - `/R` for upper-case roman numerals, - `/r` for lower-case roman numerals, - `/A` for upper-case letters, - `/a` for lower-case letters. In case of letters, the pages go from A to Z, then continue from AA to ZZ. If the `/S` entry is missing, page numbers will be omitted.
P	string	A prefix for page labels.
St	number	The first page number —or its equivalent— for the current page label range.

Looking at figure 3.4, we see three page label ranges:

1. *Index `0` (page 1)*— the page labels consist of upper-case letters,
2. *Index 2 (page 3)*— the page labels consist of decimals. As we've started a new range, the numbering restarts at 1. This means that page 3 will get "1" as page label.
3. *Index 3 (page 4)*— the page labels consist of decimals, but starts with label "2" (as defined in the `/St` entry). It also introduces a prefix (`/P`): "Custom-".

When opened in a PDF viewer, the pages of this document will be numbered *A*, *B*, *1*, *Custom-2*, *Custom-3*, *Custom-4*, and *Custom-5*. Talking about page labels was fun, but now let's find out how to obtain a page dictionary based on its sequence in the page tree.

3.3.2 Walking through the page tree

Code sample 3.2 shows how we could walk through the page tree to find all the pages in a document. This time we get the Catalog straight from the `reader` instance using the `getCatalog()` method instead of using `trailer.getAsDict(PdfName.ROOT)`. Once we have the Catalog, we get the `/Pages` entry, and pass it to the `expand()` method.

Code sample 3.2: C0302_PageTree

```
public static void main(String[] args) throws IOException {
    PdfReader reader
        = new PdfReader("src/main/resources/primes.pdf");
    PdfDictionary dict = reader.getCatalog();
    PdfDictionary pageroot = dict.getAsDict(PdfName.PAGES);
    new C0302_PageTree().expand(pageroot);
}

private int page = 1;
public void expand(PdfDictionary dict) {
    if (dict == null)
        return;
    PdfIndirectReference ref = dict.getAsIndirectObject(PdfName.PARENT);
    if (dict.isPage()) {
        System.out.println("Child of " + ref + ": PAGE " + (page++));
    }
    else if (dict.isPages()) {
        if (ref ==  null)
            System.out.println("PAGES ROOT");
        else
            System.out.println("Child of " + ref + ": PAGES");
        PdfArray kids = dict.getAsArray(PdfName.KIDS);
        System.out.println(kids);
        if (kids != null) {
            for (int i = 0; i < kids.size(); i++) {
                expand(kids.getAsDict(i));
            }
        }
    }
}
```

The `C0302_PageTree` example has a single private member variable `page` that is initialized at `1`. This variable is used in the recursive `expand()` method:

- If the dictionary passed to the method is of type `/Page`, the `isPage()` method will return `true`, and we'll increment the page number, writing it to the `System.out` along with info about the parent.
- If the dictionary passed to the method is of type `/Pages`, the `isPages()` method will return `true`, and we'll loop over all the `/Kids` array, calling the `expand()` method recursively for every branch or leaf.

The output of code sample 3.2 is consistent with what we saw in figure 3.3:

```
PAGES ROOT
[755 0 R, 756 0 R, 757 0 R]
Child of 758 0 R: PAGES
[4 0 R,24 0 R,45 0 R,66 0 R,87 0 R,108 0 R,129 0 R,150 0 R,171 0 R,192 0 R]
Child of 755 0 R: PAGES
[1 0 R,5 0 R,8 0 R,9 0 R,12 0 R,13 0 R,16 0 R,18 0 R,20 0 R,21 0 R]
Child of 4 0 R: PAGE 1
Child of 4 0 R: PAGE 2
Child of 4 0 R: PAGE 3
Child of 4 0 R: PAGE 4
Child of 4 0 R: PAGE 5
Child of 4 0 R: PAGE 6
Child of 4 0 R: PAGE 7
Child of 4 0 R: PAGE 8
Child of 4 0 R: PAGE 9
Child of 4 0 R: PAGE 10
Child of 755 0 R: PAGES
[25 0 R,26 0 R,29 0 R,31 0 R,33 0 R,34 0 R,37 0 R,38 0 R,41 0 R,43 0 R]
Child of 24 0 R: PAGE 11
Child of 24 0 R: PAGE 12
...
```

This is one way to obtain the `/Page` dictionary of a certain page. Fortunately, there's a more straight-forward method. In code sample 3.3, the `getNumberOfPages()` method provides us with the total number of pages. We loop from 1 to that number and use the `getPageN()` method to get the `/Page` dictionary for each separate page.

Code sample 3.3: C0303_PageTree

```
1  int n = reader.getNumberOfPages();
2  PdfDictionary page;
3  for (int i = 1; i <= n; i++) {
4      page = reader.getPageN(i);
5      System.out.println("The parent of page " + i + " is " + page.get(PdfName.PARENT));
6  }
```

The output of this code snippet corresponds with what we had before:

```
The parent of page 1 is 4 0 R
The parent of page 2 is 4 0 R
The parent of page 3 is 4 0 R
The parent of page 4 is 4 0 R
The parent of page 5 is 4 0 R
The parent of page 6 is 4 0 R
The parent of page 7 is 4 0 R
The parent of page 8 is 4 0 R
The parent of page 9 is 4 0 R
The parent of page 10 is 4 0 R
The parent of page 11 is 24 0 R
The parent of page 12 is 24 0 R
...
```

ISO-32000-1 and -2 define many possible entries for the `/Page` dictionary. It would lead us too far to discuss them all, but let's take a look at the most important ones.

3.4 Examining a page dictionary

Every `/Page` dictionary specifies the attributes of a single page. Table 3.3 lists the *required* entries in the `/Page` dictionary.

Table 3.3: Required entries in a page dictionary

Key	Type	Value
Type	name	Must be `/Page`
Parent	name	The page tree node that is the immediate parent of the page.
Resources	string	A dictionary containing any resources needed for the page. If the page doesn't require resources,an empty dictionary must be present. We'll discuss the possible entries in section 3.4.1.2.
MediaBox	rectangle	A rectangle defining the page size: the physical boundaries on which the page shall be displayed or printed. Other (optional) boundaries will be discussed in section 3.4.2.2.

The actual content of the page is stored in the `/Content` entry. This entry isn't listed in table 3.3 because it isn't required. If it's missing, the page is blank.

The value of the `/Contents` entry can either be a reference to a stream or an array. If it's an array, the elements consist of references to streams that need to be concatenated when rendering the page content.

3.4.1 The content stream and its resources

We'll discuss the syntax needed to describe the content in Part 2, but let's already peek at the content of the `/Contents` entry.

3.4.1.1 The content stream of a page

In code sample 3.4, we get the value of the `/Contents` entry as a `PRStream`; a `PdfStream` wouldn't be sufficient to get the stream bytes. `PdfReader` has two types of static methods to extract the contents of a stream: `getStreamBytesRaw()` gets the original bytes; `getStreamBytes()` returns the uncompressed bytes.

Code sample 3.4: C0304_PageContent

```
1 PdfDictionary page = reader.getPageN(1);
2 PRStream contents = (PRStream)page.getAsStream(PdfName.CONTENTS);
3 byte[] bytes = PdfReader.getStreamBytes(contents);
4 System.out.println(new String(bytes));
```

The first page of the document we're parsing has two paragraphs: *Hello World* and *Hello People*. You can easily recognize these sentences in the output that is produced by code sample 3.4:

```
q
BT
36 806 Td
0 -18 Td
/F1 12 Tf
(Hello World )Tj
0 0 Td
0 -18 Td
(Hello People )Tj
0 0 Td
ET
Q
```

Don't worry about the syntax. Every operator and operand will be explained in chapter 4, entitled "Graphics state" —for example: `q` and the `Q` are graphics state operators— and chapter 5, entitled "Text State" —`BT` and `ET` are text state operators.

The second page looks identical to the first page when opening the document in a PDF viewer. Internally there's a huge difference.

Code sample 3.5 shows a short-cut method to get the content stream of a page.

Code sample 3.5: C0304_PageContent

```
1 bytes = reader.getPageContent(2);
2 System.out.println(new String(bytes));
```

The resulting stream looks like this:

```
q
BT
36 806 Td
ET
Q
BT
/F1 12 Tf
88.66 788 Td
(ld)Tj
-22 0 Td
(Wor)Tj
-15.33 0 Td
(llo)Tj
-15.33 0 Td
(He)Tj
ET
q 1 0 0 1 36 763 cm /Xf1 Do Q
```

We still recognize the text *Hello World*, but it's mangled into *ld*, *Wor*, *llo* and *He*. Many PDFs aren't created in an ideal way. You're bound to encounter documents of which the content stream doesn't contain the exact words you expect. You shouldn't expect the content stream of a PDF to be human-readable.

Looking at the output of code sample 3.5, you notice that the worlds *Hello People* seem to be missing from this content stream. This text snippet is added as an *external object* or *XObject* marked as `/Xf1`. We'll find this object in the `/XObject` entry of the page resources.

3.4.1.2 The Resources dictionary

In code snippet 3.6, we get the resources from the page dictionary, and we list all of its entries.

Code sample 3.6: C0304_PageContent

```
page = reader.getPageN(2);
PdfDictionary resources = page.getAsDict(PdfName.RESOURCES);
for (PdfName key : resources.getKeys()) {
    System.out.print(key);
    System.out.print(": ");
    System.out.println(resources.getDirectObject(key));
}
```

The output of this code snippet looks like this:

```
/XObject: Dictionary
/Font: Dictionary
```

If we'd examine these dictionaries, we'd find the key `/Xf1` refering to a Form XObject in the former, and the key `/F1` refering to a font in the latter. We recognize references to these keys in the content stream resulting from code snippet 3.5.

Code sample 3.7 lists the entries and the content stream of the XObject with name `/Xf1` (line 2) and the entries of the font dictionary with name `/F1` (line 9):

Code sample 3.7: C0304_PageContent

```
1  PdfDictionary xObjects = resources.getAsDict(PdfName.XOBJECT);
2  PRStream xObject = (PRStream)xObjects.getAsStream(new PdfName("Xf1"));
3  for (PdfName key : xObject.getKeys()) {
4      System.out.println(key + ": " + xObject.getDirectObject(key));
5  }
6  bytes = PdfReader.getStreamBytes(xObject);
7  System.out.println(new String(bytes));
8  PdfDictionary fonts = resources.getAsDict(PdfName.FONT);
9  PdfDictionary font = fonts.getAsDict(new PdfName("F1"));
10 for (PdfName key : font.getKeys()) {
11     System.out.println(key + ": " + font.getDirectObject(key));
12 }
```

The resulting output for the XObject looks like this:

```
/Matrix: [1, 0, 0, 1, 0, 0]
/Filter: /FlateDecode
/Type: /XObject
/FormType: 1
/Length: 48
/Resources: Dictionary
/Subtype: /Form
/BBox: [0, 0, 250, 25]
BT
/F1 12 Tf
0 7 Td
(Hello People )Tj
ET
```

Now we clearly see *Hello People* in the content stream of the external object. This form XObject also contains a resources dictionary. We'll discuss the entries of the XObject's stream dictionary in more detail in part 2.

The resulting output for the font looks like this:

```
/Type: /Font
/BaseFont: /Helvetica
/Subtype: /Type1
/Encoding: /WinAnsiEncoding
```

A trained eye immediately notices that this dictionary represents the standard Type1 font Helvetica, a font that doesn't need to be embedded as it's one of the 14 standard Type 1 fonts. In part 2, we'll find out more about the different fonts in a PDF.

Table 3.4 offers the complete list of entries one can encounter in a `/Resources` dictionary.

Table 3.4: Possible entries in a resources dictionary

Key	Type	Value
Font	dictionary	A dictionary that maps resource names to font dictionaries.
XObject	dictionary	A dictionary that maps resource names to external objects.
ExtGState	dictionary	A dictionary that maps resource names to graphics state parameter dictionaries.
ColorSpace	dictionary	A dictionary that maps each resource name to either the name of a device-dependent colorspace or an array describing a color space.
Pattern	dictionary	A dictionary that maps resource names to pattern objects.
Shading	dictionary	A dictionary that maps resource names to shading dictionaries.
Properties	dictionary	A dictionary that maps resource names to property list dictionaries for marked content.
ProcSet	array	A feature that became obsolete in PDF 1.4 and that can safely be ignored.

Table 3.4 contains plenty of concepts that need further explaining, but we'll have to wait until part 2 before we can discuss them. Let's move on for now and take a look at page boundaries.

3.4.2 Page boundaries and sizes

The size of a page is stored in the `/MediaBox`, which was an entry listed as required in table 3.3. You can get this value as a `PdfArray` using `pageDict.getAsArray(PdfName.MEDIABOX);` but a more programmer-friendly way is shown in code sample 3.8.

Code sample 3.8: C0305_PageBoundaries

```
1  PdfReader reader =
2      new PdfReader("src/main/resources/pages.pdf");
3  show(reader.getPageSize(1));
4  show(reader.getPageSize(3));
5  show(reader.getPageSizeWithRotation(3));
6  show(reader.getPageSize(4));
7  show(reader.getPageSizeWithRotation(4));
```

We see two convenience methods, named `getPageSize()` and `getPageSizeWithRotation()`. These methods return a `Rectangle` object that is passed to the `show()` method (see code sample 3.9).

Code sample 3.9: C0305_PageBoundaries

```
1  public static void show(Rectangle rect) {
2      System.out.print("llx: ");
3      System.out.print(rect.getLeft());
4      System.out.print(", lly: ");
5      System.out.print(rect.getBottom());
6      System.out.print(", urx: ");
7      System.out.print(rect.getRight());
8      System.out.print(", lly: ");
9      System.out.print(rect.getTop());
10     System.out.print(", rotation: ");
11     System.out.println(rect.getRotation());
12 }
```

Let's discuss the difference between `getPageSize()` and `getPageSizeWithRotation()` by examining the output of code sample 3.8:

```
llx: 0.0, lly: 0.0, urx: 595.0, lly: 842.0, rotation: 0
llx: 0.0, lly: 0.0, urx: 595.0, lly: 842.0, rotation: 0
llx: 0.0, lly: 0.0, urx: 842.0, lly: 595.0, rotation: 90
llx: 0.0, lly: 0.0, urx: 842.0, lly: 595.0, rotation: 0
llx: 0.0, lly: 0.0, urx: 842.0, lly: 595.0, rotation: 0
```

The first page in the `pages.pdf` document is an A4 page in portrait orientation. If you'd extract the `/MediaBox` array, you'd get `[0 0 595 842]`.

3.4.2.1 Pages in landscape

Page 3 (see line 4 in code sample 3.8) is also an A4 page, but it's oriented in landscape. The `/MediaBox` entry is identical to the one used for the first page `[0 0 595 842]`, and that's why `getPageSize()` returns

the same result. The page is in landscape, because the `\Rotate` entry in the page dictionary is set to `90`. Possible values for this entry are `0` (which is the default value if the entry is missing), `90`, `180` and `270`. The `getPageSizeWithRotation()` method takes this value into account. It swaps the width and the height so that you're aware of the difference. It also gives you the value of the `/Rotate` entry.

Page 4 also has a landscape orientation, but in this case, the rotation is mimicked by adapting the `/MediaBox` entry. In this case the value of the `/MediaBox` is `[0 0 842 595]` and if there's a `/Rotate` entry, its value is `0`. That explains why the output of the `getPageSizeWithRotation()` method is identical to the output of the `getPageSize()` method.

3.4.2.2 The CropBox and other page boundaries

Looking at figure 3.5, we see the page labels we've discussed in section 3.3.1, we see the different orientations discussed in section 3.4.2.1, and we see something strange happening to the *Hello World* text on page 5.

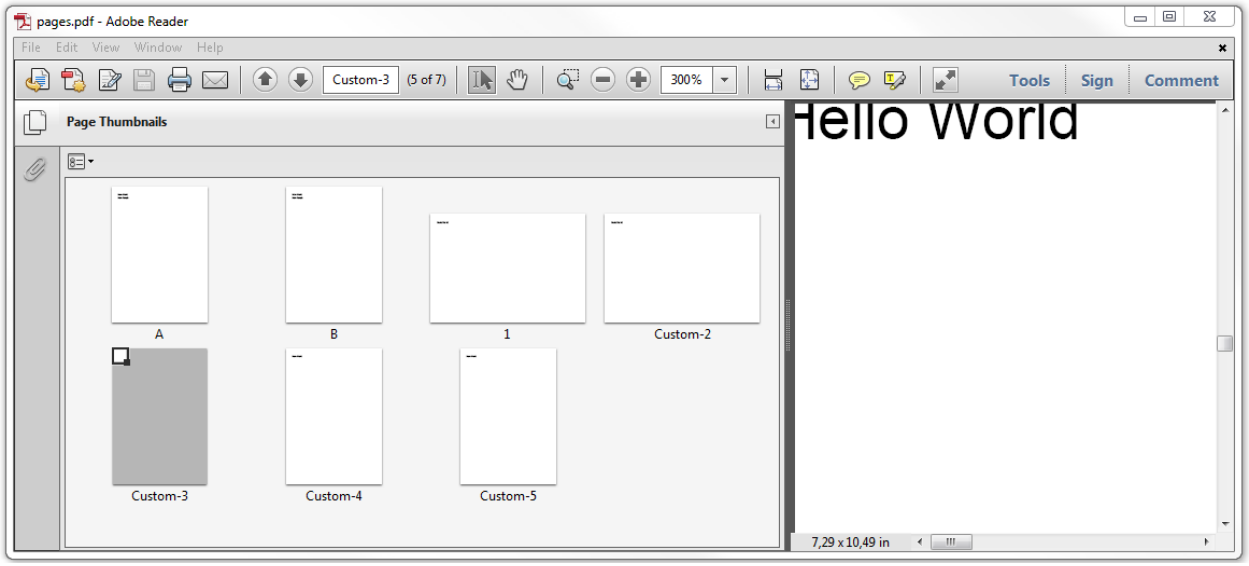


Figure 3.5: Pages and page boundaries

The text is cropped because there's a `/CropBox` entry in the page dictionary. As shown in code sample 3.10, you can get the cropbox using the `getCropBox()` method.

Code sample 3.10: C0305_PageBoundaries

```
1 show(reader.getPageSize(5));
2 show(reader.getCropBox(5));
```

These two lines result in the following output:

```
llx: 0.0, lly: 0.0, urx: 595.0, lly: 842.0, rotation: 0
llx: 40.0, lly: 40.0, urx: 565.0, lly: 795.0, rotation: 0
```

Let's consult ISO-32000-1 or -2 to find out the difference between the `/MediaBox` returned by the `getPageSize()` method and the `/CropBox` returned by the `getCropBox()` method.

- *The media box*— defines the boundaries of the physical medium on which the page is to be printed. It may include any extended area surrounding the finished page for bleed, printing marks, or other such purposes. It may also include areas close to the edges of the medium that cannot be marked because of physical limitations of the output device. Content falling outside this boundary may safely be discarded without affecting the meaning of the PDF file.
- *The crop box*— defines the region to which the contents of the page shall be clipped (cropped) when displayed or printed. Unlike the other boxes, the crop box has no defined meaning in terms of physical page geometry or intended use; it merely imposes clipping on the page contents. However, in the absence of additional information, the crop box determines how the page's contents shall be positioned on the output medium.

Summarized: the media box is the working area on your page, but only the content inside the crop box will be visible.

FAQ: I've added content to a page and it isn't visible

Maybe you're adding the content outside the visible area. The lower-left corner of the rectangle defining the media box and the crop box (if present) doesn't necessarily correspond with the coordinate `x = 0; y = 0`. You could easily have a media box defined like this: `[595 842 1190 1684]`. This is still an A4 page, but if you add a watermark at the coordinate `x = 397.5; y = 421`, that watermark won't be visible as it's outside the visible area of the page. If `rect` is the visible area of a page, you could center the watermark using these formulas for `x` and `y`:

```
float x = rect.getLeft() + rect.getWidth() / 2f;
float y = rect.getBottom() + rect.getHeight() / 2f;
```

These formulas calculate the coordinate of the middle of the page.

The definition of the media box and the crop box mention that a page can have some other boxes too:

- *The bleed box*— defines the region to which the contents of the page shall be clipped when output in a production environment. This may include any extra bleed area needed to accommodate the physical limitations of cutting, folding, and trimming equipment. The actual printed page may include printing marks that fall outside the bleed box.
- *The trim box*— defines the intended dimensions of the finished page after trimming. It may be smaller than the media box to allow for production-related content, such as printing instructions, cut marks, or color bars.
- *The art box*— defines the extent of the page's meaningful content (including potential white space) as intended by the page's creator.

If present, you'll find these boxes in the page dictionary by the following names: `/Bleedbox`, `/TrimBox` and `/ArtBox`. Code sample 3.11 shows how to obtain the media box and art box of page 7:

Code sample 3.11: C0305_PageBoundaries

```
show(reader.getBoxSize(7, "media"));
show(reader.getBoxSize(7, "art"));
```

The resulting output looks like this:

```
llx: 0.0, lly: 0.0, urx: 595.0, lly: 842.0, rotation: 0
llx: 36.0, lly: 36.0, urx: 559.0, lly: 806.0, rotation: 0
```

The `getBoxSize()` method accepts the following values for the `boxName` parameter: `media`, `crop`, `bleed`, `trim` and `art`. As you probably noticed, it can be used as an alternative for the `getPageSize()` and `getCropBox()` method.

All boxes, except for the media box can be visualized in an interactive PDF processor. This is done using a box color information dictionaries for each boundary. These box color information dictionaries are stored in the `/BoxColorInfo` entry of the page dictionary.

3.4.2.3 The measurement unit

Let's consult the PDF specification to find out what it says about the measurement unit:

ISO-32000-1 states: *The default for the size of the unit in default user space (1 / 72 inch) is approximately the same as a point, a unit widely used in the printing industry. It is not exactly the same, however; there is no universal definition of a point.* In short: 1 in. = 25.4 mm = 72 user units (which roughly corresponds to 72 pt).

The media box of the pages we've discussed so far was `[0 0 595 842]`. By default, this corresponds with the standard page size A4, measuring 210 by 297 mm (or 8.27 by 11.69 in), but that's not always the case. Figure 3.6 shows pages 5 and 6 of the `pages.pdf` document.

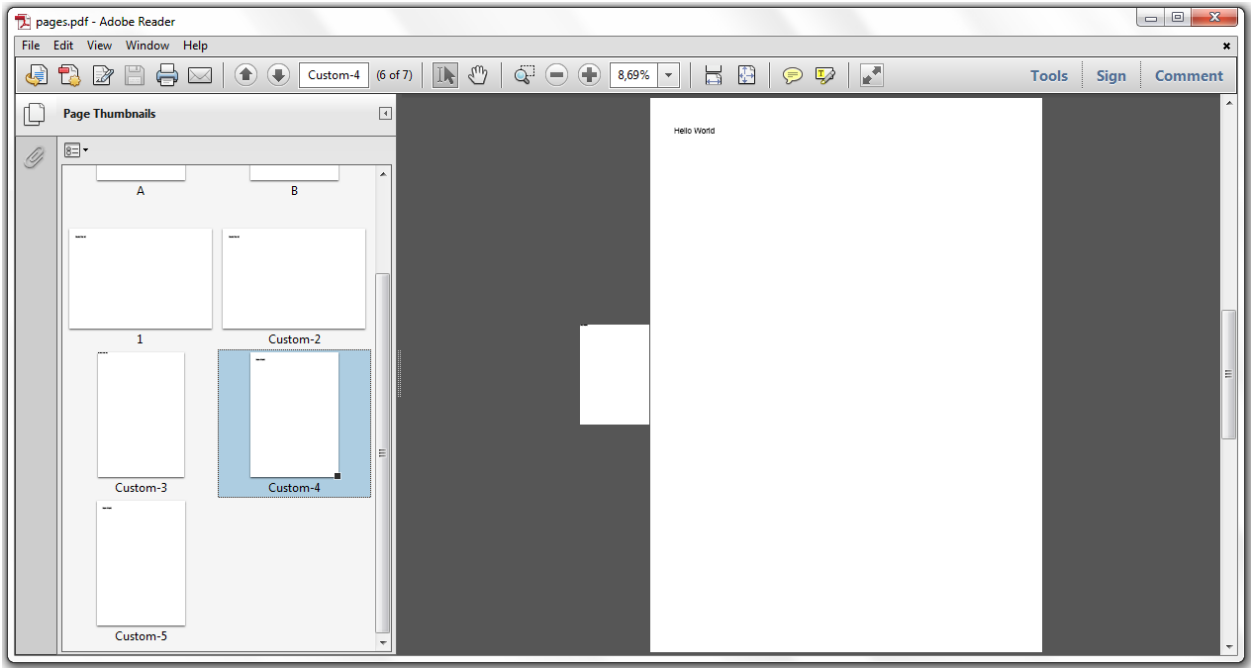


Figure 3.6: Pages with a different user unit

If we'd examine page 6, we'd discover that the media box is the only page boundary. It's defined as`[0 0 595 842]`, but when we look at the document properties, we see that the page size is 41.32 by 58.47 in. which is about 5 times bigger than the A4 page we expected. This difference is caused by the fact that a different user unit was used. In code sample 3.12, we get the `/UserUnit` value from the page dictionary of page 6. The output of this code sample is indeed 5.

Code sample 3.12: C0305_PageBoundaries

```
1 PdfDictionary page6 = reader.getPageN(6);
2 System.out.println(page6.getAsNumber(PdfName.USERUNIT));
```

Theoretically, you could create pages of any size, but the PDF specification imposes limits depending on the PDF version of the document. Table 3.5 shows how the implementation limits changed throughout the history of PDF.

Table 3.5: Possible entries in a resources dictionary

PDF version	Minimum page size	Maximum page size
PDF 1.3 and earlier	72 x 72 units (1 x 1 in.)	3240 x 3240 units (45 x 45 in.)
PDF 1.4 and later	3 x 3 units (approx. 0.04 x 0.04 in.)	14,400 x 14,400 units (200 x 200 in.)

Changing the user unit has been possible since PDF 1.6. The minimum value of the user unit is 1 (this is the default; 1 unit = 1/72 in.); the maximum value as defined in PDF 1.7 is 75,000 points (1 unit = 1042 in.).

The PDF ISO standards don't restrict the range of supported values for the user unit, but says that the value is implementation-dependent.

Let's conclude that the biggest PDF page you can create without hitting any viewer implementations measures 15,000,000 x 15,000,000 in or 381 x 381 km. That's almost 5 times the size of Belgium (the country where iText was born).

3.4.3 Annotations

Figure 3.7 shows the last page in the `pages.pdf` document. If you hover over the words *Hello World*, a link to *http://maps.google.com* appears. If you click the small page icon, a post-it saying *This is a post-it annotation* pops up. These are annotations.

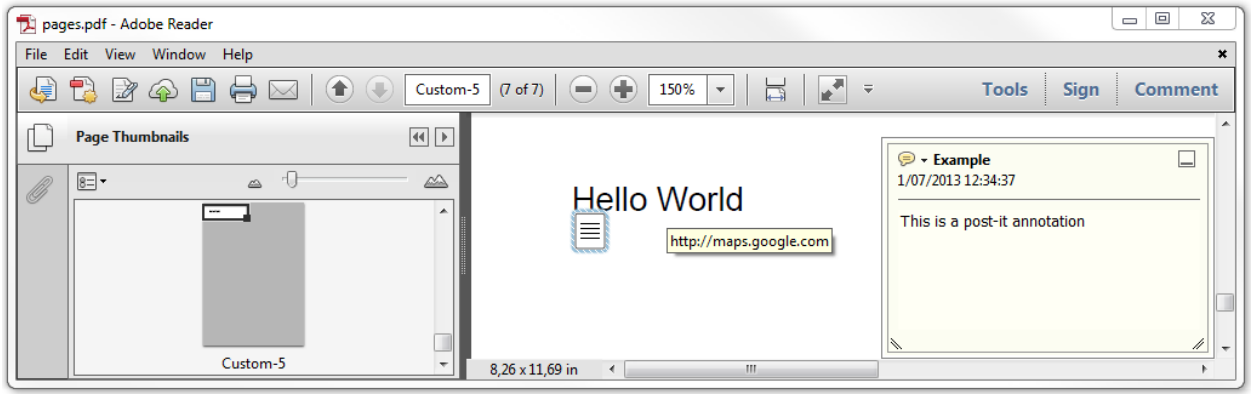


Figure 3.7: Annotations

If a page contains annotations, you'll find them in the `/Annots` entry of the page dictionary. This is an array of annotation dictionaries.

In figure 3.7, we're dealing with a `/Text` and a `/Link` annotation. Other types include *line*, *stamp*, *rich media*, *watermark*, and many other annotations. We'll get a closer look at all of these types in chapter 7. For now, it's sufficient to know that an annotation is an interactive object associated with a page.

Annotations aren't part of the content stream of a page. PDF viewers will always render visible annotations on top of all the other content of a page.

Code sample 3.13 shows how to obtain the annotation dictionaries from page 7's page dictionary.

Code sample 3.13: C0306_PageAnnotations

```
PdfArray annots = page.getAsArray(PdfName.ANNOTS);
for (int i = 0; i < annots.size(); i++) {
    System.out.println("Annotation " + (i + 1));
    showEntries(annots.getAsDict(i));
}
```

We reuse the `showEntries` method from code sample 3.1, and this gives us the following result:

```
Annotation 1
/Contents: This is a post-it annotation
/Subtype: /Text
/Rect: [36, 768, 56, 788]
/T: Example
Annotation 2
/C: [0, 0, 1]
/Border: [0, 0, 0]
/A: Dictionary
/Subtype: /Link
/Rect: [66.67, 785.52, 98, 796.62]
```

The first annotation is a simple text annotation with title *Example* and contents *This is a post-it annotation*. The `/Rect` entry defines the coordinates of the clickable icon.

The second one is a link annotation. The `/C` entry defines the color of the border, but as the third element of the `/Border` array is `0`, no border is shown. You'll learn all about the different properties available for annotations in chapter 7.

Code sample 3.14 allows us to obtain the keys of the value of the `/A` entry, an action dictionary.

Code sample 3.14: C0306_PageAnnotations

```
PdfDictionary link = annots.getAsDict(1);
showEntries(link.getAsDict(PdfName.A));
```

The resulting output looks like this:

```
/URI: http://maps.google.com
/S: /URI
```

The action is of type `/URI`. You can trigger it by clicking the rectangle defined by `/Rect` in the link annotation.

Let's take a look at some other possible entries of the page dictionary before we return to the document catalog.

3.4.4 Other entries of the page dictionary

Table 3.6 lists entries of the page dictionary we haven't discussed so far.

Table 3.6: Optional entries of the page dictionary

Key	Type	Description
Metadata	stream	A stream containing XML metadata for the page in XMP format. This entry is also available in the document catalog. We'll discuss it in the next section.
AF	array	References to embedded files associated with this page. This entry is also available in the document catalog. We'll discuss this in the next section.
AA	dictionary	Additional actions to be performed when the page is opened or closed. This entry is also available in the document catalog. We'll discuss it in the next section.
StructParents	integer	Required if the page contains structural items. We'll discuss this in chapter 6.
Tabs	name	A name containing the tab order used for the annotations on the page. Possible values are: - `/R` for row order, - `/C` for column order, - `/S` for structure order, - `/A` for annotations array order (PDF 2.0) and - `/W` for widget order (PDF 2.0).
Thumb	stream	A stream object that defines the page's thumbnail image.
VP	array	An array of viewport dictionaries. This is outside the scope of this book.
Dur	number	The page's display duration in seconds during presentations.
Trans	dictionary	A transition dictionary describing the effect when opening the page in a viewer.
PZ	number	The preferred zoom factor for the page. For more info, see next section.
PresSteps	dictionary	Used in the context of sub-page navigation while presenting a PDF.
PieceInfo	dictionary	A page piece dictionary for the page. This entry is also available in the document catalog. We'll briefly discuss it in the next section.
LastModified	date	Required if **PieceInfo** is present. Contains the date and time when the page's contents were recently modified.
B	array	An array containing indirect references to article beads.
DPart	dictionary	An indirect reference to a DPart dictionary.
OutputIntents	array	This entry is also available in the document catalog.
SeparationInfo	dictionary	Information needed to generate color separations for the page.

Table 3.6: Optional entries of the page dictionary

Key	Type	Description
Group	dictionary	Used in the context of transparency group XObject.
ID	byte string	A digital identifier used in the context of Web Capture.
TemplateInstantiated	name	Required if this page was created from a named page object, in which case it's the name of the originating page object.

Many of these entries are outside the scope of this book, or can only be described briefly here, please consult ISO-32000-1 or -2 for more info. Now it's high time to take a closer look at the document catalog.

3.5 Optional entries of the Document Catalog Dictionary

We've already used the `getCatalog()` method in section 3.3.2 to get the root of the page tree. We learned that the document catalog has two required entries, `/Type` and `/Pages`. In this section, we'll discuss the optional entries of the root dictionary.

3.5.1 The names dictionary

We were already introduced to the concept of a number tree when we discussed page labels in section 3.3.1. When the keys of such a tree structure consist of strings instead of numbers, we talk about a name tree.

What is a name tree?

A name tree serves a similar purpose to a dictionary, associating keys and values. Based on the fact that we call this structure a name tree, you may expect that the keys are names, but they aren't. The keys are strings, they are ordered, and they are structured as a tree (involving branches and leaves). The leaves are stored in an array that looks like this `[key1 value1 key2 value2 ... keyN valueN]` where the keys are strings sorted in alphabetical order and the values are either references to a string, array, dictionary or stream, or direct objects in case of null, boolean, number or name values.

A name tree —or a number tree for that matter— usually consists of the following elements:

- The *root node* is a dictionary that refers to intermediate or leaf nodes in its `/Kids` entry. Alternatively, it can have a `/Names` entry in the case of a name tree, and a `/Nums` entry in the case of a number tree.
- An *intermediate node* is a dictionary with a `/Kids` entry referring to an array of intermediate nodes or leave nodes.
- A *leaf node* is a dictionary that has a `/Names` entry in the case of a name tree, and a `/Nums` entry in the case of a number tree.

Each intermediate and leaf node also has a `/Limits` entry, which is an array of two strings in case of a name tree, and an array of two integers in the case of a number tree. The elements of the array specify the least and greatest keys present in the node or its sub-nodes.

The document's name dictionary, which is the `/Names` entry of the document catalog, refers to one or more name trees. Each entry in the name dictionary is a name tree for a specific category of objects that can be referred to by name rather than by object reference. These are the most important categories:

- `/Dests`— maps name strings to destinations. We'll learn more about this in the next section.
- `/AP`— maps name strings to annotation appearance streams. We've already seen some simple annotations in section 3.4.3. In chapter 7, we'll create a custom appearance for some more complex annotations. Such an appearance could be referred to by name.
- `/JavaScript`— maps name strings to document-level JavaScript actions. We've learned about a simple URI action, but soon we'll find out that we can also use JavaScript to program custom actions.
- `/EmbeddedFiles`— maps name strings to file specifications for embedded file streams.

If you study ISO-32000-1 or -2, you'll discover that there are more categories, but they are out of scope of this book. In the next section, we'll take a look at a name tree containing named destinations.

3.5.2 Document navigation and actions

In section 3.3, we've examined the page tree, and we've learned how to navigate through a document programmatically. Now we're going to take a look at some ways we can help the end user navigate through the document using a PDF viewer.

3.5.2.1 Destinations

There are different ways to define destinations. You can associate names with destinations, or you can use explicit destinations.

3.5.2.1.1 Named destinations When we discussed the name dictionary, we listed `/Dests` as one possible category of named objects. This name tree defines *named destinations* with string values as keys.

The concept of using a name tree for named destinations was introduced in PDF 1.2. In PDF 1.1, named destinations were defined using names instead of string. They were stored in a `/Dests` entry of the document catalog. It referred to a dictionary with the names as keys and their destination dictionary as values. Most of the PDF producers have abandoned using this Catalog entry in favor of the `/Dests` entry in the name tree.

Code sample 3.15 shows how we get the first named destination from the name tree in the name dictionary.

Code sample 3.15: C0306_DestinationsOutlines

```
PdfReader reader = new PdfReader("src/main/resources/primes.pdf");
PdfDictionary catalog = reader.getCatalog();
PdfDictionary names = catalog.getAsDict(PdfName.NAMES);
PdfDictionary dests = names.getAsDict(PdfName.DESTS);
PdfArray array = dests.getAsArray(PdfName.NAMES);
System.out.println(array.getAsString(0));
System.out.println(array.getAsArray(1));
```

This is the resulting output:

```
Prime101
[210 0 R, /XYZ, 36, 782, 0]
```

This means that we can now use the string `Prime101` as a name to refer to a destination on the page described by the indirect object with number 210. Code sample 3.16, shows a short-cut to get the same information using much less code:

Code sample 3.16: C0306_DestinationsOutlines

```
Map<String, String> map =
    SimpleNamedDestination.getNamedDestination(reader, false);
System.out.println(map.get("Prime101"));
```

The `SimpleNamedDestination` class can be used to obtain a map of all named destinations. If you use `false` as the second parameter of the `getNamedDestination()` method, you get the named destinations stored in the `/Dests` entry of the names dictionary —destinations referred to by strings (since PDF 1.2). If you use `true`, you get the named destinations stored in the `/Dests` entry —destinations referred to by names (PDF 1.1).

The `SimpleNamedDestination` class also has an `exportToXML()` method that allows you to export the named destinations to an XML file.

In code sample 3.15 and 3.16, the destination is defined using four parameters: the name `/XYZ` and three numbers, the x-position 36, the y-position 782, and the zoom factor 0 (no zoom). This is an explicit destination.

3.5.2.1.2 Explicit destinations Destinations are defined as an array consisting of a reference to a page dictionary and a location on a page, optionally including the zoom factor. Table 3.7 lists the different names and parameters that can be used to define the location and zoom factor.

Table 3.7: Destination syntax

Type	Extra parameters	Description
`/Fit`	-	The page is displayed with its contents magnified just enough to fit the document window, both horizontally and vertically.
`/FitB`	-	The page is displayed magnified just enough to fit the bounding box of the contents (the smallest rectangle enclosing all of its contents).
`/FitH`	*top*	The page is displayed so that the page fits within the document window horizontally (the entire width of the page is visible). The extra parameter specifies the vertical coordinate of the top edge of the page.
`/FitBH`	*top*	This option is almost identical to `/FitH`, but the with of the bounding box of the page is visible. This isn't necessarily the entire width of the page.
`/FitV`	*left*	The page is displayed so that the page fits within the document window vertically (the entire height of the page is visible). The extra parameter specifies the horizontal coordinate of the left edge of the page.
`/FitBV`	*left*	This option is almost identical to `/FitV`, but the height of the bounding box of the page is visible. This isn't necessarily the entire height of the page.
`/XYZ`	*left, top, zoom*	The *left* parameter defines an `x` coordinate, *top* defines a `y` coordinate, and *zoom* defines a zoom factor. If you want to keep the current `x` coordinate, the current `y` coordinate, or zoom factor, you can pass negative values or `0` for the corresponding parameter.
`/FitR`	*left, bottom, right, top*	The parameters define a rectangle. The page is displayed with its contents magnified just enough to fit this rectangle. If the required zoom factors for the horizontal and the vertical magnification are different, the smaller of the two is used.

Destinations may be associated with outline items, link annotations, or actions. Let's start with outlines.

3.5.2.2 The outline tree

Figure 3.8 shows a PDF with bookmarks for every prime number.

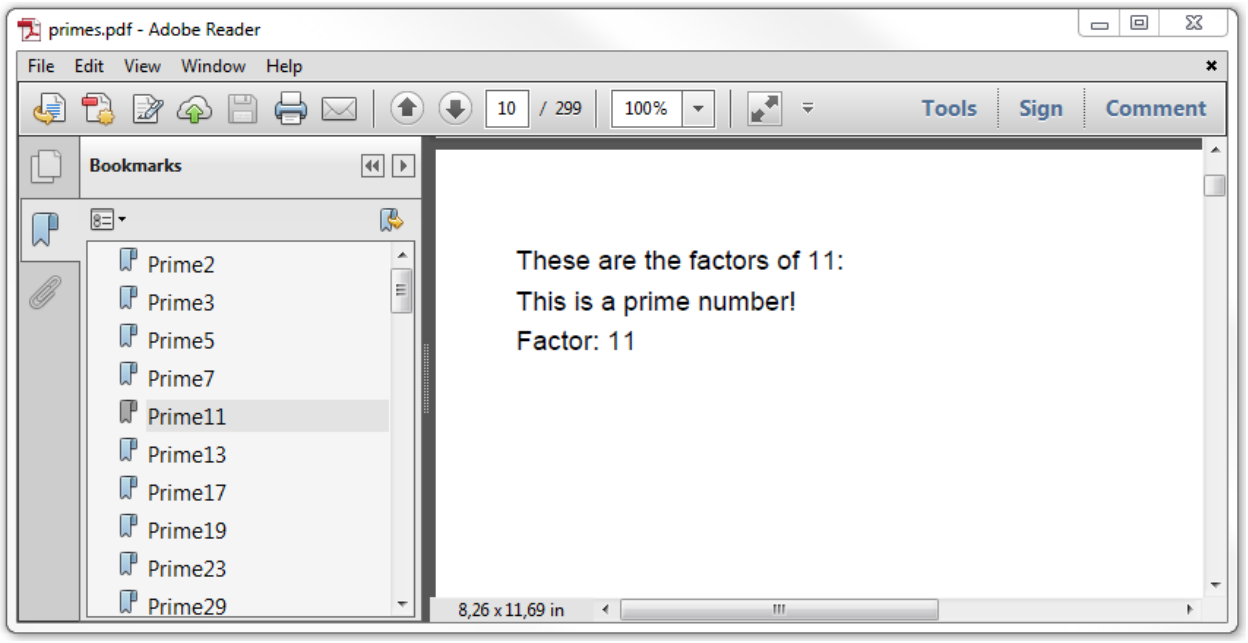


Figure 3.8: A PDF with bookmarks

Bookmarks can have a tree-structured hierarchy, but in this case, we only have a simple list of items. If we click an item, you either jump to a destination, which is the case if you click *Prime11*, or you trigger an action.

The root of this bookmarks tree is specified by the `/Outlines` entry of the document catalog. In code sample 3.17, we extract the root, its first leaf and its last leaf as a dictionary:

Code sample 3.17: C0306_DestinationsOutlines

```
1 PdfDictionary outlines = catalog.getAsDict(PdfName.OUTLINES);
2 System.out.println("Root:");
3 showEntries(outlines);
4 System.out.println("First:");
5 showEntries(outlines.getAsDict(PdfName.FIRST));
6 System.out.println("Last:");
7 showEntries(outlines.getAsDict(PdfName.LAST));
```

The output of this code sample looks like this:

```
Root:
/Type: /Outlines
/Count: 62
/Last: 754 0 R
/First: 693 0 R
First:
/Next: 694 0 R
/Parent: 692 0 R
/Title: Prime2
/Dest: [1 0 R, /Fit]
Last:
/Parent: 692 0 R
/Title: Prime293
/Dest: [614 0 R, /Fit]
/Prev: 753 0 R
```

This corresponds with what we see when we look at the outline tree using RUPS. See figure 3.9.

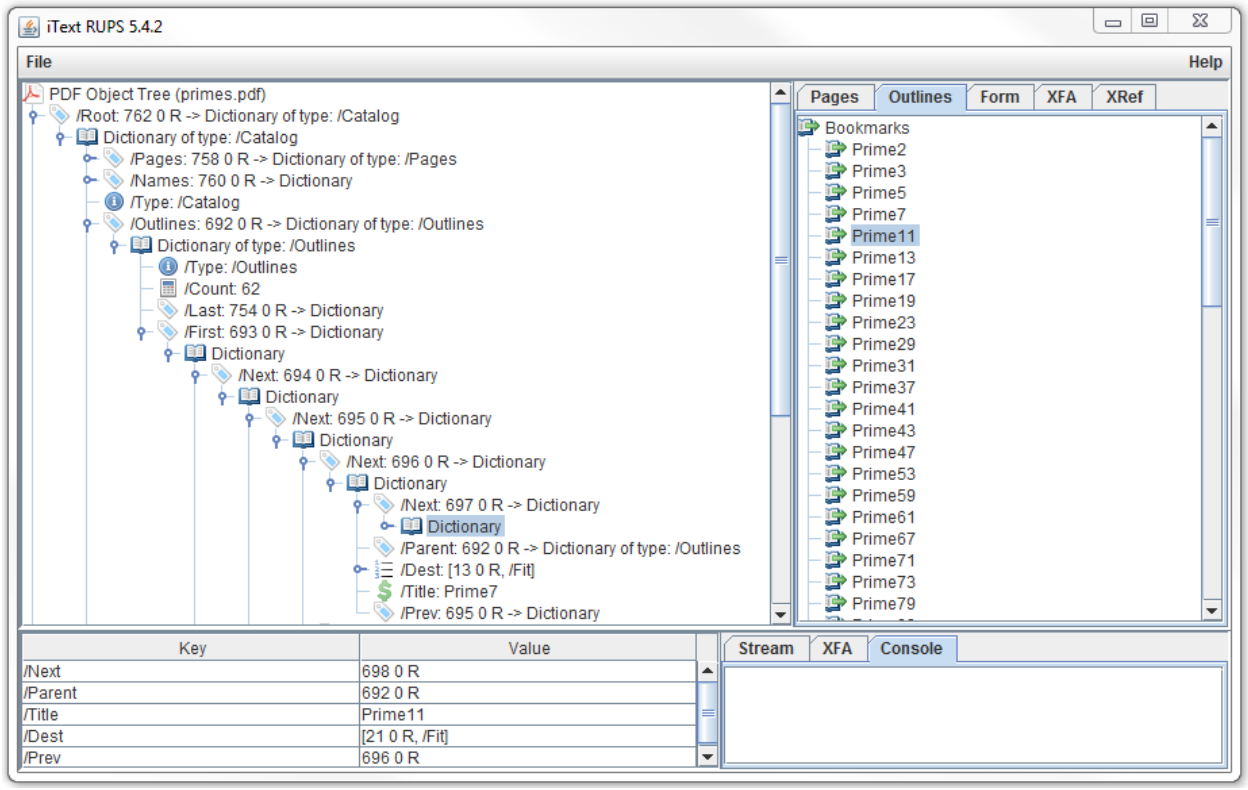


Figure 3.9: The outline tree

Note that all the entries at the same level are chained to each other as a linked list, with the `/Next` entry referring to the next item and the `/Prev` entry referring to the previous item.

The root of the outline tree is an outline dictionary. Table 3.8 explains the different entries.

Table 3.8: Entries in the outline dictionary

Key	Type	Value
Type	name	If present, the value must be `/Outlines`.
First	dictionary	An indirect reference to the first top-level outline item.
Last	dictionary	An indirect reference to the last top-level outline item.
Count	integer	The total number of open outline items. The value can't be negative. It's to be omitted if there are no open items.

Table 3.9 shows the possible entries for outline item dictionaries.

Table 3.9: Entries in the outline dictionary

Key	Type	Value
Title	string	The text for the outline as displayed in the bookmarks panel.
Parent	dictionary	The parent of this outline, this is the outline dictionary for top-level outline items, otherwise it's another outline item.
Prev	dictionary	The previous item at the current outline level. Not present for the first one.
Next	dictionary	The next item at the current outline level. Not present for the last one.
First	dictionary	An indirect reference to the first descendant of which this item is the parent.
Last	dictionary	An indirect reference to the last descendant of which this item is the parent.
Count	integer	The total number of open outline items. If the outline item is closed, a negative value is used with as absolute value the number of descendants that would be opened if the item was open.
Dest	name, string or array	In the case of named destinations a name or string will be used. In the case of explicit destinations an array is used. See section 3.5.2.1.
A	dictionary	The action that needs to be performed. See next section.
SE	dictionary	The structure item to which the outline refers; see chapter 6.
C	array	An array of three numbers ranging from `0.0` to `1.0` representing an RGB color.
F	integer	The style used for the text: by default `0` for regular text, `1` for italic, `2` for bold, `3` for bold and italic.

Just like with named destinations where we had the `SimpleNamedDestination` convenience class, there's also a `SimpleBookmark` class that allows you to get more information about bookmarks, without having to fetch

all the outline item dictionaries. Code sample 3.18 gives you an example of how to create a `List` containing all the bookmarks.

Code sample 3.18: C0306_DestinationsOutlines

```
List<HashMap<String, Object>> bookmarks = SimpleBookmark.getBookmark(reader);
for (HashMap<String, Object> item : bookmarks) {
    System.out.println(item);
}
```

The output for the `primes.pdf` starts like this:

```
{Action=GoTo, Page=1 Fit, Title=Prime2}
{Action=GoTo, Page=2 Fit, Title=Prime3}
{Action=GoTo, Page=4 Fit, Title=Prime5}
{Action=GoTo, Page=6 Fit, Title=Prime7}
{Action=GoTo, Page=10 Fit, Title=Prime11}
```

The map consists of strings such as `Open`, `Title`, `Page`, `Color`, `Style`, `Kids`, and so on. In the case of the `Kids` entry, the object is a `List` with more `HashMap` elements.

The `SimpleBookmark` class also has an `exportToXML()` method that allows you to export the bookmarks to an XML file.

Note that iText uses the keyword `Action` to indicate that clicking a bookmark item is the equivalent of a `GoTo` action.

Some outline actions, such as JavaScript actions, aren't picked up by `SimpleBookmark`.

Let's take a closer look at actions in general.

3.5.2.3 Actions

An action in a PDF document is defined using an action dictionary that specifies what the viewer should do when the action is triggered. Table 3.10 shows the entries common to all action dictionaries.

Table 3.10: Entries common to all action entries

Key	Type	Value
Type	name	If present, the value must be `/Action`.
S	name	The type of action, see table 3.11.
Next	dictionary or array	The next action or sequence of actions that must be performed after this action. This allows actions to be *chained*.

Table 3.11 shows the possible values for the /S entry.

Table 3.11: Action types

Action Type	Context	Description
Named	Nav.	Execute a predefined action.
GoTo	Nav.	Go to a destination in the current document.
GoToR	Nav.	Go to a destination in a remote document.
URI	Nav.	Resolve a uniform resource identifier (URI).
GoToE	Nav.	Go to a destination in an embedded PDF file.
GoToDp	Nav.	Go to a document part.
Hide	Ann.	Set an annotation's Hidden flag.
Sound	Ann.	Play a sound.
Movie	Ann.	Play a movie.
Rendition	Ann.	Controls playing of multimedia content.
GoTo3DView	Ann.	Sets the current view of a 3D annotation.
RichMediaExecute	Ann.	Specifies a command to be sent to a rich media annotation's handler.
SubmitForm	Form	Send form data to a uniform resource locator (URL).
ResetForm	Form	Reset the fields in a form to their default values.
ImportData	Form	Import field values from a file.
SetOCGState	OCG	Sets the states of optional content groups.
JavaScript	Misc.	Execute a JavaScript script.
Launch	Misc.	Launch an application, usually to open a file.
Trans	Misc.	Updates the display of a document using a transition dictionary.
Thread	Misc.	Begin reading an article thread.

We won't discuss all these types of action in detail. We'll look at some of the actions marked with *Nav.* which allow us to navigate through a document. We'll discuss some of the actions marked with *Ann.* triggering events related to annotations in chapter 7, and actions marked with *Form* in chapter 7. The OCG action will be discussed in chapter 6.

Actions can be triggered in many ways. For now we'll only look at two places where we can find actions.

- The /OpenAction entry in the catalog. Its value can be an array defining an explicit destination or an action dictionary. The action is triggered upon opening the document.
- The /AA entry that can occur in root, page, annotation and form field dictionaries. Its value is an *additional actions* dictionary defining actions that need to be executed in response to various trigger events.

We'll discuss the events that can be triggered from an annotation in chapter 8 and those that can be triggered from a form field in chapter 8. Table 3.12 shows the possible entries in the additional actions dictionary of a page dictionary.

Table 3.12: Entries in a page object's additional actions dictionary

Key	Type	Value
O	dictionary	An action that will be executed when the page is opened, for instance when the user navigates to it from the next or previous page.
C	dictionary	An action that will be executed when the page is closed, for instance when the user navigates away from it by clicking a link to another page.

Table 3.13 shows the possible entries in the additional actions dictionary of the document catalog.

Table 3.13: Entries in the document catalog's additional actions dictionary

Key	Type	Value
WC	dictionary	A JavaScript action to be executed before closing the document ("will close").
WS	dictionary	A JavaScript action to be executed before saving the document ("will save")
DS	dictionary	A JavaScript action to be executed after saving the document ("did save")
WP	dictionary	A JavaScript action to be executed before printing the document ("will print")
DP	dictionary	A JavaScript action to be executed after printing the document ("did print")

Just like an HTML file, a PDF document can contain JavaScript. There are some differences in the sense that you get extra objects that allow you to use functionality that is specific to a PDF viewer. We'll take a look at some JavaScript examples in chapter 7.

The "will save" action isn't triggered by a "Save As" operation. In practice this often means that it's only triggered in Adobe Acrobat, not in Adobe Reader.

Let's conclude this section about actions by looking at some of the actions marked with *Nav.* in table 3.11.

A *named action* is an action of which the value of the `/S` entry is `/Named`. It has an additional `/N` entry of which the value is one of the names listed in table 3.14.

Table 3.14: Named actions

Name	Action
NextPage	Go to the next page of the document.
PrevPage	Go to the previous page of the document.
FirstPage	Go to the first page of the document.
LastPage	Go to the last page of the document.

A *go to* action is an action of which the value of the /S entry is /GoTo. It must have a /D entry that can be a name or a string in case of named destinations, or an array in case of an explicit destination. Starting with PDF 2.0, there can also be an /SD entry to jump to a specific structure destination (see chapter 6).

The *remote go to* action is very similar. It's an action of which the value of the /S entry is /GoToR. It must have an /F entry of which the value is a file specification dictionary, as well as a /D entry. The value of the /D entry can be a name or a string in case of named destinations. When an explicit destination is defined, an array is used of which the first element isn't an indirect reference to a page dictionary, but instead the page number of the target destination.

There's an optional NewWindow entry of which the value is a Boolean. If true, the document will be opened in a new window, provided that the referring document is opened in a standalone PDF viewer. For instance: setting this entry to true won't open a new browser window if the document is opened in Adobe Reader's browser plug-in.

An *URI* action is an action of which the value of the /S entry is URI. It must have an URI entry defining the uniform resource identifier to resolve, for instance: a link to open a hyperlink in a browser.

When the Catalog of a document contains a /URI entry, it refers to a dictionary with a single entry, /Base, of which the value is the base URI that shall be used when resolving relative URI references throughout the document.

An URI action dictionary can have an /IsMap entry of which the value is a Boolean. If true, a mouse position will be added to the URI in the form of a query string containing the values of an x and a y coordinate.

Another way to send data to a server involves interactive forms.

3.5.3 Interactive forms

When you find an /AcroForm key in the root dictionary of a PDF file, your document is an interactive form. There are different flavors of forms in PDF.

1. One is based on *AcroForm* technology. These forms are defined using the PDF objects described in chapter 1, for instance an array of fields where each field is defined using a dictionary.
2. Another type of form uses the *XML Forms Architecture* (XFA). In this case, the PDF file acts as a container for a set of streams that form a single XML file defining the appearance as well as the data of the form.

3. Finally, there are also hybrid forms that contain both an AcroForm description and an XFA stream defining the form.

Forms are used for different purposes. One purpose can be to allow users to fill out a form manually and to submit the filled out data to a server. Another purpose could be to create a template that can be filled out in an automated process. For instance: one could create a PDF containing an AcroForm form that represents a voucher for an event. Such a PDF could have fields that act as placeholders for the name of an attendee, a date, a bar code, etc... These static placeholders can then be filled out automatically using data from a database. If you need dynamic fields, XFA is your only option. You could create an XFA template based on your own XML schema, and then inject different sets of XML data into this template.

We'll discuss these types of forms in more detail in chapter 8. For now, we'll just take a look at an example of each flavor and find out how to tell the different flavors apart.

3.5.3.1 AcroForm technology

Figure 3.10 shows an interactive form based on AcroForm technology. It contains text fields that can be filled out with a title of a movie, a director, etc. It also contains some check boxes and radio buttons. To the left, we see the same document opened in RUPS. We see that the Catalog has an `/AcroForm` entry. The value of the `/Fields` entry of this dictionary is an array that isn't empty. Its elements are AcroForm fields.

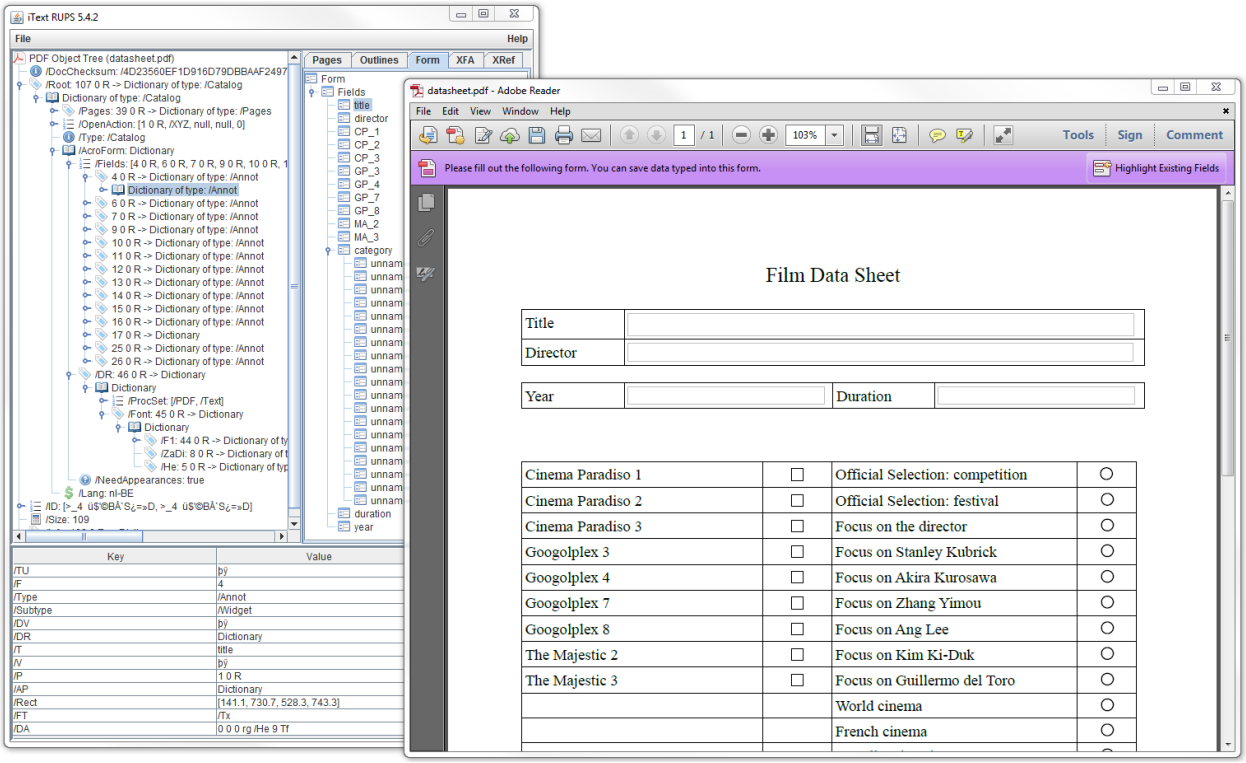


Figure 3.10: AcroForm technology

We selected one specific field in RUPS: the *title* field. In the dictionary panel, we see the combined field and annotation dictionary defining that field. The `/P` value refers to the page on which the widget annotation of

the field will appear; the /Rect entry defines the position of the field's annotation on that page. We'll discuss the other entries of this dictionary in chapter 8. For now, it's important to understand that every placeholder has its fixed place on a fixed page when using AcroForm technology. Placeholders can't resize automatically when their content doesn't fit the designated space.

We could test this form, by getting the AcroForm dictionary from the catalog, retrieving the fields array, and so on, but let's skip that, and use the AcroFields convenience class. See code sample 3.18.

Code sample 3.18: C0306_DestinationsOutlines

```
public static void inspectForm(File file) throws IOException {
    System.out.print(file.getName());
    System.out.print(": ");
    PdfReader reader = new PdfReader(file.getAbsolutePath());
    AcroFields form = reader.getAcroFields();
    XfaForm xfa = form.getXfa();
    System.out.println(xfa.isXfaPresent() ?
              form.getFields().size() == 0 ? "XFA form" : "Hybrid form"
            : form.getFields().size() == 0 ? "not a form" : "AcroForm");
    reader.close();
}
```

This code snippet can be used to check which type of form you're dealing with. It writes the name of the file to the System.out, followed by one of the following results: *XFA form*, *Hybrid form*, *not a form* or *AcroForm*. The AcroFields class inspects the /AcroForm entry, and checks if there's an XFA part. If not, it checks the number of fields. If there is no /AcroForm entry or if the /Fields array is empty, you don't have an AcroForm. In the example shown in figure 3.10, the method returns *AcroForm*.

The AcroForm technology supports four types of fields:

- *Button fields*— representing interactive controls on the screen that can be manipulate using a mouse, such as pushbuttons, check boxes and radio buttons.
- *Text fields*— consisting of boxes or spaces in which the users can enter text from the keyboard.
- *Choice fields*— containing several text items that can be selected, for instance list boxes and combo boxes.
- *Signature fields*— representing digital signatures and optional data for authenticating the name of the signer and the document's contents.

We'll cover button, text and choice fields in chapter 8, but digital signature fields are outside the scope of this book.

A digital signature involves a digital certificate, a timestamp, revocation information and so on. This information is needed as soon as you want to validate the signature. If this information is missing or expired, one can create an incremental update and add a dictionary to the document catalog containing the most up-to-date validation-related information. This dictionary is known as the Document Security Store (DSS) and it's used as the value for the /DSS entry of the root dictionary.

Digital signatures, including the Document Security Store, are discussed in great detail in the book "Sign your PDFs with iText[2]."

3.5.3.2 The XML Forms Architecture

Figure 3.11 shows such an example of a pure XFA form opened in Adobe Reader as well as in RUPS.

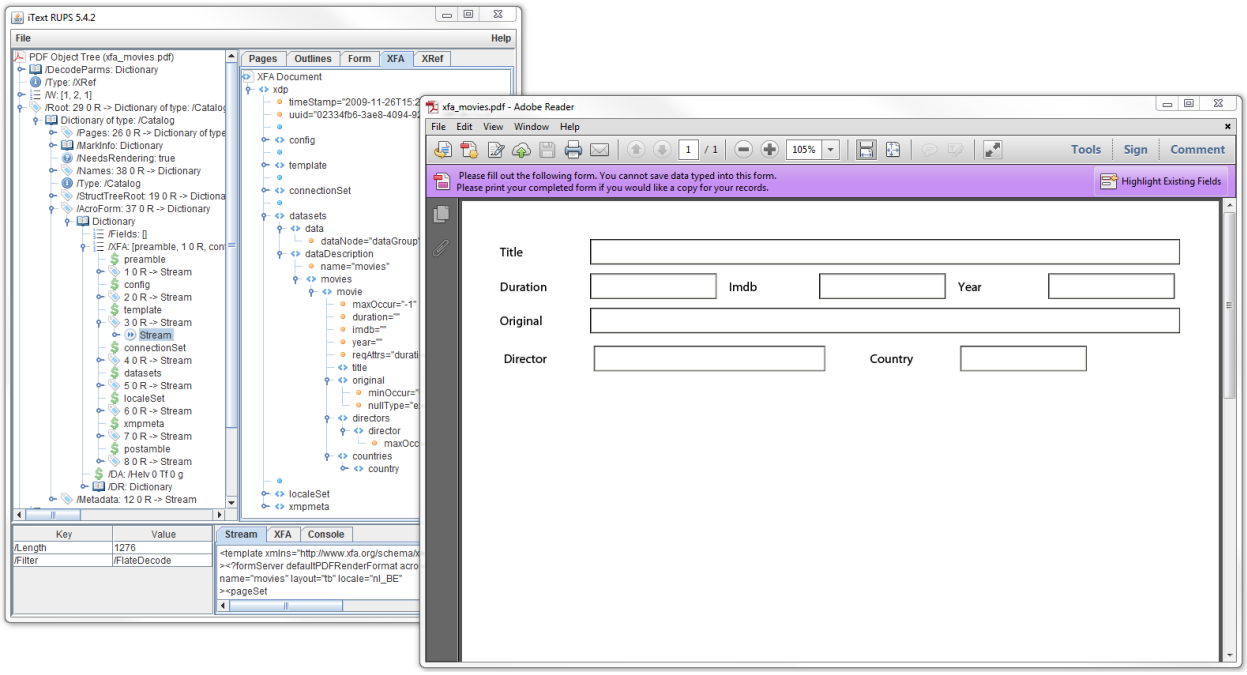


Figure 3.11: An empty XFA form

We detect a `/NeedsRendering` entry with value `true` in the Catalog. This means that the PDF viewer will have to parse XML in order to render the content of the document.

Not all PDF viewers support pure XFA forms. For instance: if you open a pure XFA form in Apple Preview, you'll see whatever content is available in the form of PDF syntax. This is usually a warning saying that your PDF viewer doesn't support this particular version of PDF documents.

The `/AcroForm` entry has an empty `/Fields` array. The form is defined in different streams that are to be concatenated to form a valid XML file.

The form shown in figure 3.11 is empty: it doesn't contain any data. Figure 3.12 shows the same form, but now filled out using an XML file that contains 120 movies. The document that originally counted only one page, now consists of 23 pages. The complete group of fields is repeated 120 times, once for each movie in the XML file. Some fields are repeated too, see for instance the *Country* field.

[2] https://leanpub.com/itext_pdfsign

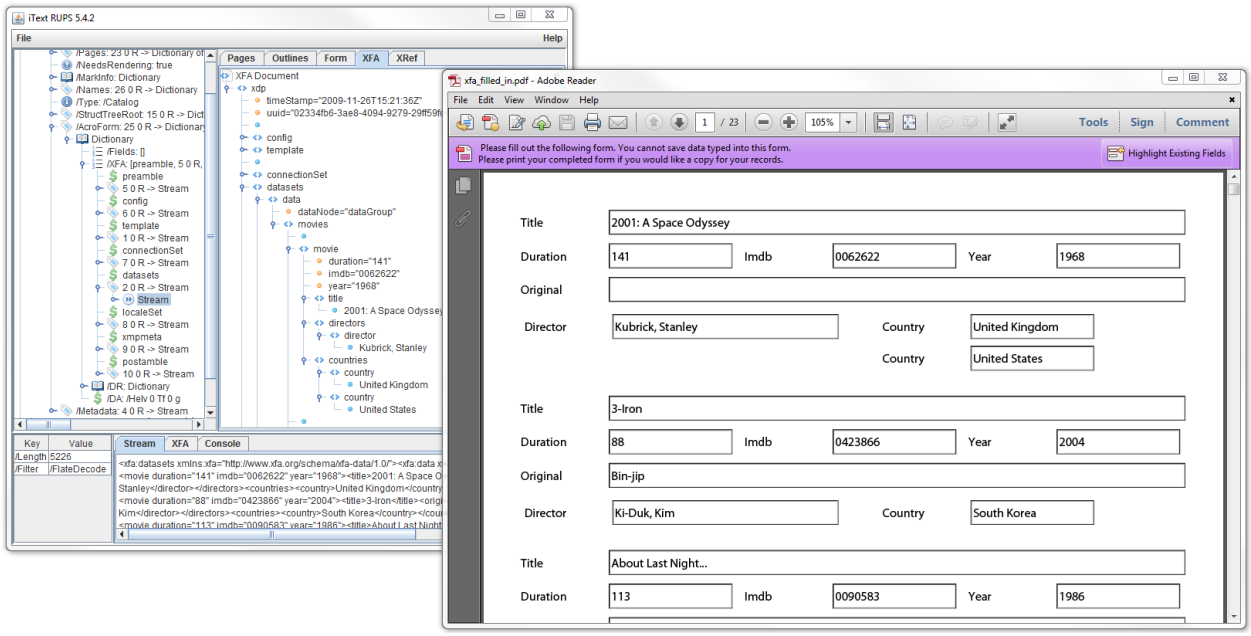


Figure 3.12: A filled out XFA form

As opposed to the static AcroForm documents, XFA forms are truly dynamic. You can use them for forms of templates of which you don't know the amount of data nor the size of the data up-front. XML forms also consist of text fields, choice fields, buttons and signature fields, but they are defined using XML, which provides much more flexibility.

3.5.3.3 Hybrid forms

Figure 3.13 shows a hybrid form. There's no `/NeedsRendering` entry in the Catalog. A PDF viewer doesn't need to parse any XML in order to render the form as the form is also defined using AcroForm technology

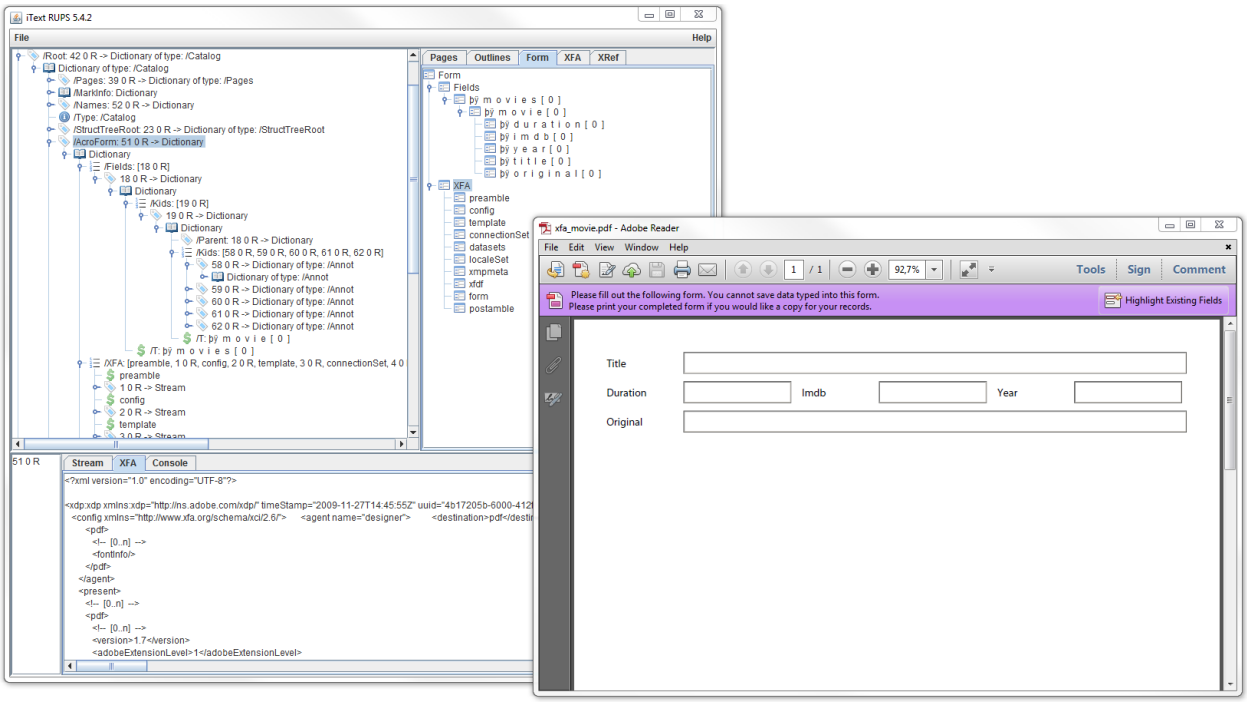


Figure 3.13: A hybrid form

Hybrid forms have the advantage that they can be viewed in PDF viewer that don't support XFA, but they have the disadvantage that the dynamic nature of XFA is lost.

3.5.4 Marked Content

In the next chapter, we'll take a closer look at the Adobe Imaging Model. We'll learn more about the operators that are needed to draw text, paths, shapes and images to a page. These operators only serve to make sure the visual representation of the document is correct.

There is also a set of operators that allow you to mark this content. Marked-content operators are used to identify a portion of a PDF content stream as an element of interest for a particular goal. For instance: a word drawn on a page doesn't necessarily know which line it belongs too. A line doesn't know which paragraph it belongs too. You can mark a sequence of text operators as a paragraph, so that software can discover the structure of your document. In this case, we add marked content operators to add structure to the document. When specific rules are followed when creating this structure, we say that the PDF is a Tagged PDF document.

FAQ: I've created a PDF with a table and now I want to extract the cells of that table

This is only possible if the PDF is tagged. If the document isn't tagged, it doesn't know there's a tabular structure on the page. What looks like a table to the human eye, is nothing more than a bunch of glyphs, lines and shapes drawn on a page. By introducing *marked content operators* into the content stream, you can mark all the different elements of a table in a way that software can detect rows, columns, headers, and so on.

Using marked content, you can also add object data. For example: if your PDF consists of a blueprint of a machine, you can use marked content operators to add specific properties for each machine part that is drawn. Marked content can also be used to make documents accessible. For instance: for each image in the document, you can add a short textual description so that people who are visually impaired can find out what the image represents. Another typical use case involves optional content. You can mark a sequence of PDF syntax in a way that it becomes visible or invisible, for instance depending on user interaction.

Several entries in the root dictionary refer to Marked Content. The `/MarkInfo` entry refers to a dictionary containing information about the document's use of Tagged PDF conventions. We'll discuss the `/StructTreeRoot` entry in detail in chapter 6. The same goes for the `/OCProperties` key which refers to optional content.

3.5.5 Embedded files

We've looked at the actual content of a document, at navigation information and at the structure of the content, but there's more. A document can also contain attachments, and these attachments can be organized in a special way.

There are different ways to attach a file to a PDF document. You can add an attachment using an annotation. In this case, you'll have a visible object on the page (for instance a paper clip) that can be clicked by the end user to open the attachment. We'll learn more about file attachment annotations in chapter 7.

You can also create document-level attachments also known as embedded files. You need to open the attachments panel in your PDF viewer to see these attachments. When double-clicking an attachment of a different format than PDF, you need an external viewer to open the attachment. See figure 3.14.

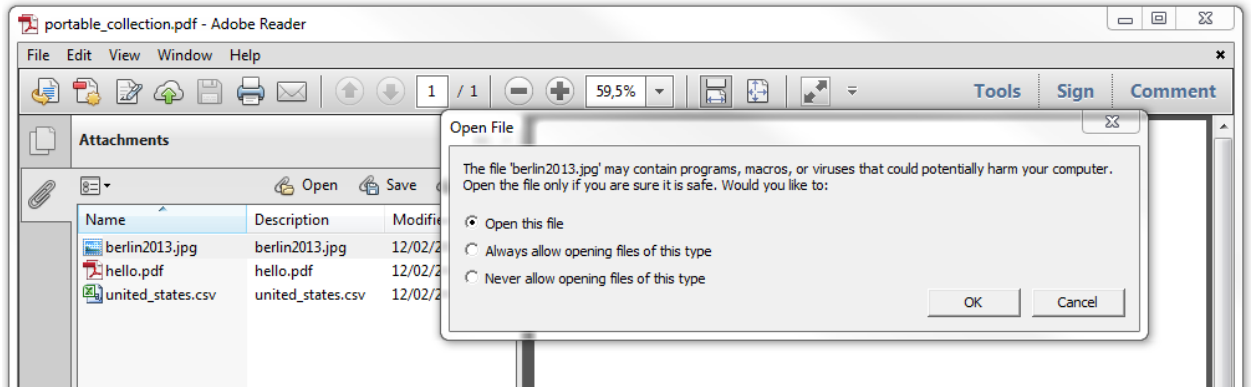


Figure 3.14: Document-level attachments

If you want an approach that is more integrated into the PDF Viewer, you may want to create a portable collection.

3.5.5.1 Portable Collections

Figure 3.15 shows the most simple type of portable collection you can create.

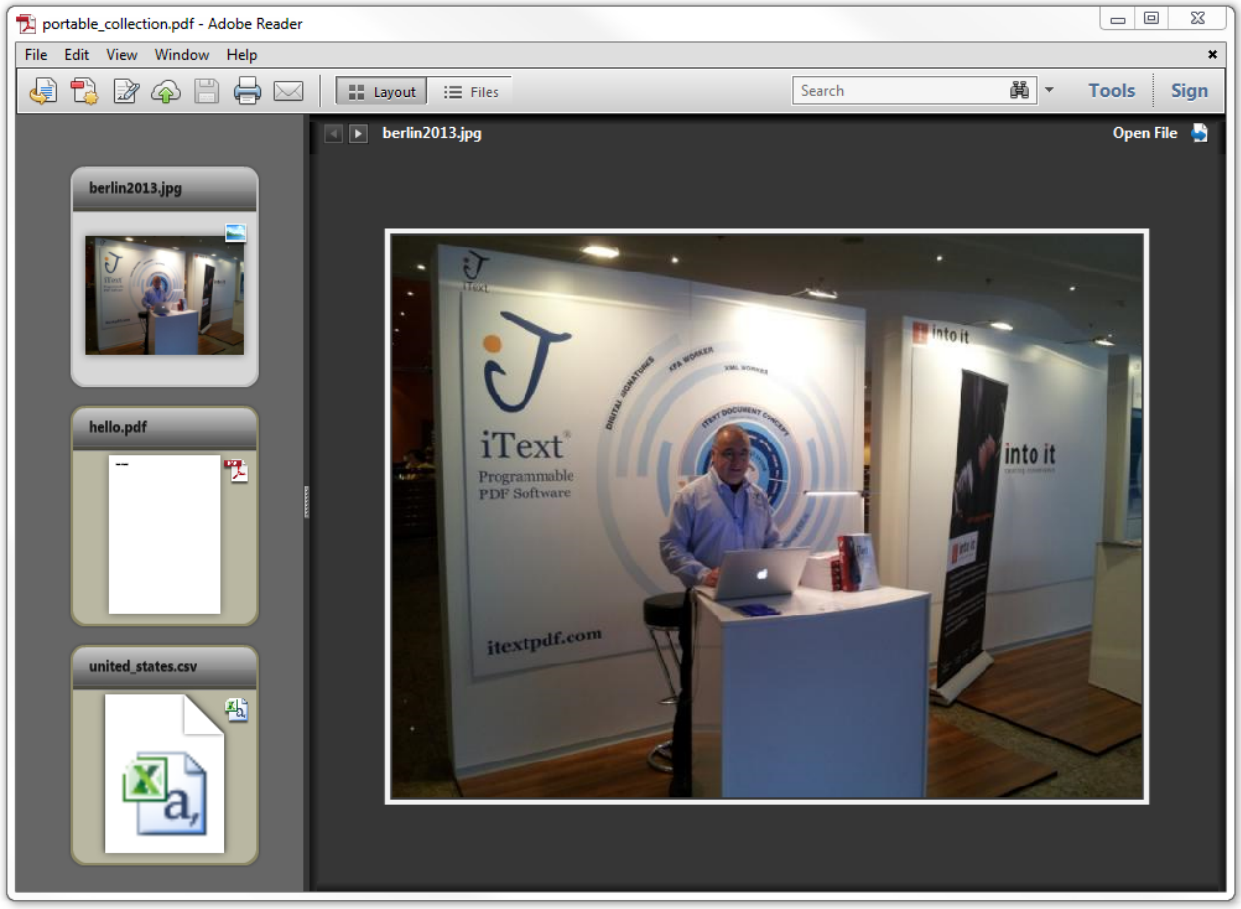


Figure 3.15: A portable collection

In this case, the PDF is defined as a portable collection.

When using a portable collection, the PDF acts as a container for different files, similar to a ZIP file. The advantage of having a PDF package over a ZIP file is the fact that some files can be rendered in Adobe Reader. For instance: you can view the JPG without having to open an external application. To open the CSV file however, you'll need an application such as Excel.

The difference between the PDF shown in figure 3.14 and the one shown in figure 1.15 consists of a single entry in the root dictionary.

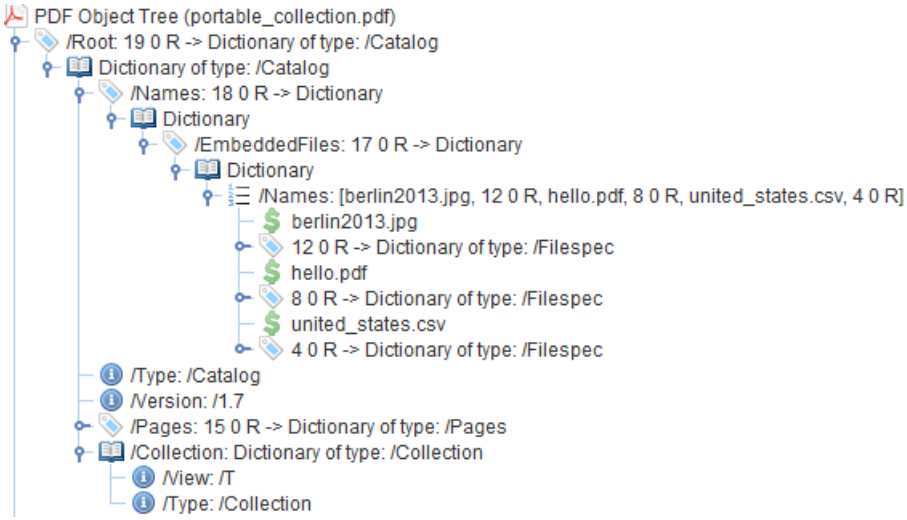


Figure 3.16: A portable collection

In both cases, there is a `/Names` entry with an `/EmbeddedFiels` name tree. In the second case, there is also a `/Collection` entry in the Catalog. In figure 3.16, the view type is `/T` for Tile. There are different types of portable collections. Instead of showing thumbnails, you can provide a table consisting of rows you can populate with data of your choice. You can also create your own Flash component to navigate through the different documents.

3.5.5.2 Associated files

When attaching files to a document, the PDF isn't aware of any relationship between the document and the attachment. To the document, the attachment is merely a sequence of bytes, unless you define an *associated files relationship*. See the `/AFRelationship` key in the filespecification of the file `united_states.csv` in figure 3.17.

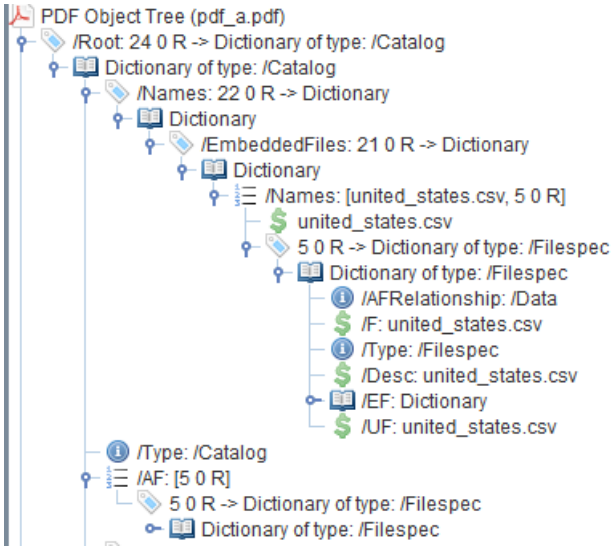


Figure 3.17: A portable collection

The file that is opened in RUPS contains a list of US states that was created based on the data file `united_states.csv`. We add an extra reference to this file specification in the array of associated files; see the value of the `/AF` key in the root dictionary. Defining the relationship between a document and its attachments is a mandatory requirement when you need to comply with the PDF/A-3 standard.

3.5.6 Viewer preferences

The `/PageLayout`, `/PageMode` and `/ViewerPreferences` entries refer to the way the document must be presented on the screen when the document is opened. The `/PageLayout` entry tells the PDF viewer how pages should be displayed. Possible values are:

- `/SinglePage`— Display one page at a time.
- `/OneColumn`— Display the pages in one column.
- `/TwoColumnLeft`— Display the pages in two columns, with the odd-numbered pages on the left.
- `/TwoColumnRight`— Display the pages in two columns, with the odd-numbered pages on the right.
- `/TwoPageLeft`— Display the pages two at a time, with the odd-numbered pages on the left.
- `/TwoPageRight`— Display the pages two at a time, with the odd-numbered pages on the right.

The `/PageMode` entry defines which panel —if any— needs to be opened next to the actual content. You can also use it to have the document opened in full screen view. Possible values are:

- `/UseNone`— Neither document outline nor thumbnail images visible.
- `/UseOutlines`— The bookmarks panel is visible, showing the outline tree.
- `/UseThumbs`— A panel with pages visualized as thumbnails is visible.
- `/FullScreen`— The document is shown in *full screen* mode.
- `/UseOC`— The panel with the optional content structure is open.
- `/UseAttachments`— The attachments panel is visible.

The values of both the page layout and the page mode are expressed as names. The `/ViewerPreferences` are stored in a dictionary. Table 3.15 lists the most important entries involving the viewer application.

Key	Value	Description
`/NonFullScreenPageMode`	Name	The document's page mode when exiting from full screen mode; this entry only makes sense if the value of the `/PageMode` entry is `/FullScreen`. `/UseNone`: no panel is opened `/UseOutlines`: the bookmarks panel is opened `/UseThumbs`: the thumbnails panels is opened `/UseOC`: the optional content panel is opened
`/FitWindow`	Boolean	Changes the zoom factor to fit the size of the first displayed page when `true`.
`/CenterWindow`	Boolean	Positions the document's window in the center of the screen when `true`.
`/DisplayDocTitle`	Boolean	Shows the title of the document as stored in the metadata in the title bar of the viewer.

Key	Value	Description
/HideToolbar	Boolean	Hides the toolbars in the PDF viewer when true.
/HideMenubar	Boolean	Hides the menubar in the PDF viewer when true.
/HideWindowUI	Boolean	Hides user interface elements in the document's window (scrollbars, navigation controls) when true.

Table 3.16 lists the most important entries with respect to printing the document.

Key	Value	Description
/PrintScaling	Name	Allows you to avoid print scaling by the viewer by setting the value to /None; the default is /AppDefault.
/Duplex	Name	The paper handling option. Possible values are: /Simplex: print single sided /DuplexFlipShortEdge: duplex and flip on the short edge of the sheet. /DuplexFlipLongEdge: duplex and flip on the long edge of the sheet.
/PickTrayByPDFSize	Boolean	If true, the check box in the print dialog associated with input paper tray will be checked.
/PrintPageRange	Array	The page numbers to initialize the print dialog box. The array consists of an even number of integers of which each pair defines a subrange of pages with the first and the last page to be printed.
/NumCopies	Integer	Presets the value of the number of copies that need to be printed.

These viewer preferences preselect or preset a value in the dialog box. They can be used to set parameters, not to actually print the document.

FAQ: How can I print a document silently?

In old versions of Adobe Reader, it was possible to print a PDF without any user interaction. This was known as silent printing. This *feature* can also be seen as a security hazard. A PDF with silent printing activated would start your printer the moment the user opens the document, without asking the user for permission. This *problem* was fixed in the more recent versions of Adobe Reader. Silent printing is no longer possible. The end user always has to confirm that the document can be printed in the print dialog.

Other possible entries in the /ViewerPreferences dictionary are:

- /Direction— to define the predominant order for text (/L2R for left to right and /R2L for right to left).
- /ViewArea, /ViewClip, /PrintArea *and* /PrintClip— to define page boundaries. These entries are deprecated and should no longer be used.
- /Enforce— a new entry introduced in PDF 2.0 with an array of viewer preferences that shall be enforced in the sense that they can't be overridden in the viewer's user interface.

Whether or not setting these viewer preferences has any effect depends on the implementation of the PDF viewer. Not all PDF viewers respect the viewer preferences as defined in the PDF document.

3.5.7 Metadata

In section 2.1.4, we've found a reference to the info dictionary in the trailer. This info dictionry contains metadata such as the title of the document, its author, some keywords, the creation and modification date, and so on. However: this type of storing metadata inside a PDF file will be deprecated in PDF 2.0. Let's find out which type of metadata will remain available in the near future.

3.5.7.1 Version

As explained in section 2.1.1, you can find the PDF version in the document header. However, there are two situations that required an alternative place to store the PDF version.

1. When creating a PDF on the fly, the first bytes can already be sent to the output stream before the document has been completed. Now suppose that your application starts by writing `%PDF-1.4`, but you decide to introduce functionality that didn't exist in PDF 1.4 —for instance Optional Content— during the writing process. You can't change the first bytes anymore. They are sent to an output stream that could be out of reach —for instance a browser on a client machine. In this case, you'll change the version at the level of the catalog. This explains why iText always writes the Document Catalog Dictionary as one of the last objects in the file structure, followed only by the `/info` dictinary. You need to be able to change the keys of the root dictionary up until the very last step in the process.
2. When creating an incremental update, you add an extra body, cross-reference table and trailer. Suppose that you want to add an extra signature to a signed PDF. Suppose that the type of signature you're adding didn't exist in the version of the original PDF. You can't change the existing header in an incremental update; if you tried, you'd break the original signature. In this case, you'll change the version by defining it in the Catalog.

The value of the `/Version` entry in the Catalog is a name object. For instance: `/1.4`, `/1.7`, `/2.0`,...

3.5.7.2 Extensions

Third party vendors can —within certain limits— extend the PDF specification with their own features. When they use these extensions in a document, they'll add an `/Extensions` dictionary that contains a prefix that serves as identification for the vendor or developer, as well as a version number for the extensions that are used in the docment.

3.5.7.3 XMP streams

The `/Info` will be deprecated in favor of using a `/Metadata` entry in the Catalog starting with PDF 2.0 (ISO-32000-2). The value of this entry is a stream of which the dictionary has two additional entries: the `/Type` entry of which the value shall be `/Metadata`, and the `/Subtype` of which the value shall be `/XML`. The metadata is stored as an XML stream. This stream is usually uncompressed so that it can be detected and parsed by applications who aren't PDF aware.

There can be more than one Metadata stream inside a document. One can add a `/Metadata` entry to a page dictionary, or any other object that requires metadata.

The XML grammar that is used for the XML is described in a separate standard (ISO-16684-1:2012) known as the Extensible Metadata Platform (XMP). This standard includes different schemas such as Dublin Core. The XMP specification is outside the scope of this book.

3.5.7.4 The natural language specification

In the context of accessibility, it is recommended that you add a `/Lang` entry to the Catalog. The value of this entry is a string that represents a language identifier that specifies the natural language for all the text in the document. The language defined in the Catalog is valid for the complete document, except where overridden by language specification for marked content or structure elements.

3.5.8 Extra information stored in the Catalog

The Catalog can also be used to store specific information about the content, the producer that created the PDF, and the reader application that will consume the PDF.

The following entries provide more information about the content in some very specific use cases:

- *Threads* — The content of a PDF can consist of different items that are logically connected, but not physically sequential. For instance: an article in a news paper can consist of different blocks of text, distributed over different pages. For instance: a title with some text on the front page, and the rest of the article somewhere in the middle of the news paper. The `/Threads` entry in the Catalog, allows you to store an array of thread dictionaries defining the separate articles.
- *Legal* — JavaScript, optional content,... PDF offers plenty of functionality that can make the rendered appearance of a document vary. This functionality could potentially be used to construct a document that misleads the recipient of the document. These situations are relevant when considering the legal implications of a *signed* PDF document. With the `/Legal` entry, we'll add a dictionary that lists how many JavaScript actions can be found in the document, how many annotations, etc. In case of a legal challenge of the document, any questionable content can be reviewed in the context of the information in this dictionary.

These entries are used by specific software products that produce PDF documents:

- *Private data from the processor* — software that produces PDF as one of its output formats can use the `/PieceInfo` entry to store private PDF processor data. This extra data will be ignored by a PDF viewer.
- *Web Capture data* — if the PDF was the result of a Web Capture operation, the `/SpiderInfo` entry can be used to store the commands that were used to create the document.

These entries are meant to be inspected by software that consumes PDF documents:

- *Permissions* — a PDF can be signed to grant the user specific permissions, for instance to save a filled out form locally. The `/Perms` entry will define the usage rights granted for this document.

- *Requirements* — not all PDF consumers support the complete PDF specification. The `/Requirements` entry allows you to define an array of the minimum functionality a processor must support (optional content, digital signatures, XFA,...) as well as a penalty if these requirements aren't met.
- *Reader requirements* — The `/ReaderRequirements` entry is similar to the `/Requirements` entry, but defines specific reader requirements, for instance related to output intents.
- *Output intents* — The `/OutputIntents` entry consists of an array of output intent dictionaries specifying color characteristics of output devices on which the document might be rendered.

This concludes the overview of possible entries in the root dictionary aka catalog of a PDF document.

3.6 Summary

We've covered a lot of ground in this chapter. After examining the file structure in chapter 2, we've now learned how to obtain an instance of the objects discussed in chapter 1.

We've started exploring the pages of a document starting from the root dictionary and we've gotten used to the concept of using dictionaries to store destinations, outline items, action, and many other elements. While doing so, we've discovered that there's more to a page than meets the eye. We'll elaborate on some concepts such as annotations and optional content in later chapters.

The same goes for the other entries in the document catalog. We've only scratched the surface of what is available in the PDF reference.

In part 2, we'll dive into the content of a page. We'll talk about graphics and text, as well as about structure.

II Part 2: The Adobe Imaging Model

We studied the Carousel Object System and the structure of PDF files and documents in the previous part. While doing so, we briefly looked at a specific type of stream, more specifically a stream containing PDF syntax that draws lines, shapes, text and images to a page. In this part, we'll take a closer look at this syntax.

We'll start by looking at the different operators and operands that can be used to draw lines and shapes and to change properties such as the color, line widths, and so on. We usually refer to the *graphics state* in this context.

In the next chapter, we'll discuss the *text state*, which is a subset of the graphics state. We'll discover how to show text on a page, referring to a font program that knows how to draw each glyph.

Finally, we'll revisit some of the entries in the Catalog dictionary that we discussed only briefly, involving marked content, tagged PDF and optional content.

4. Graphics State

In section 3.4.1.1, we've already seen a glimpse of a content stream when we looked at the content stream of a page. Let's take a look at a similar content snippet:

```
BT
36 788 Td
/F1 12 Tf
(Hello World )Tj
ET
q
0 0 m
595 842 l
S
Q
```

This code snippet writes the words *"Hello Word"* to a pages and strokes a diagonal line.

4.1 Understanding the syntax

Before we start with a syntax overview, let's start by looking at the syntax notation, and find out how the imaging model was implemented in iText.

4.1.1 PDF Syntax Notation

The Portable Document Format evolved from the PostScript language and uses the same syntax notation known as postfix, aka reverse Polish notation. In reverse Polish notation, the operators follow their operands. Table 4.1 shows the different notations that can be used to note down the addition of the integers 10 and 6.

Table 4.1: Mathematical notations

Notation	Example	Description
prefix	+ 10 6	Polish notation
infix	10 + 6	The common arithmetic and logical formula notation
postfix	10 6 +	Reverse Polish notation

Interpreters of the postfix notation are often stack-based. Operands are pushed onto a stack, and when an operation is performed, its operands are popped from a stack and its result is pushed back on. This has the advantage of being easy to implement and very fast.

Let's take a look at the snippet `595 842 l` taken from the content stream in our example. We see the *path construction operator* `l` preceded by its operands, `595` and `842`, which are in this case values for an (x , y) coordinate. You'll find a corresponding method for this operator in iText. There's a `lineTo()` method in the `PdfContentByte` class that is responsible for writing two parameters, `x` and `y`, to a byte buffer, followed by the operator `l`. You can use this method if you want to create PDF at the lowest-level, using PDF syntax instead of the high-level objects described in the book "Create your PDFs with iText[1]."

4.1.2 Creating a PDF using low-level PDF syntax

Creating a PDF using iText always requires five basic steps:

1. Create a `Document` object
2. Get a `Pdfwriter`instance
3. Open the `Document`
4. Add content
5. Close the `Document`

Code sample 4.1 shows the five steps. In the fourth step, we use the `PdfContentByte` object to add some text and some graphics. The corresponding PDF syntax is added as a comment after each line.

Code sample 4.1: C0401_ImagingModel

```
// step 1
Document document = new Document();
// step 2
PdfWriter writer = PdfWriter.getInstance(document, new FileOutputStream(dest));
// step 3
document.open();
// step 4
PdfContentByte canvas = writer.getDirectContent();
canvas.beginText();                              // BT
canvas.moveText(36, 788);                        // 36 788 Td
canvas.setFontAndSize(BaseFont.createFont(), 12); // /F1 12 Tf
canvas.showText("Hello World ");                 // (Hello World )Tj
canvas.endText();                                // ET
canvas.saveState();      // q
canvas.moveTo(0, 0);     // 0 0 m
canvas.lineTo(595, 842); // 595 842 l
canvas.stroke();         // S
canvas.restoreState();   // Q
// step 5
document.close();
```

[1] https://leanpub.com/itext_pdfcreate

Figure 4.1 shows the result of this code sample. We see the text *Hello World* positioned more or less at the top of the page (36, 788). We also see a diagonal line going from the lower-left corner (0, 0) to the upper-right corner (595, 842).

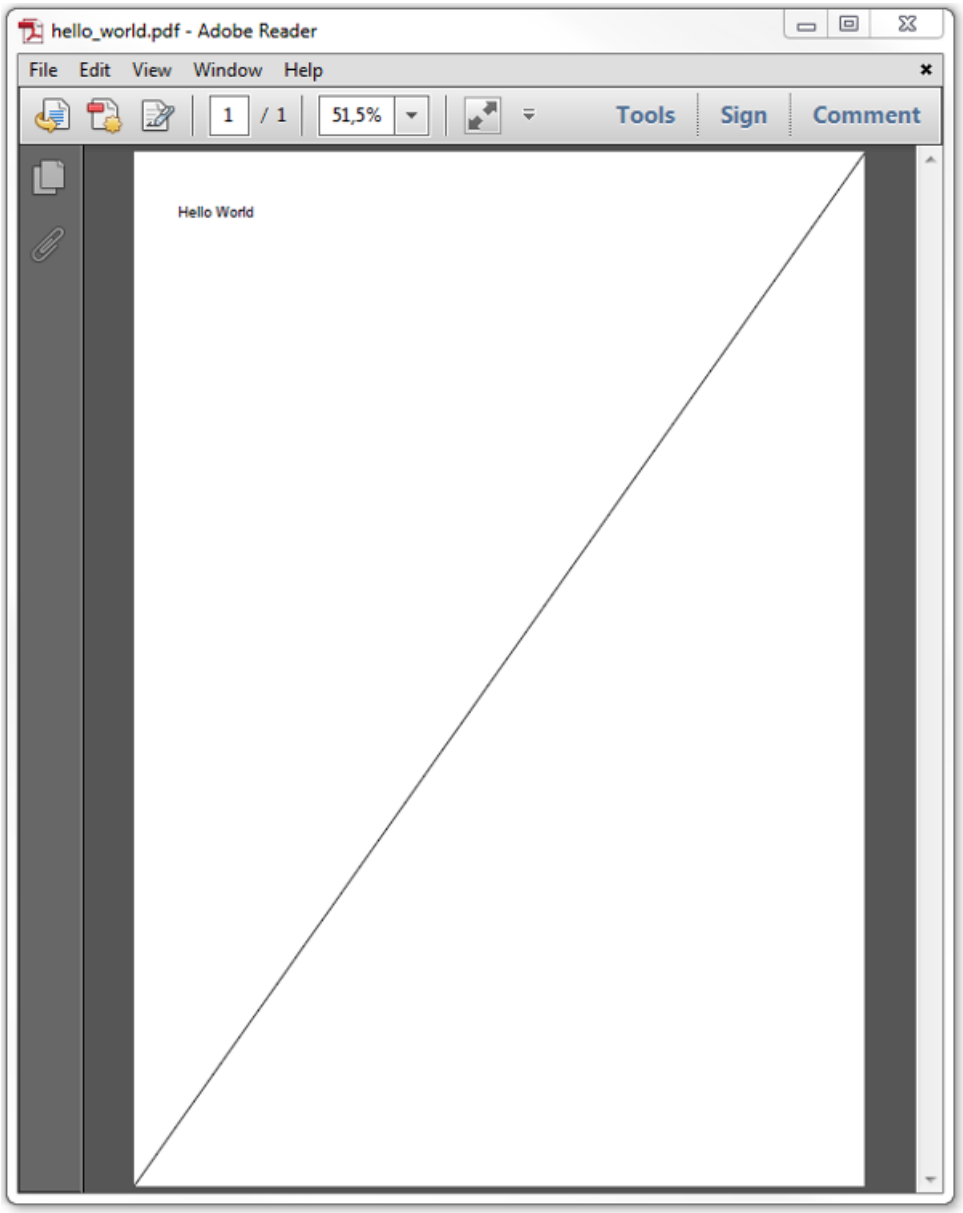


Figure 4.1: Hello World example

Now lets take a look at figure 4.2 where RUPS shows what's under the hood of the PDF.

RUPS inflates the compressed stream to allow us to see the PDF syntax. It removes or introduces spaces and newlines: all operands are separated by a single space character, each operator is shown on a separate line. It highlights the syntax in different colors: text state operators are shown in blue; pure graphics state operators are shown in orange; operands are shown in black. The *begin text* and *end text* operators, as well as the *save state* and *end state* operators are also highlighted differently.

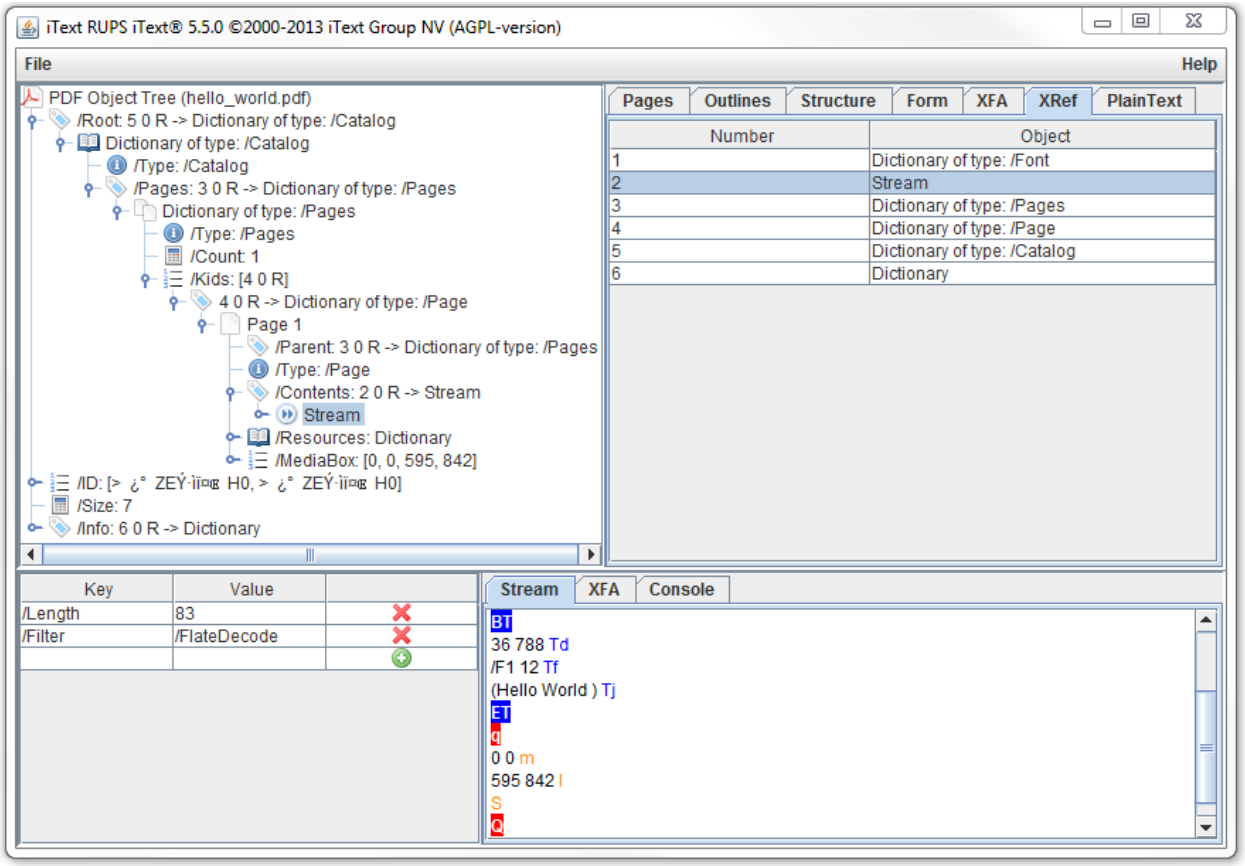


Figure 4.2: Syntax of the Hello World example

As you can see, RUPS makes it really easy for PDF-savvy people to read the syntax of different graphics objects.

4.1.3 Graphics objects

There are 5 types of graphics objects in PDF:

1. A *path object* is a shape created using path construction and painting operators. The path construction operators allow you to define lines, rectangles and curves. With the path painting operators you can fill or stroke the paths. You can also use a path to clip content.
2. A *text object* consists of a sequence of operators enclosed between the begin text and end text operator. A text object always refers to a font program that knows how to draw glyphs. Just like paths, these glyphs can be filled, stroked or used to clip content.
3. An *external object* (XObject) is an object defined outside the content stream and referenced by its name. The most common types of XObjects are *form XObjects* refering to another content stream that is to be considered as a single graphics object, and *image XObjects* refering to a raster image such as a JPEG, a CCITT image, and so on.
4. An *inline image object* uses special syntax to include raster image data within the content stream. This is only allowed for images with a size up to 4096 bytes.

5. A *shading object* describes a geometric shape whose color at a specific position is defined by a function, for instance a function defining a gradient transitioning from one color to another. A shading can also be used as a color when painting other graphics objects. It's not considered to be a separate graphics object in that case.

These objects are created using graphics state operators.

4.2 Graphics State Operators

There are different categories of graphics state operators: general graphics state operators, special graphics state operators, color operators, shading operators, path construction operators, path painting operators, clipping path operators, XObjects operators, inline images operators, text objects operators, text state operators, text positioning operators, text showing operators, Type 3 fonts operators, marked content operators and compatibility operators.

All of these operators are supported in iText, except for the compatibility operators. The `BX` and `EX` operators are used to begin and end a sequence of operators that may not be recognized by a PDF processor. Such a PDF processor will ignore unrecognized operators without reporting an error.

Lets start with the operators that allow us to draw a path object.

4.2.1 Constructing path objects

A path object always starts with one of the following operators: `m` or `re`. It ends either with a path painting or a path clipping operator. All the available path construction operators are shown in figure 4.3.

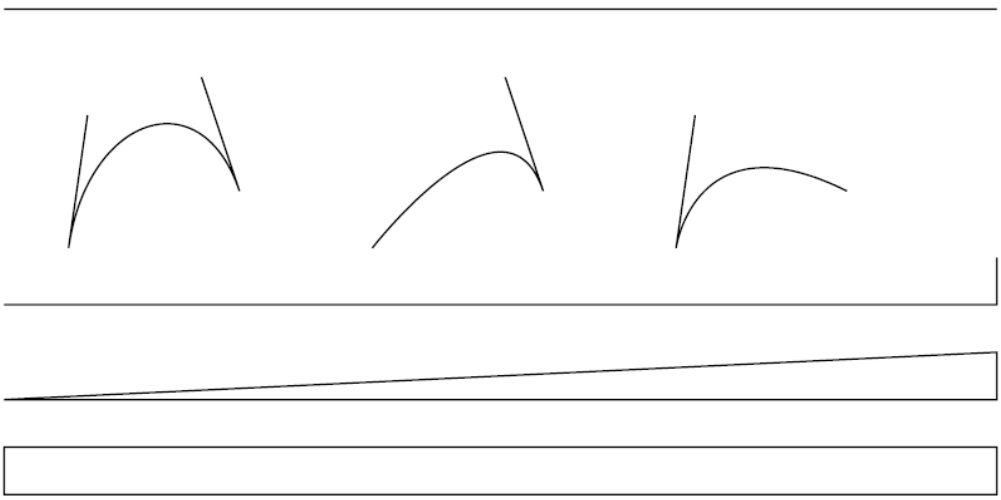


Figure 4.3: Path construction operators

Let's take a look at the overview of all the available path construction operators and find out if we can recognize them in figure 4.3.

In the first column, we have the PDF operator; in the second column you'll find the corresponding iText method; the parameters for those methods (the operands needed by the operator) are listed in the third column; the fourth column gives us a description.

Table 4.2: PDF path construction operators and operands

PDF	iText	Parameters	Description
`m`	`moveTo`	`(x, y)`	Moves the current point to coordinates `(x, y)`, omitting any connecting line segment. This begins a new (sub)path.
`l`	`lineTo`	`(x, y)`	Moves the current point to coordinates `(x, y)`, appending a line segment from the previous to the new current point.
`c`	`curveTo`	`(x1, y1, x2, y2, x3, y3)`	Moves the current point to coordinates `(x3, y3)`, appending a cubic Bézier curve from the previous to the new current point, using `(x1, y1)` and `(x2, y2)` as Bézier control points
`v`	`curveTo`	`(x2, y2, x3, y3)`	Moves the current point to coordinates `(x3, y3)`, appending a cubic Bézier curve from the previous to the new current point, using the previous current point and `(x2, y2)` as Bézier control points
`y`	`curveTo`	`(x1, y1, x3, y3)`	Moves the current point to coordinates `(x3, y3)`, appending a cubic Bézier curve from the previous to the new current point, using `(x1, y1)` and `(x3, y3)` as Bézier control points
`h`	`closePath`	`()`	Closes the current subpath by appending a straight line segment from the current point to the starting point of the subpath.
`re`	`rectangle`	`(x, y, w, h)`	Starts a new path with a rectangle or appends this rectangle to the current path as a complete subpath. The `x` and `y` parameter define the coordinate of the lower-left corner; `w` and `h` define the width and the height of the rectangle.

Code sample 4.2 shows the code that was used to create the PDF in figure 4.3.

Code sample 4.2: C0302_PathConstruction

```
PdfContentByte canvas = writer.getDirectContent();
// a line
canvas.moveTo(36, 806);
canvas.lineTo(559, 806);
// lines and curves
canvas.moveTo(70, 680);
canvas.lineTo(80, 750);
canvas.moveTo(140, 770);
canvas.lineTo(160, 710);
canvas.moveTo(70, 680);
canvas.curveTo(80, 750, 140, 770, 160, 710);
canvas.moveTo(300, 770);
canvas.lineTo(320, 710);
canvas.moveTo(230, 680);
canvas.curveTo(300, 770, 320, 710);
canvas.moveTo(390, 680);
canvas.lineTo(400, 750);
canvas.moveTo(390, 680);
canvas.curveTo(400, 750, 480, 710);
// two sides of a triangle
canvas.moveTo(36, 650);
canvas.lineTo(559, 650);
canvas.lineTo(559, 675);
// three sides of a triangle
canvas.moveTo(36, 600);
canvas.lineTo(559, 600);
canvas.lineTo(559, 625);
canvas.closePath();
// a rectangle
canvas.rectangle(36, 550, 523, 25);
// nothing is drawn unless we stroke:
canvas.stroke();
```

We start with a `moveTo()` and a `lineTo()` operation. This draws the first line.

Then we draw some more lines followed by the three flavors of the `curveTo()` method. These `curveTo()` methods create *Bézier curves*.

Bézier curves are parametric curves developed in 1959 by Paul de Casteljau (using *de Casteljau's algorithm*). They were widely publicized in 1962 by Paul Bézier, who used them to design automobile bodies. Nowadays they're important in computer graphics.

Cubic Bézier curves are defined by four points: the two *endpoints* —the current point and point `(x3, y3)`— and two *control points* —`(x1, y1)` and `(x2, y2)`. The curve starts at the first endpoint going onward to the

first control point. In general, the curve doesn't pass through the control points. They're only there to provide directional information. The distance between an endpoint ad its corresponding control point determines how long the curve moves toward the control point before turning toward the other endpoint. In figure 4.3, we've added lines that connect the endpoints with their corresponding control point. In the second curve, the endpoint to the left coincides with the first control point (the `v` operator was used instead of `c`). In the third curve, the endpoint to the right coincides with the second control point (the `y` operator was used).

Right under the curves, we see a subpath consisting of two lines. It is followed by another subpath that was constructed by a single `moveTo()` and two `lineTo()` operators, but instead of two lines, we now see a triangle. That's because we've used the `closePath()` operator. This operator adds a linear segment to the subpath that connects the current endpoint with the original startpoint of the subpath that was started with a `moveTo()` operation. Finally, we've also used the `rectangle()` method to draw a rectangle.

FAQ: I've constructed a path, but I can't see any line or shape in my document

The operators we've discussed so far can be used to *construct* a path. This doesn't mean the path is actually drawn. To draw the path, you need a path painting operator. In the example, we used the `stroke()` method to stroke the paths.

The PDF specification doesn't have any operator that allows you to draw a circle or an ellipse. Instead, you're supposed to combine the path construction operators listed in table 4.2. For instance: a circle consists of one `moveTo()` and four `curveTo()` operations. This isn't trivial. Fortunately, iText provides a handful of convenience methods, as listed in table 4.3.

Table 4.3: Convenience methods for specific shapes

iText method	Parameters	Description
`ellipse()`	`(x1, y1, x2, y2)`	Constructs the path of an ellipse inscribed within the rectangle `[x1 y1 x2 y2]`.
`arc()`	`(x1, y1, x2, y2, a, e)`	Constructs a path of a partial ellipse inscribed within the rectangle `[x1 y1 x2 y2]`; starting at `a` degrees (the start angle) and covering `e` degrees (the extent). Angles start with `0` to the right and increase counterclockwise.
`circle()`	`(x, y, r)`	Constructs the path of a circle with center `(x, y)` and radius `r`.
`roundRectangle()`	`(x, y, w, h, r)`	Constructs the path of a rounded rectangle: `(x, y)` is the coordinate of the lower-left corner; `w` and `h` define the width and the height. The radius used for the rounded corners is `r`.

Now that we know how to construct paths, let's find out how to paint them.

4.2.2 Painting and Clipping Path Objects

We've already used one painting operator in the previous examples: the `stroke()` operator `S`. To explain all the possible painting operators, we'll work with a series of paths that represent a set of triangles as shown in figure 4.4.

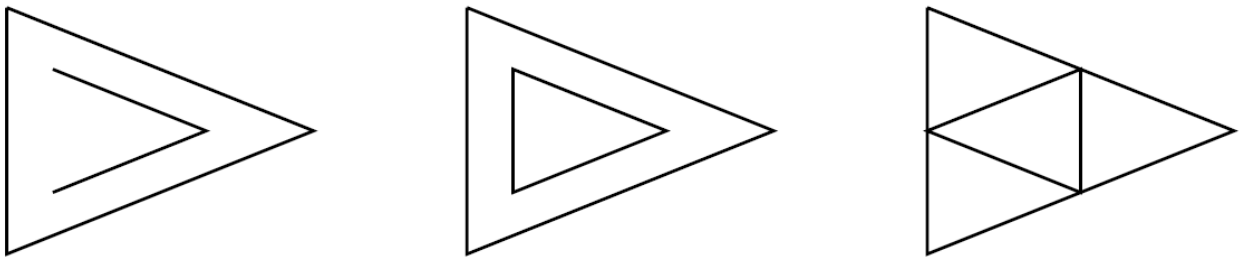


Figure 4.4: stroke() and closePathStroke()

Code sample 4.3 shows how we constructed the paths rendered in figure 4.4.

Code sample 4.3: C0403_PathPainting

```
protected void triangles1(PdfContentByte canvas) {
    canvas.moveTo(50, 760);
    canvas.lineTo(150, 720);
    canvas.lineTo(50, 680);
    canvas.lineTo(50, 760);
    canvas.moveTo(65, 740);
    canvas.lineTo(115, 720);
    canvas.lineTo(65, 700);
}
protected void triangles2(PdfContentByte canvas) {
    canvas.moveTo(200, 760);
    canvas.lineTo(300, 720);
    canvas.lineTo(200, 680);
    canvas.lineTo(200, 760);
    canvas.moveTo(215, 740);
    canvas.lineTo(265, 720);
    canvas.lineTo(215, 700);
}
protected void triangles3(PdfContentByte canvas) {
    canvas.moveTo(350, 760);
    canvas.lineTo(450, 720);
    canvas.lineTo(350, 680);
    canvas.lineTo(350, 760);
    canvas.moveTo(400, 740);
    canvas.lineTo(350, 720);
    canvas.lineTo(400, 700);
}
```

In the `triangles1()` and `triangles2()` method, we draw one large triangle using three `lineTo()` methods, one for each side of the triangle. We start with the upper-left corner, draw a line that goes down to the right, followed by a line that returns down to the left. We close the path by connecting the lower-left corner with the upper-left corner. Inside this large triangle, we draw two sides of a smaller rectangle. Again we start with the upper-left corner, we draw a line to the right, followed by a line that returns to the left.

The `triangles3()` method is slightly different. The outer triangle is drawn in exactly the same way as before, but when we draw the inner triangle, we start to the right, we add a line that moves down to the right, followed by a line that moves down to the left. In `triangles1()` and `triangles2()` the two triangles are drawn using the clockwise orientation. In `triangles3()` one triangle is drawn clockwise, the other one counterclockwise.

The orientation of the paths doesn't matter when we merely stroke the paths. Code sample 4.4 shows how we've painted the paths shown in figure 4.4.

Code sample 4.4: C0403_PathPainting

```
1 PdfContentByte canvas = writer.getDirectContent();
2 triangles1(canvas);
3 canvas.stroke();
4 triangles2(canvas);
5 canvas.closePathStroke();
6 triangles3(canvas);
7 canvas.closePathStroke();
```

This code snippet explains why the second and third triangle have three sides in figure 4.4 in spite of the fact that we only constructed two lines. The `stroke()` method will only stroke two lines; the `closePathStroke()` method will close the path first, then stroke it.

Figure 4.5 shows what happens if we fill the path instead of stroking it.

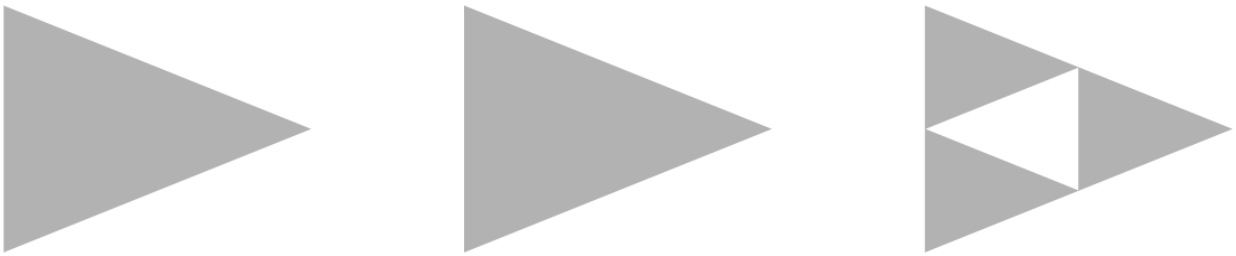


Figure 4.5: fill()

In the first two sets of triangles, both the other and the inner triangle are filled. This isn't the case in the third set: the inner triangle made a hole in the outer triangle. Let's take a look at code sample 4.5 and then discover why that hole is there.

Code sample 4.5: C0403_PathPainting

```
triangles1(canvas);
canvas.fill();
triangles2(canvas);
canvas.fill();
triangles3(canvas);
canvas.fill();
```

We have filled the different sets of triangles using the `fill()` method. This method uses the *nonzero winding number rule* to determine whether or not a given point is inside a path.

With the nonzero winding number rule, you need to draw a line from that point in any direction, and examine every intersection of the path with this line. Start with a count of zero; add one each time a subpath crosses the line from left to right; subtract one each time a subpath crosses from right to left. Continue doing this until there are no more path segments to cross. If the final result is zero, the point is outside the path; otherwise it's inside.

This explains why the orientation we used to draw the segments of the triangles matters. The winding number count for the points inside the inner triangle is 2: when drawing a line from inside the inner triangle to outside the outer triangle, we encounter two segments drawn from left to right. In the third set, the count is 0 because we encounter a segment drawn from right to left (subtract one), followed by a segment drawn from left to right (add one).

Figure 4.6 shows two more methods that use the nonzero winding number rule:

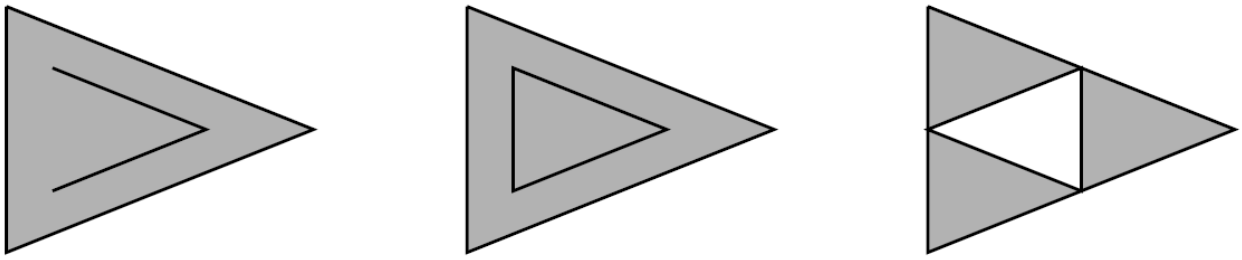


Figure 4.6: fillStroke() and closePathFillStroke()

Code sample 4.6 shows how these triangles were drawn.

Code sample 4.6: C0403_PathPainting

```
triangles1(canvas);
canvas.fillStroke();
triangles2(canvas);
canvas.closePathFillStroke();
triangles3(canvas);
canvas.closePathFillStroke();;
```

The `fillStroke()` method is a combination of the `fill()` and the `stroke()` method. The `closePathFillStroke()` method combines `closePath()`, `fill()` and `stroke()`.

Figure 4.7 shows a different way to fill the paths. In this case there's also a hole in the first two sets of triangles.

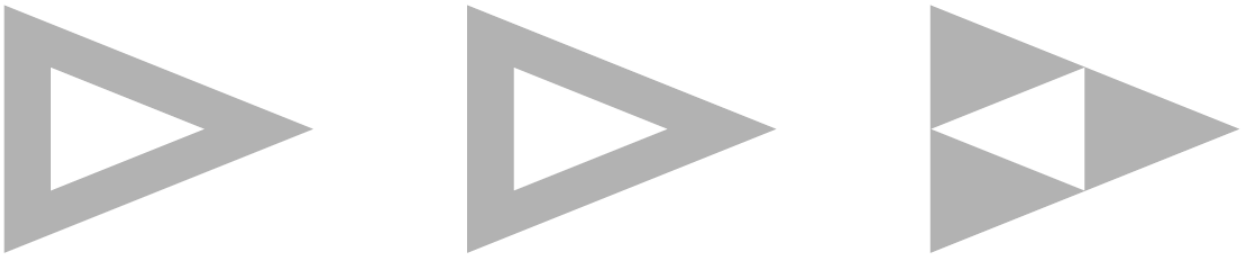


Figure 4.7: eoFill()

Please compare code sample 4.7 with code sample 4.5.

Code sample 4.7: C0403_PathPainting

```
triangles1(canvas);
canvas.eoFill();
triangles2(canvas);
canvas.eoFill();
triangles3(canvas);
canvas.eoFill();
```

The `eoFill()` method uses the *even-odd rule* to determine whether or not a given point is inside a path.

With the even-odd rule, you draw a line from the point that's being examined to infinity. Now count the number of path segments that are crossed, regardless of their orientation. If this number is odd, the point is inside; if even, the point is outside.

In this case, the orientation we used to draw the triangles doesn't matter.

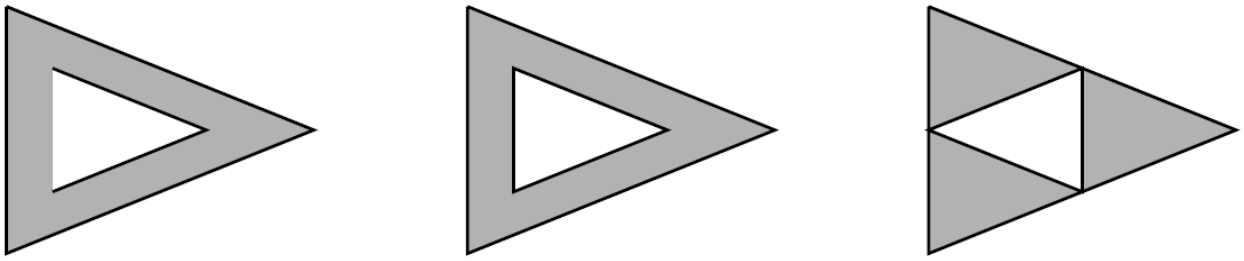


Figure 4.8: eoFillStroke() and closePathEoFillStroke()

Figure 4.8 shows two more methods using the even-odd rule. These methods are used in code sample 4.8.

Code sample 4.8: C0403_PathPainting

```
triangles1(canvas);
canvas.eoFillStroke();
triangles2(canvas);
canvas.closePathEoFillStroke();
triangles3(canvas);
canvas.closePathEoFillStroke();
```

The nonzero winding number rule and the even-odd rule can also be used for clipping. Figure 4.9 looks identical to figure 4.5.

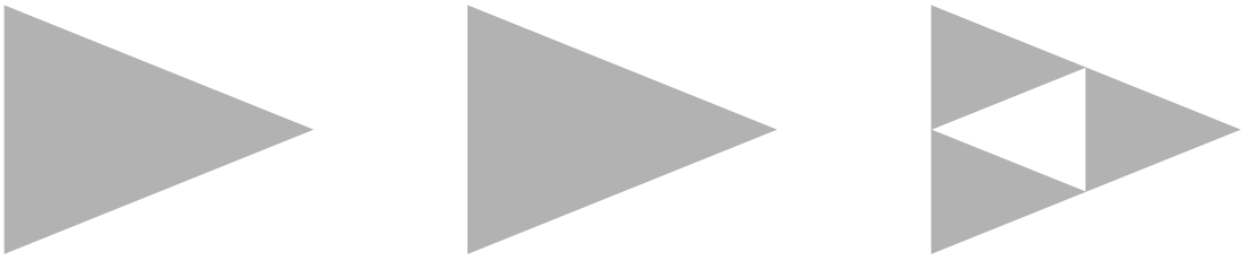


Figure 4.9: clip()

In spite of the resemblance, the code is completely different. See code sample 4.9.

Code sample 4.9: C0403_PathPainting

```
canvas.saveState();
triangles1(canvas);
canvas.clip();
canvas.newPath();
canvas.rectangle(45, 675, 120, 100);
canvas.fill();
canvas.restoreState();
canvas.saveState();
triangles2(canvas);
canvas.clip();
```

```
canvas.newPath();
canvas.rectangle(195, 675, 120, 100);
canvas.fill();
canvas.restoreState();
canvas.saveState();
triangles3(canvas);
canvas.clip();
canvas.newPath();
canvas.rectangle(345, 675, 120, 100);
canvas.fill();
canvas.restoreState();
```

In this snippet, we draw the same paths, but we use them as clipping paths by invoking the `clip()` method. It's not our intention to draw the paths we've constructed, hence we start a new path with the `newPath()` method. Then we draw a rectangle and we fill that rectangle. The result a rectangle that is clipped using the path of the triangles. As we used the `clip()` method, the nonzero winding number rule is used.

Figure 4.10 shows what happens when we use the even-odd rule for the clipping path.

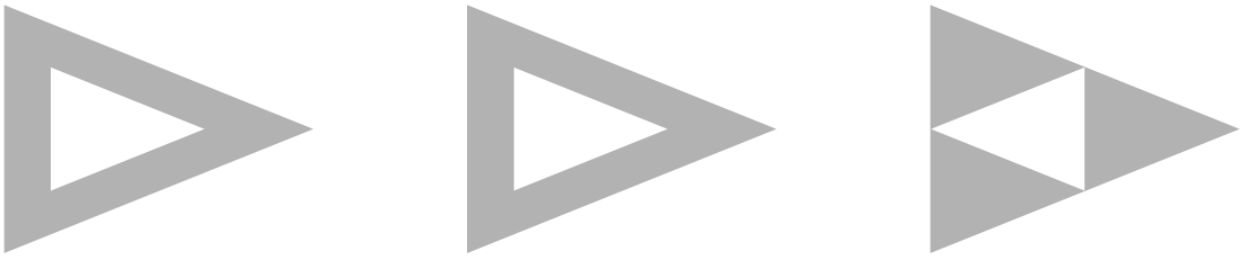


Figure 4.10: eoClip()

Code sample 4.10 shows how it's done.

Code sample 4.10: C0403_PathPainting

```
canvas.saveState();
triangles1(canvas);
canvas.eoClip();
canvas.newPath();
canvas.rectangle(45, 675, 120, 100);
canvas.fill();
canvas.restoreState();
canvas.saveState();
triangles2(canvas);
canvas.eoClip();
canvas.newPath();
canvas.rectangle(195, 675, 120, 100);
canvas.fill();
canvas.restoreState();
```

```
canvas.saveState();
triangles3(canvas);
canvas.eoClip();
canvas.newPath();
canvas.rectangle(345, 675, 120, 100);
canvas.fill();
canvas.restoreState();
```

In these code snippets, we introduced some methods we haven't discussed yet. We've also silently changed the fill color: the default color is black, not gray. Before we continue with more types of graphics states operators, let's look at an overview all the path painting operators in table 4.4.

Table 4.4: Path painting and clipping path operators

PDF	iText	Description
S	`stroke()`	Strokes the path: lines only; the shape isn't filled.
s	`closePathStroke()`	Closes and strokes the path. This is the same as doing `closePath()` and `stroke()`.
f	`fill()`	Fills the path using the nonzero winding number rule. Open subpaths are closed implicitly.
F	—	Deprecated! Equivalent to `f`, and included for compatibility. ISO-32000-1 says that PDF writer applications should use `f` instead.
f*	`eoFill()`	Fills the path using the even-odd rule.
B	`fillStroke()`	Fills the path using the nonzero winding number rule, and then strokes the path. This is equivalent to `fill()` followed by `stroke()`.
B*	`eoFillStroke()`	Fills the path using the even-odd rule, and then strokes the path. This is equivalent to `eoFill()` followed by `stroke()`.
b	`closePathFillStroke()`	Closes, fills, and strokes the path, as is done with `closePath()` followed by `fillStroke()`.
b*	`closePathEoFillStroke()`	Closes, fills, and strokes the path, as is done with `closePath()` followed by `eoFillStroke()`.
n	`newPath()`	Ends the path object without filling or stroking it. Used primarily after defining a clipping path.
W	`clip()`	Modifies the current clipping path by intersecting it with the current path, using the nonzero winding number rule.
W*	`eoClip()`	Modifies the current clipping path by intersecting it with the current path, using the even-odd rule.

In figures 4.4 to 4.10, we used the default stroke color, but we changed the fill color to paint the path. Let's take a closer look at the way colors are defined in PDF.

4.2.3 Color, color spaces and shading

When painting a path with a stroke or fill method, the *current color*. This color is defined using a value that consists of one or more color components, for instance: a set of three numbers for the values of the red, green and blue components. These values are interpreted according to the current color space. The PDF specification distinguishes 11 different color spaces, grouped into three categories:

1. *Device color spaces*— these specify colors or shades of gray that the output device must produce: grayscale, RGB (red-green-blue) or CMYK (cyan-magenta-yellow-black), corresponding to the color space families **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**.
2. *CIE-based color spaces*— these are based on the international standard for color specification created by the *Commission Internationale de l'éclairage* (CIE). The colors are specified in a device-independent way: **CalGray**, **CalRGB**, **Lab** and **ICCBased**.
3. *Special color spaces*— these add features to an underlying color space. They include facilities for patterns, color mapping, separations, and high-fidelity and multitone clor: **Pattern**, **Indexed**, **Separation**, and **DeviceN**.

Not all of these color spaces are currently supported when creating a document from scratch using iText. It wouldn't be difficult to extend iText to support them, but so far, we haven't received any request to do so. We'll focus on the functionality that is available and discuss device colors, spot colors and painting patterns. While we're at it, we'll also discuss shadings.

4.2.3.1 Device colors

Device colors allow you to control color precisely for a particular device. The family consists of three color spaces:

1. *DeviceRGB*— This is an additive color model: red, green, and blue light is used to produce the other colors. For example, if you add red light (`#FF0000`) to green light (`#00FF00`), you get yellow light (`#FFFF00`). This is how a TV-screen works: the colors are composed of red, green and blue dots. RGB is typically used for graphics that need to be rendered on a screen.
2. *DeviceCMYK*— This is a subtractive color model: If you look at an object using white light, you see a color because the object reflects and absorbs some of the wavelengths that make up white light. A yellow object absorbs blue and reflects red and green. White (`#FFFFFF`) minus blue (`#0000FF`) equals yellow (`#FFFF00`). The subtractive color model is used when printing a document. You don't use red, green, and blue, but cyan (C), magenta (M), yellow (Y), and black (K).
3. *DeviceGray*— This is the default color space when drawing lines or shapes in PDF is gray. It is expressed as the intensity of achromatic light, represented by a single number in the range 0 to 1, where 0 corresponds to black, 1 to white, and intermediate values to different gray levels.

The iText classes corresponding with these color spaces are `BaseColor`, `CMYKColor`, and `GrayColor`. The color values can either be defined using `int` values ranging from 0 to 255, or as `float` ranging values from 0.0 to 1.0. Figure 4.11 shows the result of using these classes.

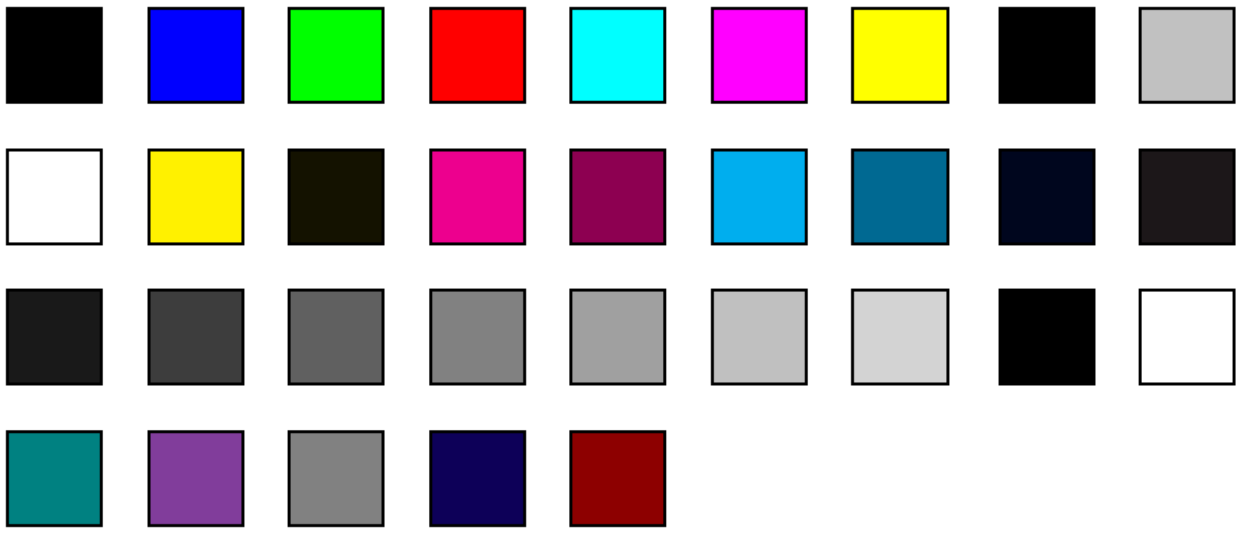


Figure 4.11: Device colors

Code sample 4.11 shows a method that will be used in the next couple of examples involving color. It changes the color using the `setColorFill()` method and then draws a rectangle that will be filled in that color.

Code sample 4.11: drawing a colored rectangle

```
public void colorRectangle(PdfContentByte canvas,
        BaseColor color, float x, float y, float width, float height) {
    canvas.saveState();
    canvas.setColorFill(color);
    canvas.rectangle(x, y, width, height);
    canvas.fillStroke();
    canvas.restoreState();
}
```

Let's start by drawing some squares using the `BaseColor` class.

4.2.3.1.1 RGB

Code sample 4.12 draws the first row of squares in figure 4.11.

Code sample 4.12: C0404_DeviceColor

```
colorRectangle(canvas, new BaseColor(0x00, 0x00, 0x00), 36, 770, 36, 36);
colorRectangle(canvas, new BaseColor(0x00, 0x00, 0xFF), 90, 770, 36, 36);
colorRectangle(canvas, new BaseColor(0x00, 0xFF, 0x00), 144, 770, 36, 36);
colorRectangle(canvas, new BaseColor(255, 0, 0), 198, 770, 36, 36);
colorRectangle(canvas, new BaseColor(0f, 1f, 1f), 252, 770, 36, 36);
colorRectangle(canvas, new BaseColor(1f, 0f, 1f), 306, 770, 36, 36);
colorRectangle(canvas, new BaseColor(1f, 1f, 0f), 360, 770, 36, 36);
colorRectangle(canvas, BaseColor.BLACK, 416, 770, 36, 36);
colorRectangle(canvas, BaseColor.LIGHT_GRAY, 470, 770, 36, 36);
```

In the first four lines, we create a `BaseColor` instance using four `int` values.

I have the habit of writing the integers in hexadecimal form. Although both notations are semantically identical, I find `(0x00, 0x00, 0xFF)` easier to read than `(0, 0, 255)` because it refers to the way colors are usually defined in HTML: `#0000FF`. That's only a matter of taste. In the fourth line, I use the decimal notation equivalent to `new BaseColor(0xFF, 0x00, 0x00)`.

In lines 5 to 7, we use `float` values. We recognize the values for cyan, magenta and yellow, because we only used ones and zeros. Should we have used values between 0 and 1, we'd have a harder time to recognize which colors are selected.

Finally, we have two lines where we use colors that are predefined as constants in the `BaseColor` class. Possible values are `WHITE`, `LIGHT_GRAY`, `GRAY`, `DARK_GRAY`, `BLACK`, `RED`, `PINK`, `ORANGE`, `YELLOW`, `GREEN`, `MAGENTA`, `CYAN`, and `BLUE`.

4.2.3.1.2 CMYK

The second row of squares in figure 4.11 are CMYK colors. Code sample 4.13 shows how these colors were created.

Code sample 4.13: C0404_DeviceColor

```
1 colorRectangle(canvas, new CMYKColor(0x00, 0x00, 0x00, 0x00), 36, 716, 36, 36);
2 colorRectangle(canvas, new CMYKColor(0x00, 0x00, 0xFF, 0x00), 90, 716, 36, 36);
3 colorRectangle(canvas, new CMYKColor(0x00, 0x00, 0xFF, 0xFF), 144, 716, 36, 36);
4 colorRectangle(canvas, new CMYKColor(0x00, 0xFF, 0x00, 0x00), 198, 716, 36, 36);
5 colorRectangle(canvas, new CMYKColor(0f, 1f, 0f, 0.5f), 252, 716, 36, 36);
6 colorRectangle(canvas, new CMYKColor(1f, 0f, 0f, 0f), 306, 716, 36, 36);
7 colorRectangle(canvas, new CMYKColor(1f, 0f, 0f, 0.5f), 360, 716, 36, 36);
8 colorRectangle(canvas, new CMYKColor(1f, 0f, 0f, 1f), 416, 716, 36, 36);
9 colorRectangle(canvas, new CMYKColor(0f, 0f, 0f, 1f), 470, 716, 36, 36);
```

We now use the `CMYKColor` class. It's a subclass of the abstract `ExtendedColor` class which is in turn a subclass of the `BaseColor` class. All other available color classes in iText are subclasses of `ExtendedColor`.

There are currently no constants available for specific CMYK colors.

4.2.3.1.3 Gray

The third row of squares in figure 4.11 are nine shades of gray. Code sample 4.14 shows how these colors were created.

Code sample 4.14: C0404_DeviceColor

```
colorRectangle(canvas, new GrayColor(0x20), 36, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0x40), 90, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0x60), 144, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0.5f), 198, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0.625f), 252, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0.75f), 306, 662, 36, 36);
colorRectangle(canvas, new GrayColor(0.825f), 360, 662, 36, 36);
colorRectangle(canvas, GrayColor.GRAYBLACK, 416, 662, 36, 36);
colorRectangle(canvas, GrayColor.GRAYWHITE, 470, 662, 36, 36);
```

In the first three lines, we use an `int` values, in the next four lines a `float`, and in the final two lines, we use the two constants that are available in the `GrayColor` class: `GRAYBLACK` and `GRAYWHITE`.

4.2.3.1.4 Other methods to define the fill color

There's also a fourth lines of squares in figure 4.11. We used variations of the `setColorFill()` method that don't require a `BaseColor` instance. See code sample 4.15.

Code sample 4.14: C0404_DeviceColor

```
canvas.setRGBColorFill(0x00, 0x80, 0x80);
canvas.rectangle(36, 608, 36, 36);
canvas.fillStroke();
canvas.setRGBColorFillF(0.5f, 0.25f, 0.60f);
canvas.rectangle(90, 608, 36, 36);
canvas.fillStroke();
canvas.setGrayFill(0.5f);
canvas.rectangle(144, 608, 36, 36);
canvas.fillStroke();
canvas.setCMYKColorFill(0xFF, 0xFF, 0x00, 0x80);
canvas.rectangle(198, 608, 36, 36);
canvas.fillStroke();
canvas.setCMYKColorFillF(0f, 1f, 1f, 0.5f);
canvas.rectangle(252, 608, 36, 36);
canvas.fillStroke();
```

We'll list all the available methods in a table at the end of section 4.2.3.

We've covered the three types of device colors, let's now talk about ink, more specifically about spot colors.

4.2.3.2 Spot colors

A spot color is any color generated by an ink (pure or mixed) that is printed in a single run. Spot colors are using in the context of Separation color spaces. Section 8.6.6.4 of ISO-32000-1, titled "Separation Color Spaces," contain the following note:

> When printing a page, most devices produce a single composite page on which all process colorants (and spot colorants, if any) are combined. However, some devices, such as image setters, produce a separate, monochromatic rendition of the page, called a separation, for each colorant. When the separations are later combined—on a printing press, for example—and the proper inks or other colorants are applied to them, the result is a full color page.
>
> The term separation is often misused as a synonym for an individual device colorant. In the context of this discussion, a printing system that produces separations generates a separate piece of physical medium (generally a film) for each colorant. It is these pieces of physical medium that are correctly referred to as separations. A particular colorant properly constitutes a separation only if the device is generating physical separations, one of which corresponds to the given colorant. The Separation color space is so named for historical reasons, but it has evolved to a broader purpose of controlling the application of individual colorants in general, regardless of whether they are actually realized as physical separations.
>
> **—ISO-32000-1, section 8.6.6.4**

Every color in the Separation color space has a name. Every color value consists of a single tint component in the range of 0.0 to 1.0. A tint value of 0.0 denotes the lightest color that can be achieved with a given colorant, and 1.0 is the darkest. Figure 4.12 shows a number of colorants that were created using the `PdfSpotColor` and `SpotColor` classes.

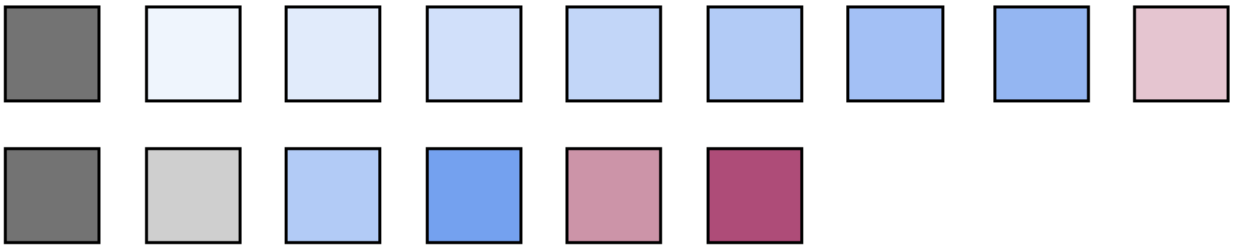


Figure 4.12: Separation colors

Listing 4.15 shows how we define the colorants. Observe that I gave each colorant a name: `iTextSpotColorGray`, `iTextSpotColorRGB` and `iTextSpotColorCMYK`. The colorant itself is defined using the color classes we've discovered in the previous section about device colors.

Code sample 4.15: C0405_SeparationColor

```
PdfSpotColor psc_g = new PdfSpotColor(
    "iTextSpotColorGray", new GrayColor(0.9f));
PdfSpotColor psc_rgb = new PdfSpotColor(
    "iTextSpotColorRGB", new BaseColor(0x64, 0x95, 0xed));
PdfSpotColor psc_cmyk = new PdfSpotColor(
    "iTextSpotColorCMYK", new CMYKColor(0.3f, .9f, .3f, .1f));
```

As shown in figure 4.13, we'll find references to these colors in the `/ColorSpace` entry of the page where the color is used.

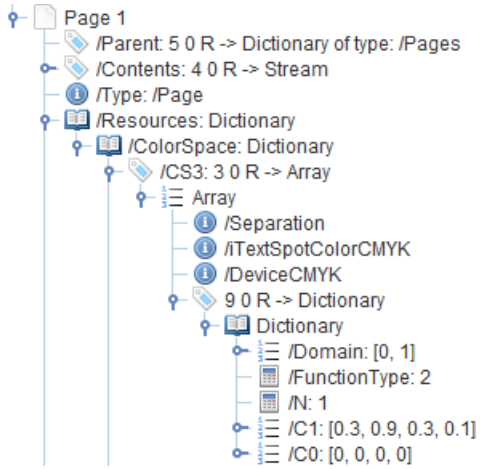


Figure 4.13: Separation colors

We use these colorants in code sample 4.16.

Code sample 4.16: C0405_SeparationColor

```
colorRectangle(canvas, new SpotColor(psc_g, 0.5f), 36, 770, 36, 36);
colorRectangle(canvas, new SpotColor(psc_rgb, 0.1f), 90, 770, 36, 36);
colorRectangle(canvas, new SpotColor(psc_rgb, 0.2f), 144, 770, 36, 36);
...
colorRectangle(canvas, new SpotColor(psc_rgb, 0.6f), 360, 770, 36, 36);
colorRectangle(canvas, new SpotColor(psc_rgb, 0.7f), 416, 770, 36, 36);
colorRectangle(canvas, new SpotColor(psc_cmyk, 0.25f), 470, 770, 36, 36);
```

We create a `SpotColor` instance using the `PdfSpotColor` colorant and a value between 0.0 and 1.0 for the tint. In the first rows of figure 4.12, we start with a gray square. We then have sever squares with different tints of blue. The row is closed with a light pink square.

In code sample 4.17, we use the colorants without creating an instance of the `SpotColor` class. We define the color for the squares in the second row using a special `setColorFill()` method.

Code sample 4.17: C0405_SeparationColor

```
canvas.setColorFill(psc_g, 0.5f);
canvas.rectangle(36, 716, 36, 36);
canvas.fillStroke();
canvas.setColorFill(psc_g, 0.9f);
canvas.rectangle(90, 716, 36, 36);
canvas.fillStroke();
...
canvas.setColorFill(psc_cmyk, 0.9f);
canvas.rectangle(306, 716, 36, 36);
canvas.fillStroke();
```

So far, we've always used a single color when stroking or filling a path, but it's also possible to apply "paint" that consists of repeating graphical figures or a smoothly varying color gradient. In this case, we're talking about pattern colors that use either a *tiling pattern* (a repeating figure) or a *shading pattern* (a smooth gradient).

4.2.3.3 Tiling Pattern

Figure 4.14 shows a couple of examples of tiled patterns.

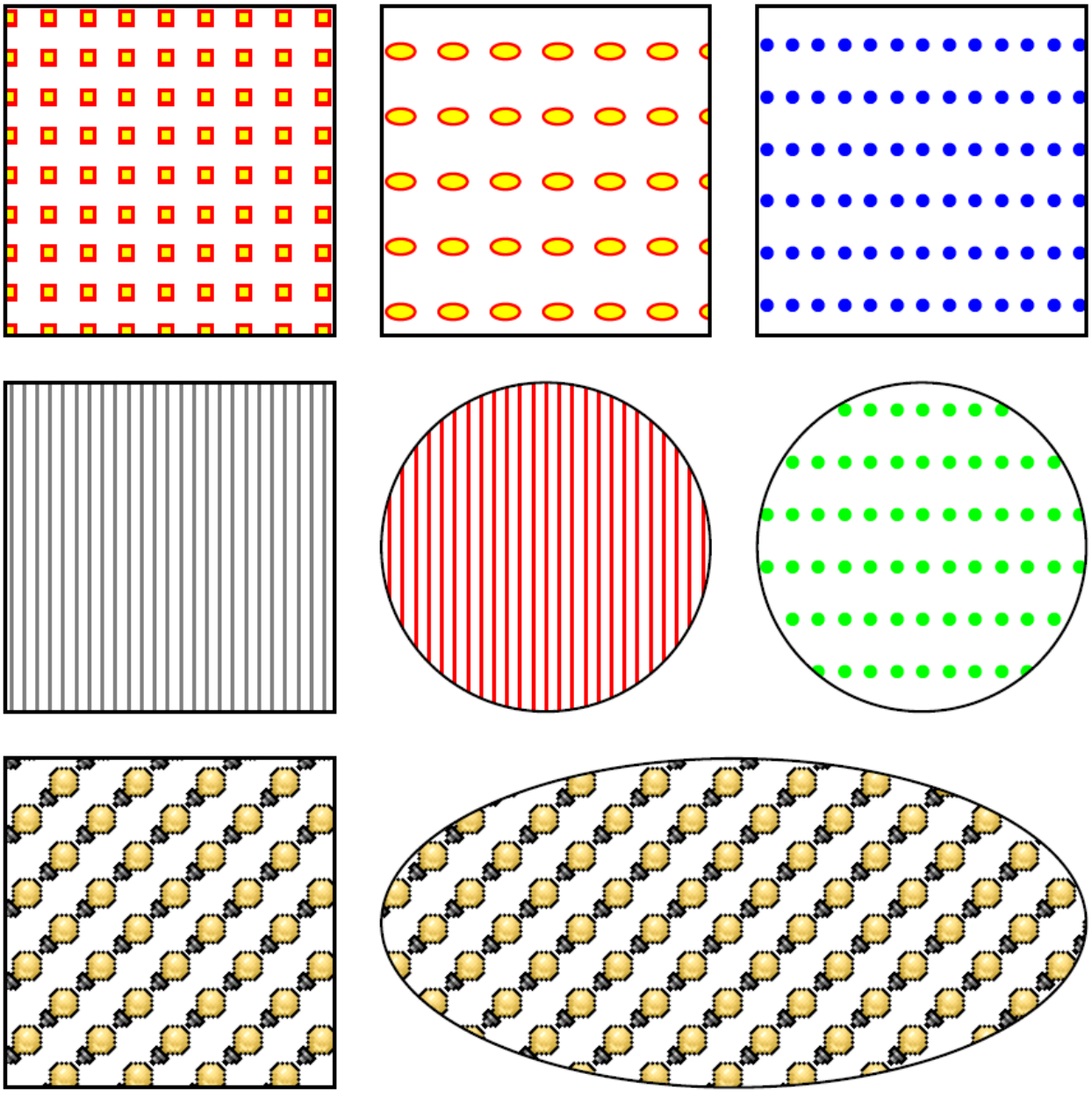


Figure 4.14: Pattern colors

To created a tiled pattern color, we must construct a *pattern cell*, or, in the context of iText, an instance of the `PdfPatternPainter` class. We can create such a pattern cell from the `PdfContentByte` object with the `createPattern()` method. This pattern object inherits all the graphics state methods discussed in this chapter from the `PdfContentByte` class. There are two kinds of tiling patterns: *colored tiling patterns* and *uncolored tiling patterns*.

4.2.3.3.1 Colored tiling pattern

A colored tiling pattern is self-contained. In the course of painting the pattern cell, the pattern's content stream explicitly sets the color of each graphical element it paints. Code sample 4.18 shows how we create two pattern cells, used for the yellow squares with the red borders and the ditto ellipses, used to paint the top left squares in figure 4.13.

Code sample 4.18: C0406_PatternColor

```
PdfContentByte canvas = writer.getDirectContent();
PdfPatternPainter square = canvas.createPattern(15, 15);
square.setColorFill(new BaseColor(0xFF, 0xFF, 0x00));
square.setColorStroke(new BaseColor(0xFF, 0x00, 0x00));
square.rectangle(5, 5, 5, 5);
square.fillStroke();
PdfPatternPainter ellipse = canvas.createPattern(15, 10, 20, 25);
ellipse.setColorFill(new BaseColor(0xFF, 0xFF, 0x00));
ellipse.setColorStroke(new BaseColor(0xFF, 0x00, 0x00));
ellipse.ellipse(2f, 2f, 13f, 8f);
ellipse.fillStroke();
```

In this code sample, we see two variations of the `createPattern()` method. The simplest version accepts two `float` values: one for the width and one for the height of the pattern cell. Additionally, you can also specify an X and Y step. This is the desired horizontal and vertical spacing between pattern cells. There are similar variations to create cells for uncolored tiling patterns.

4.2.3.3.2 Uncolored tiling pattern

A colored tiling pattern can consist of different colors, but uncolored tiling patterns are monochrome. You can create such a pattern by adding a default color (or `null`) as a parameter for the `createPattern()` method. This is shown in code sample 4.19.

Code sample 4.19: C0406_PatternColor

```
PdfPatternPainter circle = canvas.createPattern(15, 15, 10, 20, BaseColor.BLUE);
circle.circle(7.5f, 7.5f, 2.5f);
circle.fill();
PdfPatternPainter line = canvas.createPattern(5, 10, null);
line.setLineWidth(1);
line.moveTo(3, -1);
line.lineTo(3, 11);
line.stroke();
```

In this code sample, we create a pattern with a circle that is blue by default (line 1-3), and we create a pattern with an uncolored line (line 4-8). We'll use these patterns in a moment.

4.2.3.3.3 Changing the pattern matrix

In code sample 20, we create a colored tiling pattern using an image, and we also change the pattern matrix.

Code sample 4.20: C0406_PatternColor

```
1 Image img = Image.getInstance(IMG);
2 img.scaleAbsolute(20, 20);
3 img.setAbsolutePosition(0, 0);
4 PdfPatternPainter img_pattern = canvas.createPattern(20, 20, 20, 20);
5 img_pattern.addImage(img);
6 double d45 = -Math.PI / 4;
7 img_pattern.setPatternMatrix(
8     (float)Math.cos(d45), (float)Math.sin(d45),
9     -(float)Math.sin(d45), (float)Math.cos(d45), 0f, 0f);
```

We'll discuss the transformation matrix later on in this chapter. In this case, we change the matrix in a way that the pattern cell with the image is rotated by 45 degrees.

4.2.3.3.4 Using the pattern cell

Once we have a `PdfPatternPainter` object, we can use it in two ways. We can either create a `PatternColor` object, or we can use one of the `setPatternFill()` methods. This is shown in code sample 21.

Code sample 4.21: C0406_PatternColor

```
1  colorRectangle(canvas, new PatternColor(square), 36, 696, 126, 126);
2  colorRectangle(canvas, new PatternColor(ellipse), 180, 696, 126, 126);
3  colorRectangle(canvas, new PatternColor(circle), 324, 696, 126, 126);
4  colorRectangle(canvas, new PatternColor(line), 36, 552, 126, 126);
5  colorRectangle(canvas, new PatternColor(img_pattern), 36, 408, 126, 126);
6  canvas.setPatternFill(line, BaseColor.RED);
7  canvas.ellipse(180, 552, 306, 678);
8  canvas.fillStroke();
9  canvas.setPatternFill(circle, BaseColor.GREEN);
10 canvas.ellipse(324, 552, 450, 678);
11 canvas.fillStroke();
12 canvas.setPatternFill(img_pattern);
13 canvas.ellipse(180, 408, 450, 534);
14 canvas.fillStroke()
```

The first two `PatternColor` instances are created using colored tiling patterns, so is the pattern color with the image. The third and the fourth `PatternColor` instances are created using an uncolored tiling pattern.

For the pattern with the circles, we defined blue as the default color. We didn't define a color for the line. Looking at figure 4.13, we see that Adobe Reader renders gray lines. Other PDF viewers may not show any pattern at all. It's safer to use the `setPatternFill()` method for uncolored tiling patterns and to pass the color that needs to be used to paint the pattern as an extra parameter. This method can also be used for colored tiling patterns, in which case no color must be defined.

A better name for uncolored tiling patterns might have been monochrome tiling patterns, but the next type of pattern will be much more colorful.

4.2.3.4 Shading pattern

When we defined graphics objects in section 4.1.3, we talked about a *shading object* that describes a geometric shape whose color at a specific position is defined by a function. In figure 4.15, we see two such objects.

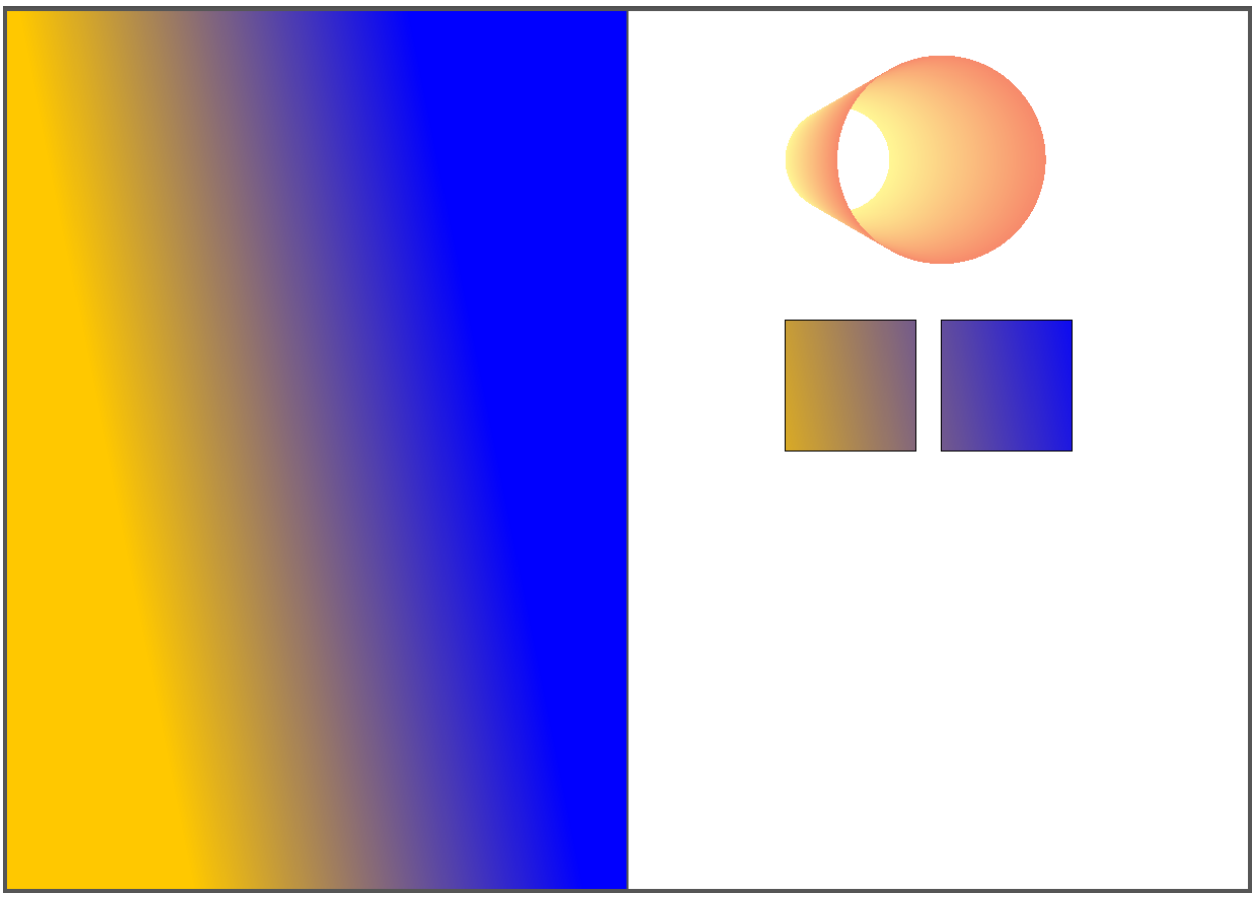


Figure 4.15: Shading objects and colors

We see two different types of shading objects:

1. *Axial shadings (Type 2 function in ISO-32000-1)*—These define a color blend that varies along a linear axis between two endpoints, in our case `(36, 716)` (orange) and `(396, 788)` (blue), and extends indefinitely perpendicular to that axis.

2. *Radial shadings (Type 3 function in ISO-32000-1)*—These define a color blend that varies between two circles, in our case a circle with center `(200, 700)` and radius `50` and a circle with center `(300, 700)` and radius `100`.

Code sample 4.22 shows how these graphics objects were created. We use `PdfShading`'s static `simpleAxial()` or `simpleRadial()` method, and we paint the object using the `paintShading()` method.

Code sample 4.22: C0407_Shading

```
PdfContentByte canvas = writer.getDirectContent();
PdfShading axial = PdfShading.simpleAxial(writer,
    36, 716, 396, 788, BaseColor.ORANGE, BaseColor.BLUE);
canvas.paintShading(axial);
document.newPage();
PdfShading radial = PdfShading.simpleRadial(writer,
    200, 700, 50, 300, 700, 100,
    new BaseColor(0xFF, 0xF7, 0x94),
    new BaseColor(0xF7, 0x8A, 0x6B), false, false);
canvas.paintShading(radial);
```

The boolean values passed to the `simpleRadial()` method indicate whether or not the color should be extended beyond the circle.

We can also use a shading as a color. For instance by wrapping it inside a `ShadingColor` object, or by using the `setShadingFill()` method. This is done in code sample 4.23.

Code sample 4.23: C0407_Shading

```
PdfShadingPattern shading = new PdfShadingPattern(axial);
colorRectangle(canvas, new ShadingColor(shading), 150, 420, 126, 126);
canvas.setShadingFill(shading);
canvas.rectangle(300, 420, 126, 126);
canvas.fillStroke();
```

Shadings are created using specific types of functions. So far we've seen an example involving type 2 (axial) and type 3 (radial) functions. The PDF specification includes five more types. If you want to use these other types, you need to combine one or more of the static `type()` methods of the `PdfShading` class. This goes beyond the scope of the ABC of PDF. Please consult ISO-32000-1 for more info, and inspect the implementation of the `simpleAxial()` and `simpleRadial()` methods in the iText source code for inspiration.

As promised, we conclude the section about color, color spaces and shadings with an overview of the operators and methods we've discussed.

4.2.3.5 Overview of the color operators

You can change the color of the current graphics state using the methods `setColorStroke()` and `setColorFill()`. These methods accept an instance of the `BaseColor` class. This class has many different subclasses, and the type of the subclass will determine which operator is used.

Table 4.5 lists the different operators and operands that are at play.

Table 4.5: Color and shading operators

PDF	iText	Parameters	Description
g	setGrayFill	(gray)	Sets the color space to DeviceGray and changes the current gray tint for filling paths to a float value from 0 (black) to 1 (white).
G	setGrayStroke	(gray)	Sets the color space to DeviceGray and changes the current gray tint for stroking paths to a float value from 0 (black) to 1 (white).
rg	setRGBColorFill	(r, g, b)	Sets the color space to DeviceRGB and changes the current color for filling paths. The color values are integers from 0 to 255.
RG	setRGBColorStroke	(r, g, b)	Sets the color space to DeviceRGB and changes the current color for stroking paths. The color values are integers from 0 to 255.
rg	setRGBColorFillF	(r, g, b)	Sets the color space to DeviceRGB and changes the current color for filling paths. The color values are floats from 0 to 1.
RG	setRGBColorStrokeF	(r, g, b)	Sets the color space to DeviceRGB and changes the current color for stroking paths. The color values are floats from 0 to 1.
k	setCMYKColorFill	(c, m, y, k)	Sets the color space to DeviceCMYK and changes the current color for filling paths. The color values are integers from 0 to 255.
K	setCMYKColorStroke	(c, m, y, k)	Sets the color space to DeviceCMYK and changes the current color for stroking paths. The color values are integers from 0 to 255.
k	setCMYKColorFillF	(c, m, y, k)	Sets the color space to DeviceCMYK and changes the current color for filling paths. The color values are floats from 0 to 1.
K	setCMYKColorStrokeF	(c, m, y, k)	Sets the color space to DeviceCMYK and changes the current color for stroking paths. The color values are floats from 0 to 1.
cs		name	Sets the color space for nonstroking operations. this is done implicitly by iText when necessary.

Table 4.5: Color and shading operators

PDF	iText	Parameters	Description
`CS`		`name`	Sets the color space for stroking operations. this is done implicitly by iText when necessary.
`sc`		`c1 c2 c3 ...`	Sets the color to use for nonstroking operations in a device, CIE-based (other than ICCBased), or Indexed color space. Not used in iText.
`SC`		`c1 c2 c3 ...`	Same as `sc` for stroking operations.
`scn`		`c1 c2 c3 ... name`	Same as `sc` but also supports Pattern, Separation, DeviceN, and ICCBased colore spaces. The operand is implicitly used in the methods `setColorFill(PdfSpotColor sp, float tint)`, `setPatternFill(PdfPatternPainter p)`, `setPatternFill(PdfPatternPainter p, BaseColor color)`, `setPatternFill(PdfPatternPainter p, BaseColor color, float tint)`, and `setShadingFill(PdfShadingPattern shading)`.
`SCN`		`c1 c2 c3 ... name`	Same as `SC` but also supports Pattern, Separation, DeviceN, and ICCBased colore spaces. The operand is implicitly used in the methods `setColorStroke(PdfSpotColor sp, float tint)`, `setPatternStroke(PdfPatternPainter p)`, `setPatternStroke(PdfPatternPainter p, BaseColor color)`, `setPatternStroke(PdfPatternPainter p, BaseColor color, float tint)`, and `setShadingStroke(PdfShadingPattern shading)`.
`sh`	`paintShading`	`(shading)`	Paints the shape and color shading described by a shading dictionary. In iText the shading dictionary can be a `PdfShading` or `PdfShadingPattern` object.

This concludes the overview of color in the context of iText that are important when executing a `fill()` operation. Now let's take a look at some operators that can change the way paths are drawn when performing a `stroke()` operation.

4.2.4 General graphics state operators

The general graphics state operators allow you to change the line style when performing a `stroke()` operation. Figure 4.16 shows five different operators in action.

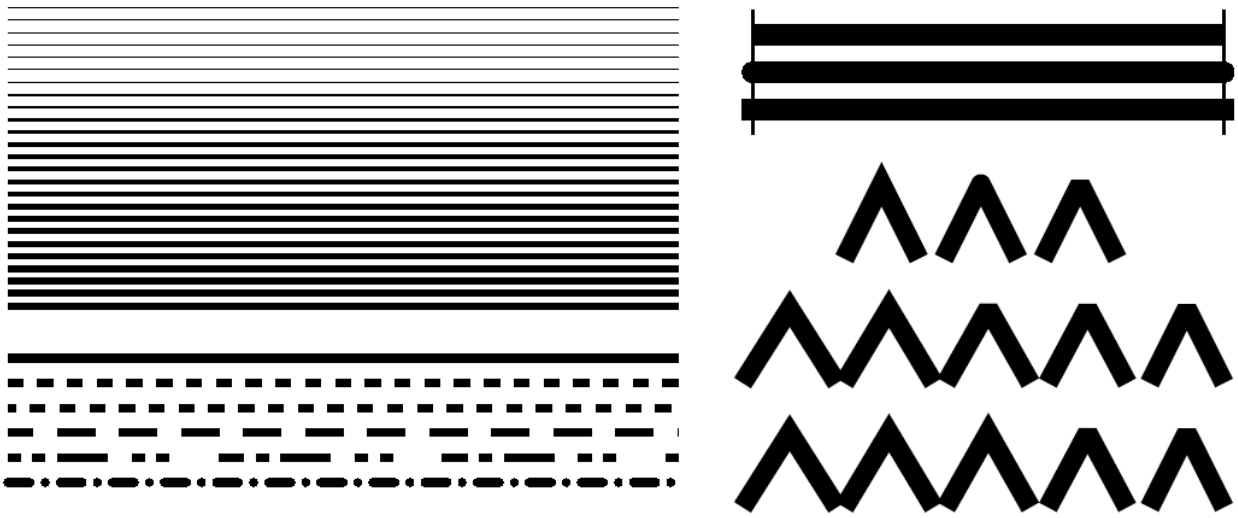


Figure 4.16: General graphics state operators

4.2.4.1 Line width

We can change the line width using the `setLineWidth()` method as shown in example 4.24.

Code sample 4.24: C0408_GeneralGraphicsOperators

```
for (int i = 25; i > 0; i--) {
    canvas.setLineWidth((float) i / 10);
    canvas.moveTo(50, 806 - (5 * i));
    canvas.lineTo(320, 806 - (5 * i));
    canvas.stroke();
}
```

The corresponding PDF syntax looks like this:

```
2.5 w 50 681 m 320 681 l S
2.4 w 50 686 m 320 686 l S ...
```

We recognize the `m`, `l` and `S` operator, we're now introducting to the `w` operator to change the width of the line.

4.2.4.2 Line cap

When we draw a thick line from one coordinate to another, we can choose between different line cap styles. This is shown in figure 4.17.

Figure 4.17: Line cap types

Code sample 4.25 shows how the line cap style can be changed using iText.

Code sample 4.25: C0408_GeneralGraphicsOperators

```
canvas.setLineCap(PdfContentByte.LINE_CAP_BUTT);
canvas.moveTo(350, 790);
canvas.lineTo(540, 790);
canvas.stroke();
canvas.setLineCap(PdfContentByte.LINE_CAP_ROUND);
canvas.moveTo(350, 775);
canvas.lineTo(540, 775);
canvas.stroke();
canvas.setLineCap(PdfContentByte.LINE_CAP_PROJECTING_SQUARE);
canvas.moveTo(350, 760);
canvas.lineTo(540, 760);
canvas.stroke();
```

When we translate the Java code to PDF syntax, we get:

```
0 J 350 790 m 540 790 l S
1 J 350 775 m 540 775 l S
2 J 350 760 m 540 760 l S
```

The line cap can be changed using the `J` operator, and there are three possible values as listed in table 4.6.

Table 4.6: Line Cap styles

PDF	iText	Description
`0`	`LINE_CAP_BUTT`	The stroke is squared off at the endpoint of the path. This is the default.
`1`	`LINE_CAP_ROUND`	A semicircular arc with diameter equal to the line width is drawn around the endpoint.

Table 4.6: Line Cap styles

PDF	iText	Description
2	LINE_CAP_PROJECTING_SQUARE	The stroke continues beyond the endpoint of the path for a distance equal to half of the line width.

These are the styles for the endpoints of a path. We can also define the way lines are joined.

4.2.4.3 Line join styles

Figure 4.18 shows the three different line join styles.

Figure 4.18: Line join types

This figure was created using the iText code shown in code sample

Code sample 4.26: C0408_GeneralGraphicsOperators

```
canvas.setLineJoin(PdfContentByte.LINE_JOIN_MITER);
canvas.moveTo(387, 700);
canvas.lineTo(402, 730);
canvas.lineTo(417, 700);
canvas.stroke();
canvas.setLineJoin(PdfContentByte.LINE_JOIN_ROUND);
canvas.moveTo(427, 700);
canvas.lineTo(442, 730);
canvas.lineTo(457, 700);
canvas.stroke();
canvas.setLineJoin(PdfContentByte.LINE_JOIN_BEVEL);
canvas.moveTo(467, 700);
canvas.lineTo(482, 730);
canvas.lineTo(497, 700);
canvas.stroke();
```

This translates to:

```
0 j 387 700 m 402 730 l 417 700 l S
1 j 427 700 m 442 730 l 457 700 l S
2 j 467 700 m 482 730 l 497 700 l S
```

Table 4.7 shows the possible values for the `j` operator in PDF or the `setLineJoin()`.

Table 4.7: Line Join Styles

PDF	iText	Description
`0`	`LINE_JOIN_MITER`	The outer edges of the strokes for two segments are extended until they meet at an angle. This is the default.
`1`	`LINE_JOIN_ROUND`	An arc of a circle with diameter equal to the line width is drawn around the point where the two line segments meet.
`2`	`LINE_JOIN_BEVEL`	The two line segments are finished with butt caps.

When you define miter joins, and two line segments meet at a sharp angle, it's possible for the miter to extend far beyond the thickness of the line stroke.

4.2.4.4 Miter limit

If φ is the angle between both line segments, the miter limit equals the line width divided by `sin(φ/2)`. You can define a maximum value for the ratio of the miter length to the line width. The maximum is called the *miter limit*. When this limit is exceeded, the join is converted from a miter to a bevel. Figure 4.19 shows two rows of hooks. In every row, the angle of the hook decreases from left to right.

Figure 4.19: Miter limits

In spit of the fact that the PDF syntax to draw the hooks is identical for both rows, the appearance of the third hook is different when comparing both rows. This is due to the fact that we defined a different miter limit as shown in code sample 4.27:

Code sample 4.27: C0408_GeneralGraphicsOperators

```
1 canvas.setMiterLimit(2);
2 // draw first row of hooks
3 canvas.setMiterLimit(2.1f);
4 // draw second row of hooks
```

In PDF syntax, you'll find the `M` operator, preceded by the value for the miter limit.

4.2.4.5 Dash patterns

There's one aspect of figure 4.16 we haven't discussed yet. See figure 4.20.

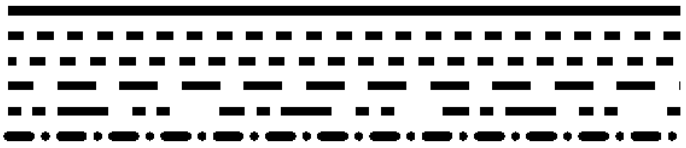


Figure 4.20: Dash patterns

First let's take a look at the code that was used to draw these lines.

Code sample 4.28: C0408_GeneralGraphicsOperators

```
canvas.setLineWidth(3);
canvas.moveTo(50, 660);
canvas.lineTo(320, 660);
canvas.stroke();
canvas.setLineDash(6, 0);
canvas.moveTo(50, 650);
canvas.lineTo(320, 650);
canvas.stroke();
canvas.setLineDash(6, 3);
canvas.moveTo(50, 640);
canvas.lineTo(320, 640);
canvas.stroke();
canvas.setLineDash(15, 10, 5);
canvas.moveTo(50, 630);
canvas.lineTo(320, 630);
canvas.stroke();
float[] dash1 = { 10, 5, 5, 5, 20 };
canvas.setLineDash(dash1, 5);
canvas.moveTo(50, 620);
canvas.lineTo(320, 620);
canvas.stroke();
float[] dash2 = { 9, 6, 0, 6 };
canvas.setLineCap(PdfContentByte.LINE_CAP_ROUND);
canvas.setLineDash(dash2, 0);
canvas.moveTo(50, 610);
canvas.lineTo(320, 610);
canvas.stroke();
```

This results in the following PDF syntax:

```
3 w 50 660 m 320 660 l S
[6] 0 d 50 650 m 320 650 l S
[6] 3 d 50 640 m 320 640 l S
[15, 10] 5 d 50 630 m 320 630 l S
[10, 5, 5, 5, 20] 5 d 50 620 m 320 620 l S
1 J [9, 6, 0, 6] 0 d 50 610 m 320 610 l
```

Six lines are drawn:

1. The first line is drawn using the default line style, which is solid.
2. For the second line, the line dash is set to a dash pattern of 6 units with phase 0. This means that the line starts with a dash of 6 units long, then there's a gap of 6 units, then there's a dash of 6 units, and so on.
3. The same goes for the third line, but it uses a different phase.
4. In line four, you have a dash of 15 units and a gap of 10 units. The phase is 5, so the first dash is 10 units long (15 - 5).
5. Line five uses a more complex pattern. You start with a dash of 5 (10 - 5), then there's a gap of 5, followed by a dash of 5, a gap of 5, and a dash of 20, and so on.
6. Line six is also special: a dash of 9, a gap of 6, a dash of 0, a gap of 6. The dash of 0 may seem odd, but as you're using round caps (`1 J`), a dot is drawn instead of a 0-length dash.

Let's take a look at an overview of all the available general graphics state operators.

4.2.4.6 Overview of the general graphics state operators

Table 4.8 lists the operators as defined in the PDF specification and in the iText API.

Table 4.8: General graphics state operators

PDF	iText	Parameters	Description
`w`	`setLineWidth`	`(width)`	Sets the line width. The parameter represents the thickness of the line in user units (default = 1).
`J`	`setLineCap`	`(style)`	Defines the line cap style.
`j`	`setLineJoin`	`(style)`	Defines the line join style.
`M`	`setMiterLimit`	`(miterLimit)`	Defines a limit for joining lines. When it's exceeded, the join is converted from a miter to a bevel.
`d`	`setLineDash`	`(phase)`, `(unitsOn, phase)`, `(unitsOn, unitsOff, phase)`, `(array, phase)`	Sets the line dash type. The default line dash is a solid line. You can create all sorts of dashed lines by using the different iText methods that change the dash pattern.

Table 4.8: General graphics state operators

PDF	iText	Parameters	Description
`ri`	`setRenderingIntent`	`(intent)`	Sets the color rendering intent. The value is a name; possible values are `/AbsoluteColorimetric`, `/RelativeColorimetric`, `/Saturation`, and `/Perceptual`.
`i`	`setFlatness`	`(flatness)`	Sets the maximum permitted distance, in device pixels, between the mathematically correct path and an approximation constructed from straight line segments. This is a value between 0 and 100. Smaller values yield greater precision at the cost of more computation.
`gs`	`setGState`	`(gState)`	Sets a group of paramters in the graphics state using a graphics state parameter dictionary. Possible entries are listed in table 4.7.

We've discused five operators already. The rendering intent is used when CIE colors need to be rendered to device colors. The flatness indicates the level of tolerance when rendering paths. We'll discuss the `gs` operator in section 4.2.7.

4.2.5 Special graphics state operators

In previous code samples, we changed the graphics state and we've constructed and painted paths, but we skipped a couple of important operators, such as the operators that change the coordinate system and those who save and restore the graphics state stack. These are the 'special' graphics state operators.

4.2.5.1 Transforming the coordinate system

In previous examples, we always assumed:

- that the coordinate system has its origin in the lower-left corner,
- that the x axis has increasing x values from left to right, and
- that the y axis has increasing y values from bottom to top.

When we talked about page boundaries and page sizes in section 3.4.2, we discovered that the origin of the coordinate system can have a different location, depending on how the *MediaBox* was defined. In this section, we'll discuss another way to transform the coordinate system.

Let's take a look at figure 4.21, which is the screen shot of a page that contains five triangles.

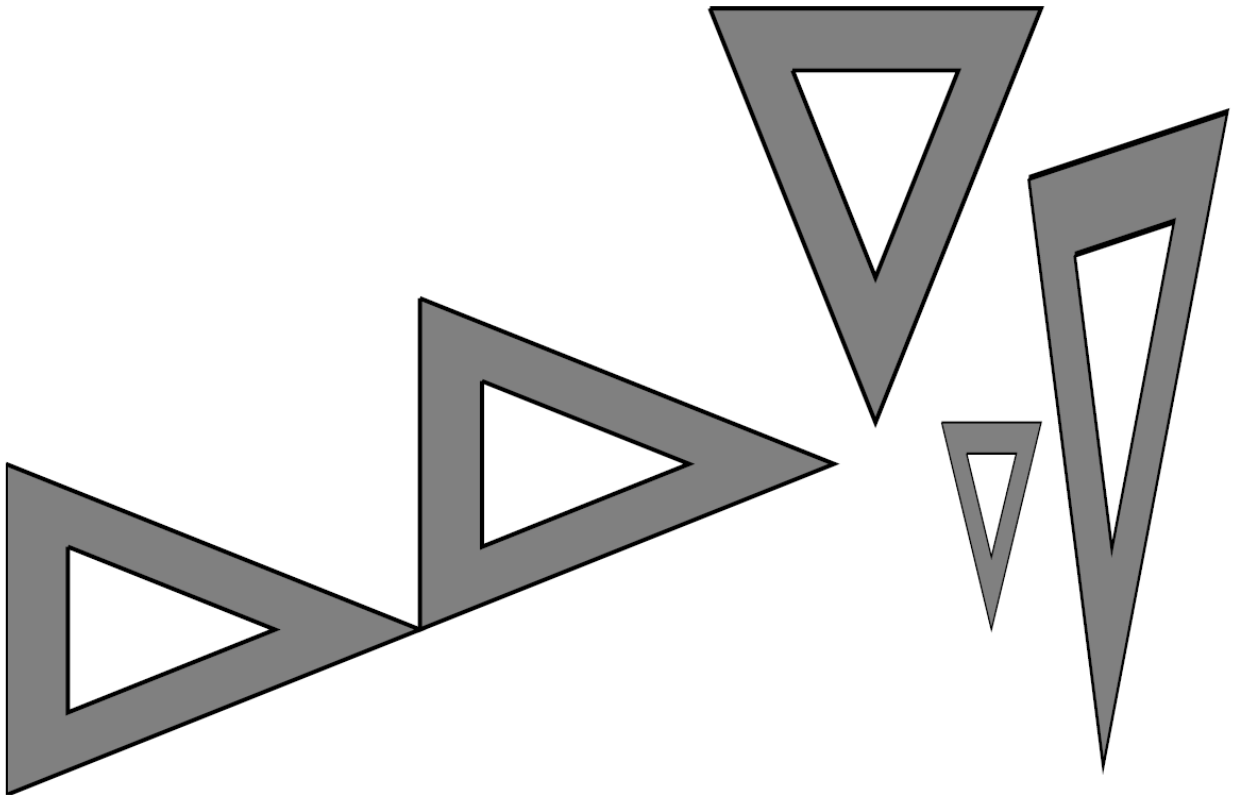


Figure 4.21: Coordinate system transformations

These five triangles are drawn using the exact same `triangle()` method. See listing 4.29.

Code sample 4.29: C0409_CoordinateSystem

```
protected void triangle(PdfContentByte canvas) {
    canvas.moveTo(0, 80);
    canvas.lineTo(100, 40);
    canvas.lineTo(0, 0);
    canvas.lineTo(0, 80);
    canvas.moveTo(15, 60);
    canvas.lineTo(65, 40);
    canvas.lineTo(15, 20);
    canvas.lineTo(15, 60);
    canvas.eoFillStroke();
}
```

The paths of the five triangles are identical, even when you look inside the PDF file using RUPS. However, when looking at them in a PDF viewer, the triangles are drawn at different positions, using a different scale or orientation. This is due to a changed coordinate system.

Listing 4.30 shows how the coordinate system was changed.

Code sample 4.30: C0430_CoordinateSystem

```
canvas.setColorFill(BaseColor.GRAY);
triangle(canvas);
canvas.concatCTM(1, 0, 0, 1, 100, 40);
triangle(canvas);
canvas.concatCTM(0, -1, -1, 0, 150, 150);
triangle(canvas);
canvas.concatCTM(0.5f, 0, 0, 0.3f, 100, 0);
triangle(canvas);
canvas.concatCTM(3, 0.2f, 0.4f, 2, -150, -150);
triangle(canvas);
```

The six values of the `concatCTM()` method are elements of a matrix that has three rows and three columns.

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

You can use this matrix to express a transformation in a two-dimentional system.

$$[x'\ y'\ 1] = [x\ y\ 1] = \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Carrying out this multiplication results in this:

```
x' = a * x + c * y + e
y' = b * x + d * y + f
```

The third column in the matrix is fixed: you're working in two dimensions, so you don't need to calculate a new z coordinate.

When studying analytical geometry in high school, you've probably learned how to apply transformations to objects. In PDF, we use a slightly different approach: instead of transforming objects, we transform the coordinate system.

By default the Current Transformation Matrix (CTM) is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The `concatCTM()` method changes the CTM by multiplying it with a new transformation matrix. In listing 4.30, we transform the coordinate system like this `canvas.concatCTM(1, 0, 0, 1, 100, 40)`:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\times\begin{bmatrix}1&0&0\\0&1&0\\100&40&1\end{bmatrix}=\begin{bmatrix}1&0&0\\0&1&0\\100&40&1\end{bmatrix}$$

As a result, the second triangle will be translated 100 user units to the right and 40 user units upwards. The next transformation `canvas.concatCTM(0, -1, -1, 0, 150, 150)` rotates by 90 degrees and translates the coordinate system:

$$\begin{bmatrix}1&0&0\\0&1&0\\100&40&1\end{bmatrix}\times\begin{bmatrix}0&-1&0\\-1&0&0\\150&150&1\end{bmatrix}=\begin{bmatrix}0&-1&0\\-1&0&0\\110&50&1\end{bmatrix}$$

The `concatCTM(0.5f, 0, 0, 0.3f, 100, 0)` transformation scales down using a different factor for the x and y axis, and introduces a translation in the x direction. As we've already rotated the coordinate system, it is perceived as a downward translation.

$$\begin{bmatrix}0&-1&0\\-1&0&0\\110&50&1\end{bmatrix}\times\begin{bmatrix}0.5&0&0\\0&0.3&0\\100&0&1\end{bmatrix}=\begin{bmatrix}0&-0.3&0\\-0.5&0&0\\155&15&1\end{bmatrix}$$

Finally, we scale, skew and translate: `concatCTM(3, 0.2f, 0.4f, 2, -150, -150)`.

$$\begin{bmatrix}0&-0.3&0\\-0.5&0&0\\155&15&1\end{bmatrix}\times\begin{bmatrix}3&0.2&0\\0.4&2&0\\-150&-150&1\end{bmatrix}=\begin{bmatrix}-0.12&-0.6&0\\-1.5&0.1&0\\321&-89&1\end{bmatrix}$$

The order in which the transformations of the CTM is important. If you change this order, you'll get a different result. In listing 4.31, we have switched two `concatCTM()` operations when compared to listing 4.30.

Code sample 4.31: C0409_CoordinateSystem

```
canvas.setColorFill(BaseColor.GRAY);
triangle(canvas);
canvas.concatCTM(1, 0, 0, 1, 100, 40);
triangle(canvas);
canvas.concatCTM(0.5f, 0, 0, 0.3f, 100, 0);
triangle(canvas);
canvas.concatCTM(0, -1, -1, 0, 150, 150);
triangle(canvas);
canvas.concatCTM(3, 0.2f, 0.4f, 2, -150, -150);
triangle(canvas);
```

If you'd multiply the matrices in that order, the final CTM will be:

$$\begin{bmatrix}-0.2&-1&0\\-0.9&-0.06&0\\264&-122.4&1\end{bmatrix}$$

This result is different from what we had before, and that is also shown in figure 4.21. The first couple of triangles are identical to the corresponding triangles in figure 4.20, but the final triangle is quite different because we switch the rotating and the scaling operation.

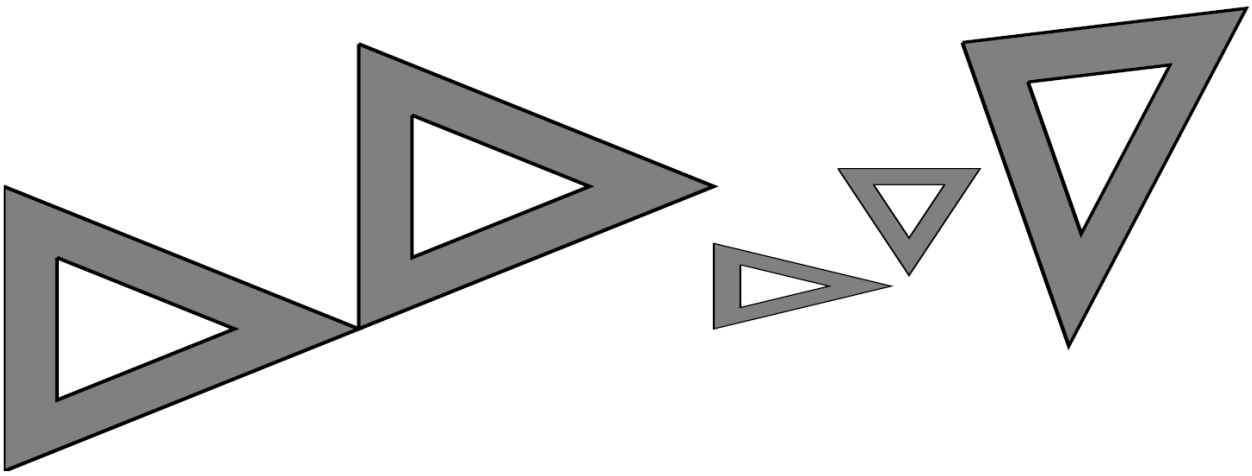


Figure 4.22: Coordinate system transformations

The `concatCTM()` method is the iText equivalent of the `cm` operator in PDF. All coordinates used after a transformation took place are expressed in the transformed coordinate system. Switching back to the original (or a previous) coordinate system could be achieved by calculating a new transformation matrix —the inverse matrix of the CTM—, but there's a much easier way to achieve the same result. That's what the saving and restoring the graphics state stack is about.

4.2.5.2 Saving and restoring the graphics state stack

When we talk about graphics state, we refer to an internal data structure that holds current graphics control parameters. These parameters have an impact on the graphics objects we draw. Figure 4.23 shows some more triangles that demonstrate differences on the graphics state level.

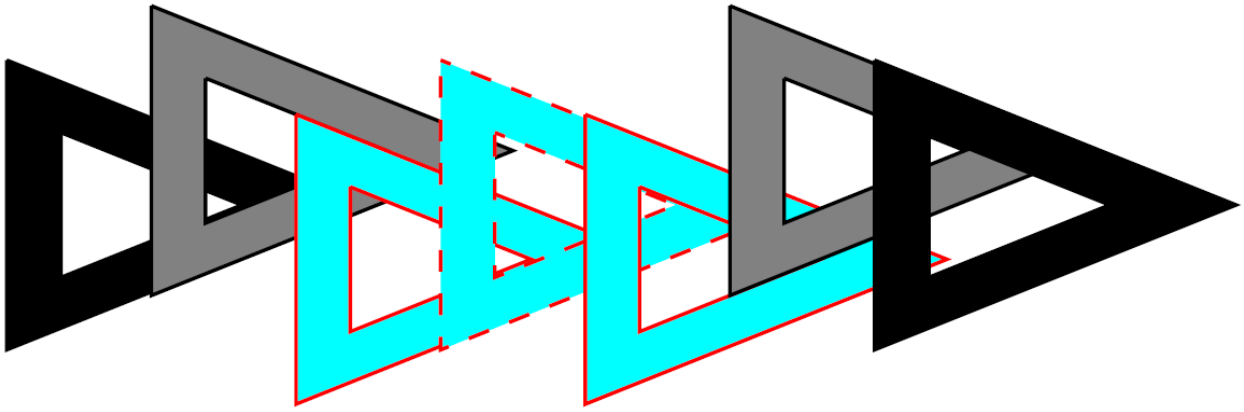


Figure 4.23: Graphics State Stack

The path of each triangle is constructed using the code from listing 4.32.

Code sample 4.32: C0410_GraphicsState

```
protected void triangle(PdfContentByte canvas, float x) {
    canvas.moveTo(x, 760);
    canvas.lineTo(x + 100, 720);
    canvas.lineTo(x, 680);
    canvas.lineTo(x, 760);
    canvas.moveTo(x + 15, 740);
    canvas.lineTo(x + 65, 720);
    canvas.lineTo(x + 15, 700);
    canvas.lineTo(x + 15, 740);
    canvas.eoFillStroke();
}
```

Not all triangles have the same appearance because we changed the graphics state before filling and stroking the paths of each individual triangle.

Code sample 4.33: C0410_GraphicsState

```
triangle(canvas, 50);
canvas.saveState();
canvas.concatCTM(1, 0, 0, 1, 0, 15);
canvas.setColorFill(BaseColor.GRAY);
triangle(canvas, 90);
canvas.saveState();
canvas.concatCTM(1, 0, 0, 1, 0, -30);
canvas.setColorStroke(BaseColor.RED);
canvas.setColorFill(BaseColor.CYAN);
triangle(canvas, 130);
canvas.saveState();
canvas.setLineDash(6, 3);
canvas.concatCTM(1, 0, 0, 1, 0, 15);
triangle(canvas, 170);
canvas.restoreState();
triangle(canvas, 210);
canvas.restoreState();
triangle(canvas, 250);
canvas.restoreState();
triangle(canvas, 290);
```

The PDF syntax generated with this iText code looks like this:

```
50 760 m 150 720 l 50 680 l 50 760 l 65 740 m 115 720 l 65 700 l 65 740 l B*
q
1 0 0 1 0 15 cm
0.50196 0.50196 0.50196 rg
90 760 m 190 720 l 90 680 l 90 760 l 105 740 m 155 720 l 105 700 l 105 740 l B*
q
1 0 0 1 0 -30 cm
1 0 0 RG
0 1 1 rg
130 760 m 230 720 l 130 680 l 130 760 l 145 740 m 195 720 l 145 700 l 145 740 l B*
q
[6] 3 d
1 0 0 1 0 15 cm
170 760 m 270 720 l 170 680 l 170 760 l 185 740 m 235 720 l 185 700 l 185 740 l B*
Q
210 760 m 310 720 l 210 680 l 210 760 l 225 740 m 275 720 l 225 700 l 225 740 l B*
Q
250 760 m 350 720 l 250 680 l 250 760 l 265 740 m 315 720 l 265 700 l 265 740 l B*
Q
290 760 m 390 720 l 290 680 l 290 760 l 305 740 m 355 720 l 305 700 l 305 740 l B*
```

Now let's explain this syntax, line by line:

- In line 1, we add a triangle with an offset of 50 user units. We didn't change the state, which means the fill color as well as the stroke color are black.
- We save the state in line 2 and change the current state in lines 3 and 4. In line 3 we add an upward translation. In line 4, we change the fill color to gray. We draw another triangle with horizontal offset 90 in line 5. Now we have a gray triangle with black borders. Due to the transformation of the CTM, it's no longer at the same height as the first triangle.
- We save the state in line 6. We perform another transformation in line 7. We change the stroke color in line 8 and the fill color in line 9. We add a third triangle with offset 130 in line 10. We now have a cyan triangle with a red border that is displayed at a lower y coordinate than the previous ones.
- We save the state once more in line 11. We introduce a dash pattern in line 12 and a CTM transformation that brings the CTM back to the default CTM in line 13. We add a triangle with offset 170 in line 14. It's a cyan triangle with a dashed, red border.
- In line 15, we restore the graphics state to the previous state in the stack. That is: to the situation before line 11. We add a triangle with offset 210 in line 16. This triangle looks identical to the third triangle added in line 10, because we're using the exact same graphics state.
- In line 17, we restore the graphics state to the situation that was in place before line 6. The triangle with offset 250 that is added in line 18 is drawn using the same graphics state as the second triangle added in line 5.
- With our final restore operation in line 19, we return to the default graphics state stack. The triangle with offset 290 from line 20 has black borders and is filled in black.

The `saveState()` and `restoreState()` method introduce `q` and `Q` operators. These operators should always be balanced.

FAQ: Why am I getting an InvalidPdfSyntaxException saying *Unbalanced save/restore state operators*?

You can't introduce `restoreState()` before you've used the `saveState()` method. In PDF syntax: you can't have a `Q` without a preceding `q`. For every `saveState()`, you must have a `restoreState()` operator somewhere in the same content stream. In other words: for every `q` there must be at least one `Q`. This exception tells you this isn't the case.

In this context, the same content stream doesn't necessarily mean the same stream object. When we discussed the page dictionary, we explained that the value of the `/Contents` entry can either be a reference to a stream or an array. If it's an array, the elements consist of references to streams that need to be concatenated when rendering the page content. In this case, you can have a `q` in one stream, and a `Q` in the next one. When we say that the save and restore operators need to be balanced, we refer to the resulting stream, not to each separate stream in the array.

Note that each new page starts with a new, empty graphics state stack. If you changed the state on one page, those changes won't be transferred to the next page automatically. The new page starts with default values for the graphics state. We can use the `gs` operator to reuse a specific graphics state, but before we do so, let's take a look at the overview of the special graphics state operators.

4.2.5.3 Overview of the special graphics state operators

Table 4.9 summarizes the methods and operators we've discussed in this section.

Table 4.9: Special graphics state operators

PDF	iText	Parameters	Description
`cm`	`concateCTM`	`(a, b, c, d, e, f)`	Modifies the current transformation matrix (CTM) by concatenating the matrix defined by a, b, c, d, e, and f.
`q`	`saveState`	`()`	Saves the current graphics state on the graphics state stack.
`Q`	`restoreState`	`()`	Restores the graphics state by removing the most recently saved state from the stack, making it the current state.

When we look at the PDF syntax that was used to draw the triangles in figure 4.23, we see that the line painting the path of the triangles is repeated over and over again, although using different operators. Let's find a way to optimize this syntax by introducing an external object, also known as an XObject.

4.2.6 XObjects

An external object is an object that is defined outside the content stream and referenced as a named resource. When we discussed page dictionaries and more specifically the resources dictionary, we already encountered such an XObject. We can distinguish two major types of XObjects:

- a form XObject is an entire content stream to be treated as a single graphics object,
- an image XObject defines a rectangular array of color samples to be painted.

Let's start with an example of a form XObject.

4.2.6.1 Form XObjects

Figure 4.24 shows the page dictionary of a page that contains an external object.

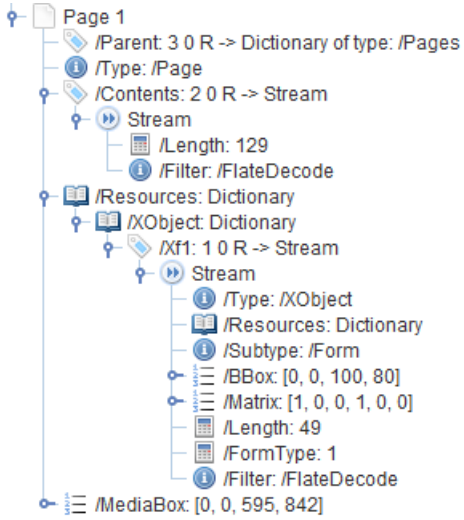


Figure 4.24: Page dictionary of a page with an XObject

The indirect object with object number 1 is a stream that is defined as a form XObject. We recognize a bounding box and a transformation matrix. Note that we could have omitted the `/FormType` entry; 1 is the default type, but also the only possible type that is currently available in the PDF specification.

The content of the stream looks like this:

```
0 80 m
100 40 l
0 0 l
0 80 l
15 60 m
65 40 l
15 20 l
15 60 l
B*
```

The stream is referenced in the resources dictionary of the page using the name `/Xf1`.

If there were more pages, it could have been referenced by the same name or by any other name in the resources dictionaries of those other pages.

In the content stream of this page however, we'll find references to `/Xf1`:

```
1  q 1 0 0 1 50 680 cm /Xf1 Do Q
2  q
3  1 0 0 1 0 15 cm
4  0.50196 0.50196 0.50196 rg
5  q 1 0 0 1 90 680 cm /Xf1 Do Q
6  q
7  1 0 0 1 0 -30 cm
8  1 0 0 RG
9  0 1 1 rg
10 q 1 0 0 1 130 680 cm /Xf1 Do Q
11 q
12 [6] 3 d
13 1 0 0 1 0 15 cm
14 q 1 0 0 1 170 680 cm /Xf1 Do Q
15 Q
16 q 1 0 0 1 210 680 cm /Xf1 Do Q
17 Q
18 q 1 0 0 1 250 680 cm /Xf1 Do Q
19 Q
20 q 1 0 0 1 290 680 cm /Xf1 Do Q
```

These 20 lines correspond with the 20 lines of PDF syntax we had before, except for the fact that we now use the `Do` operator and the `/Xf1` operand to draw the triangles. We position the form XObject using a `cm` operator and we use `q` / `Q` to make sure we don't change the coordinate system permanently.

The iText code corresponding with this PDF syntax is shown in listing 4.34:

Code sample 4.34: C0411_GraphicsState

```
PdfContentByte canvas = writer.getDirectContent();
PdfTemplate template = canvas.createTemplate(100, 80);
template.moveTo(0, 80);
template.lineTo(100, 40);
template.lineTo(0, 0);
template.lineTo(0, 80);
template.moveTo(15, 60);
template.lineTo(65, 40);
template.lineTo(15, 20);
template.lineTo(15, 60);
template.eoFillStroke();
canvas.addTemplate(template, 50, 680);
canvas.saveState();
canvas.concatCTM(1, 0, 0, 1, 0, 15);
canvas.setColorFill(BaseColor.GRAY);
canvas.addTemplate(template, 90, 680);
canvas.saveState();
canvas.concatCTM(1, 0, 0, 1, 0, -30);
canvas.setColorStroke(BaseColor.RED);
canvas.setColorFill(BaseColor.CYAN);
canvas.addTemplate(template, 130, 680);
canvas.saveState();
canvas.setLineDash(6, 3);
canvas.concatCTM(1, 0, 0, 1, 0, 15);
canvas.addTemplate(template, 170, 680);
canvas.restoreState();
canvas.addTemplate(template, 210, 680);
canvas.restoreState();
canvas.addTemplate(template, 250, 680);
canvas.restoreState();
canvas.addTemplate(template, 290, 680);
```

Images can be added in the exact same way. Instead of a stream of PDF syntax, we'll have a compressed stream of pixel values.

4.2.6.2 PDF and images

In figure 4.25, we see a PDF showing three light bulbs. The original image for the single light bulb was bulb.gif.

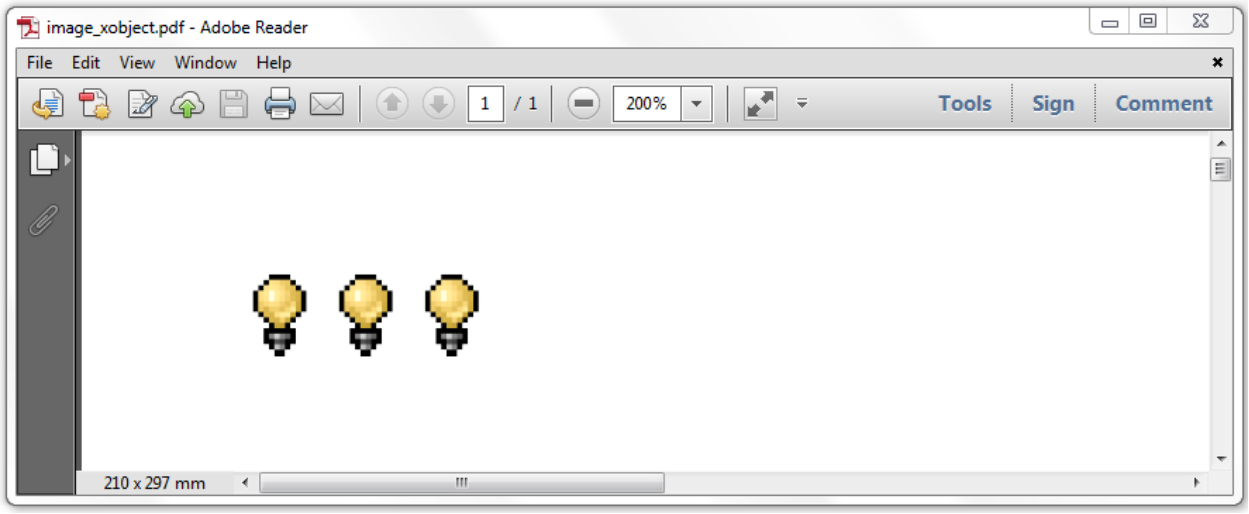


Figure 4.25: Images in PDF

In the ideal situation, the image bytes are stored in the PDF only once. This requires that the image is added as an XObject as shown in figure 4.26.

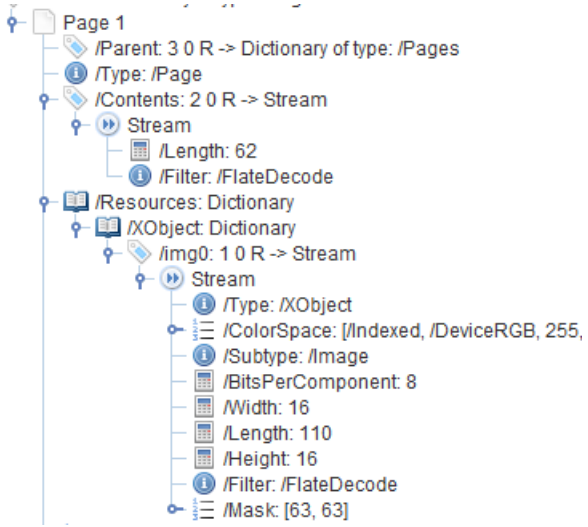


Figure 4.26: Image as XObject

The stream object with object number 1 contains an image with a width and a height of 16 pixels. Each component consists of 8 bits and we're using an Indexed colorspace with a selection of 256 RGB colors. The sequence of color values is compressed to 110 bytes.

The page stream looks like this:

```
q 20 0 0 20 36 786 cm /img0 Do Q
q 20 0 0 20 56 786 cm /img0 Do Q
q 20 0 0 20 76 786 cm /img0 Do Q
```

The alternative is to add the image inline. In that case, there is no XObject, but the bytes that define the color

space and the image are repeated in the content stream:

```
q 20 0 0 20 36 786 cm
BI
/CS [/Indexed/DeviceRGB 255(***binary stuff**)]
/BPC 8 /W 16 /H 16 /F /FlateDecode /L 110
ID
***binary stuff***
EI
Q
q 20 0 0 20 56 786 cm
BI
/CS [/Indexed/DeviceRGB 255(***binary stuff***)]
/BPC 8 /W 16 /H 16 /F /FlateDecode /L 110
ID
***binary stuff***
EI
Q
q 20 0 0 20 76 786 cm
BI
/CS [/Indexed/DeviceRGB 255(***binary stuff***)]
/BPC 8 /W 16 /H 16 /F /FlateDecode /L 110
ID
***binary stuff***
EI
Q
```

Note that we didn't print the binary values of the colorspace and the compressed image bytes in this PDF snippet. An inline image starts with the `BI` operator, following by a series of key value pairs. Then there's the `ID` operator followed by the image bytes. The inline image object is closed with the `EI` operator.

This snippet also introduces a value that will be introduced in ISO-32000-2 (PDF 2.0). In ISO-32000-1, there is no `/L` value for the length. Without this value, a PDF parser needs to search for the `EI` operator and the white space delimiters for that operator to find the end of the image. While this will work for *most* images, this won't work for *all* images, more specifically for images for which the binary data contains a sequence *

ISO-32000-1 recommended not to use inline images with a length higher than 4 KB. ISO-32000-2 makes this normative: the value for `/L` shall not exceed 4096.

The difference in iText code is minimal. Listing 4.35 shows the solution that uses image XObjects; listing 4.36 shows the solution that uses inline images.

Code sample 4.35: C0411_GraphicsState

```
PdfContentByte canvas = writer.getDirectContent();
Image img = Image.getInstance(IMG);
canvas.addImage(img, 20, 0, 0, 20, 36, 786);
canvas.addImage(img, 20, 0, 0, 20, 56, 786);
canvas.addImage(img, 20, 0, 0, 20, 76, 786);
```

Code sample 4.36: C0411_GraphicsState

```
Image img = Image.getInstance(IMG);
canvas.addImage(img, 20, 0, 0, 20, 36, 786, true);
canvas.addImage(img, 20, 0, 0, 20, 56, 786, true);
canvas.addImage(img, 20, 0, 0, 20, 76, 786, true);
```

iText supports JPEG, JPEG2000, GIF, PNG, BMP, WMF, TIFF, CCITT and JBIG2 images. This doesn't mean that these images types are also supported in PDF.

- JPEG images are kept as is by iText. You can take the content stream of an Image XObject of type JPEG, copy it into a file and you'll have a valid JPEG image. You can recognize these images by their filter: `/DCTDecode`.
- JPEG2000 is supported since PDF 1.5. The name of the filter is `JPXDecode`.
- Although PDF supports images with LZW compression (used for GIFs), iText decodes GIF images into a raw image. If you create an `Image` in iText with a path to a GIF file, you'll get an image with filter `/FlateDecode` in your PDF.
- PNG isn't supported in PDF, which is why iText will also decode PNG images into rw images. If the color space of the image is DeviceGray and if the image only has 1 bit per component, CCITT will be used as compression and you'll recognize the filter `/CCITTFaxDecode`. Otherwise, the filter `/FlateDecode` will be used.
- BMP files will be stored as a series of compressed pixels using `/FlateDecode` as filter.
- WMF is special. If you insert a WMF file into a PDF document using iText, iText will convert that image into PDF syntax. Instead of adding an Image XObject, iText will create a form XObject.
- When the image data is encoded using the CCITT facsimile standard, the `/CCITTFaxDecode` filter will be used. These are typically monochrome images with one bit per pixel.
- TIFFs will be examined by iText. Depending on the TIFFs parameters, iText can decide to use `/CCITTFaxDecode`, `FlateDecode` or even `DCTDecode` as filter.
- JBIG2 uses the `/JBIG2Decode` filter.

Normally, you don't need to worry about the image type. The `Image` class takes care of choosing the right compression method for you.

4.2.6.3 Overview of the XObject and image operators

Table 4.10 is somewhat different than the tables we had before. The `Do` operator can be introduced using the `addTemplate()` method and the `addImage()` method. Using the `addImage()` method can introduce either a `q cm Do Q` sequence or a `BI ID EI` sequence.

Table 4.10: form XObject and Image operators

PDF	iText methods	Description
`Do`	`addTemplate(template, e, f)`, `addTemplate(template, a, b, c, d, e, f)`	The operator `Do`, preceded by a name of a form XObject, such as `/Xf1`, paints the XObject. iText will take care of handling the template object, as well as saving the state, performing a transformation of the CTM that's used for adding the XObject, and restoring the state.
`Do`	`addImage(template)`, `addImage(image, false)`, `addImage(image, a, b, c, d, e, f)`, `addImage(image, a, b, c, d, e, f, false)`	The operator `Do`, preceded by the name of an image XObject, such as `Img0`, paints the image. iText will take care of storing the image stream correctly, as well as saving the state, performing a transformation of the CTM, and restoring the state.
`BI` / `ID` / `EI`	`addImage(image, true)`, `(addImageimage, a, b, c, d, e, f, true)`	Inline images are enclosed by the `BI` and `EI` operator. The `ID` operator marks where the actual image data begins. These operators should not be used for images larger than 4096 bytes.

We used XObjects to reuse large snippets of PDF code or images. Now let's take a look at the graphics state dictionary that allows us to reuse graphics state parameters.

4.2.7 Graphics state dictionary

Suppose that you want to draw the triangle shape we used before with a different line width, line join and dash pattern. See for instance figure 4.27.

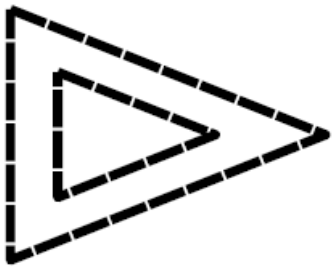


Figure 4.27: Triangle with different line width, line join and dash pattern

You could change the graphics state like we did before with the `setLineWidth()`, `setLineJoin()` and `setDashPattern()` methods, but the moment you start a new page, the state is lost. If you draw the same shape on the next page, it looks like this:

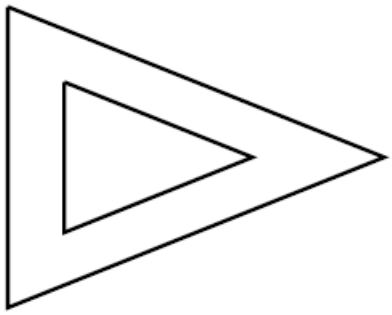


Figure 4.28: Ordinary triangle

The graphics state dictionary allows you to reuse graphics state as a resource.

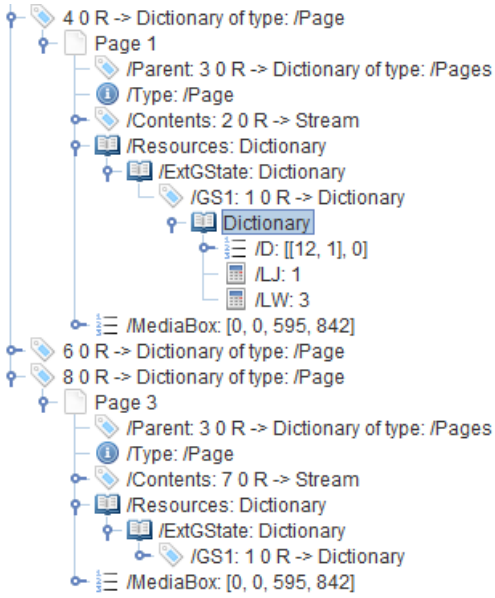


Figure 4.29: Graphics State dictionary

Object 1 is a dictionary with three entries, `/D` for the dash pattern (`[[12 1] 0]`), `/LJ` for the line join (`1`) and `/LW` for the line width (`3`). This object is used on page 1 and page three like this:

```
/GS1 gs
50 680 m 150 640 l 50 600 l 50 680 l
65 660 m 115 640 l 65 620 l 65 660 l
S
```

The line `/GS1 gs` set the line width, line join and dash pattern all at once. Listing 4.37 shows how to create a graphics state dictionary:

Code sample 4.37: C0411_GraphicsState

```
1  PdfContentByte canvas = writer.getDirectContent();
2  PdfGState gs = new PdfGState();
3  gs.put(new PdfName("LW"), new PdfNumber(3));
4  gs.put(new PdfName("LJ"), new PdfNumber(1));
5  PdfArray dashArray = new PdfArray(new int[]{12, 1});
6  PdfArray dashPattern = new PdfArray();
7  dashPattern.add(dashArray);
8  dashPattern.add(new PdfNumber(0));
9  gs.put(new PdfName("D"), dashPattern);
10 canvas.setGState(gs);
11 triangle(canvas);
12 document.newPage();
13 triangle(canvas);
14 document.newPage();
15 canvas.setGState(gs);
16 triangle(canvas);
```

In this code snippet, we draw the triangles three times (on lines 11, 13 and 16) on three different pages. The graphics state dictionary is for the triangles on pages 1 and 3. These triangles look like figure 4.27. The triangles on page 2 look like figure 4.28.

Table 4.8 shows a selection of possible entries of the entries in a graphics state dictionary. For more entries, see table 58 in ISO-32000-1.

Table 4.8: Entries in a graphics state dictionary

Name	iText method	Parameter	Description
/LW		PdfNumber	The line width of the graphics state.
/LC		PdfNumber	The line cap style of the graphics state.
/LJ		PdfNumber	The line join style of the graphics state.
/ML		PdfNumber	The miter limit of the graphics state.
/D		PdfArray	The line dash pattern of the graphics state. The pattern is expressed as an array of the form [dashArray dashPhase] where dashArray is itself a PdfArray and dashPhase is a PdfNumber.
/RI	setRenderingIntent	(ri)	The parameter is the name of the rendering intent (see table 4.7).

Table 4.8: Entries in a graphics state dictionary

Name	iText method	Parameter	Description
/op	setOverPrintNonStroking	(op)	The parameter is a Boolean value that specifies whether or not to apply overprint for painting operations other than stroking. If the entry is absent, this parameter will also be set by the /OP entry (if present).
/OP	setOverPrintStroking	(op)	The parameter is a Boolean value that specifies whether or not to apply overprint. If there's also an /op entry in the dictionary, the /OP entry will only set the parameter for stroking operations.
/OPM	setOverPrintMode	(opm)	The parameter is an integer value, either 0 or 1. It specifies the overprint mode and it's only taken into account if the overprint parameter is true. It controls the tint value in the context of DeviceCMYK colors.
/FL		PdfNumber	Specifies the flatness tolerance.
/SM		PdfNumber	Specifies the smoothness tolerance.
/SA		PdfBoolean	Specifies the stroke adjustment.
/BM	setBlendMode	(bm)	The parameter is a name that specifies the current blend mode that will be used.
/ca	setFillOpacity	(ca)	The parameter is a float value that specifies the opacity of the shapes that are painted in the transparent imaging model.
/CA	setStrokeOpacity	(ca)	The parameter is a float value that specifies the opacity of the path that are stroked in the transparent imaging model.
/AIS	setAlphaIsShape	(ais)	The parameter is a Boolean value that specifies whether the current soft mask and alpha constant must be interpreted as shape values (true) or opacity values (false).
/TK	setTextKnockout	(tk)	The parameter is a Boolean value that determines the behavior of overlapping glyphs within a text object in the transparant imaging model.

When looking at table 4.8, we see a series of entries introducing transparency. Let's take a closer look at these entries in the next section.

4.2.8 Graphics state and transparency

The chapter on transparency in ISO-32000-1 is about 40 pages long. Using snippets from that chapter, one could summarize it as follows:

> A given object shall be composited with a backdrop. Ordinarily, the backdrop consists of the stack of all objects that have been specified previously. The result of the compositing shall then be treated as the backdrop for the next object. However, within certain kinds of transparancy groups, a different backdrop may be chosen.
>
> During the compositing of an object with its backdrop, the color at each point shall be computed using a specified blend mode, which is a function of both the object's color and the backdrop color …
>
> Two scalar quantities called shape and opacity mediate compositing of an object with its backdrop … Both shape and opacity vary from 0.0 (no contribution) to 1.0 (maximum contribution) … Shape and opacity are conceptually very similar. In fact, they can usually be combined into a single value, called alpha, which controls both the color compositing computation and the fading between an object and its backdrop. However, there are a few situations in which they shall be treated separately; see knockout groups.

In the next couple of examples, we'll explain concepts such as transparency, transparency groups, isolation and knockout using a couple of simple examples.

4.2.8.1 Transparency

Let's take a look at figure 4.30. In both cases, the backdrop consists of a square of which half is painted gray. On this backdrop, we add three full circles in a specific order: red, yellow, blue.

In the figure to the left, the red circle covers part of the backdrop, the yellow circle covers part of the backdrop and part of the red circle, the blue circle covers part of the backdrop, part of the red circle and part of the yellow circle. There is no transparency involved. The opacity is 1.

In the figure to the right, we have introduced an opacity of 0.5 for the circles. This makes the circles transparent. The colors are mixed where the circles overlap, and the color of the circles is blended with the color of the backdrop.

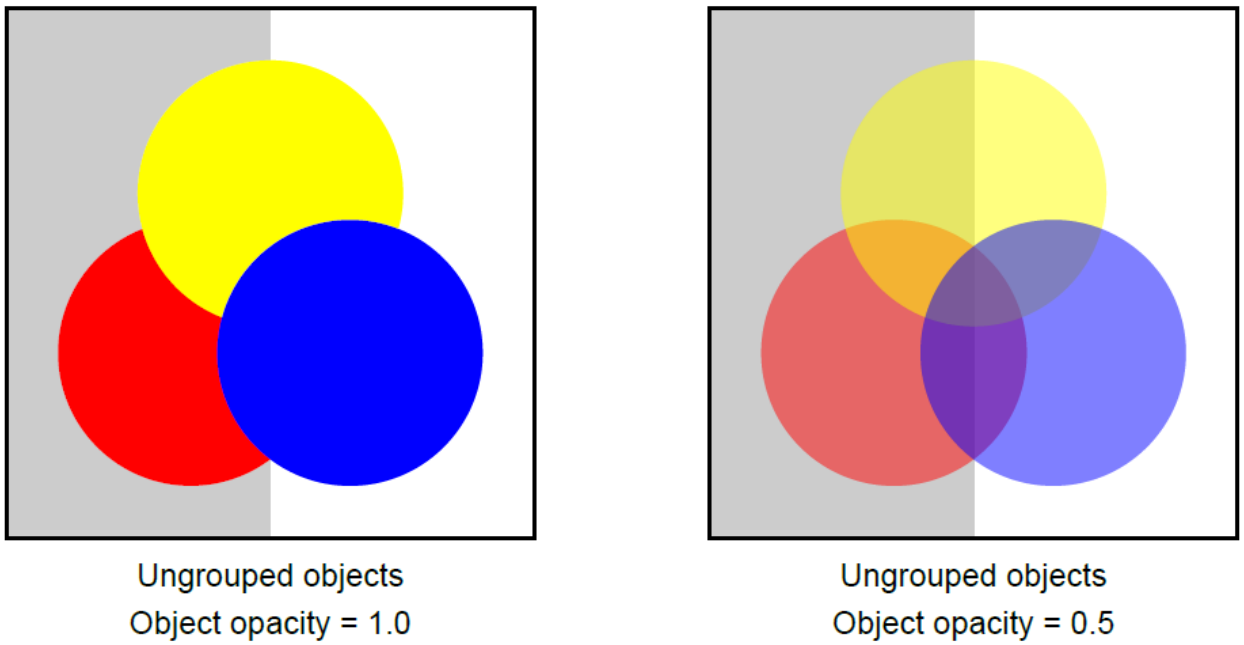


Figure 4.30: Opaque circles, transparent circles

Code sample 4.38 shows how the transparency was introduced.

Code sample 4.38: C0415_TransparencyGroups

```
1  PdfGState gs1 = new PdfGState();
2  gs1.setFillOpacity(0.5f);
3  cb.setGState(gs1);
4  drawCircles(200 + 2 * gap, 500, cb);
```

We can also define the transparency at the level of a group of objects.

4.2.8.2 Transparency Groups

Looking at figure 4.31, we see three circles that aren't transparent among each other to the left. As a group they are made transparent against the backdrop. To the right, we see that the circles are transparent as objects, but there's no extra transparency as a group. We've also introduced a special blend mode for the circles to the right.

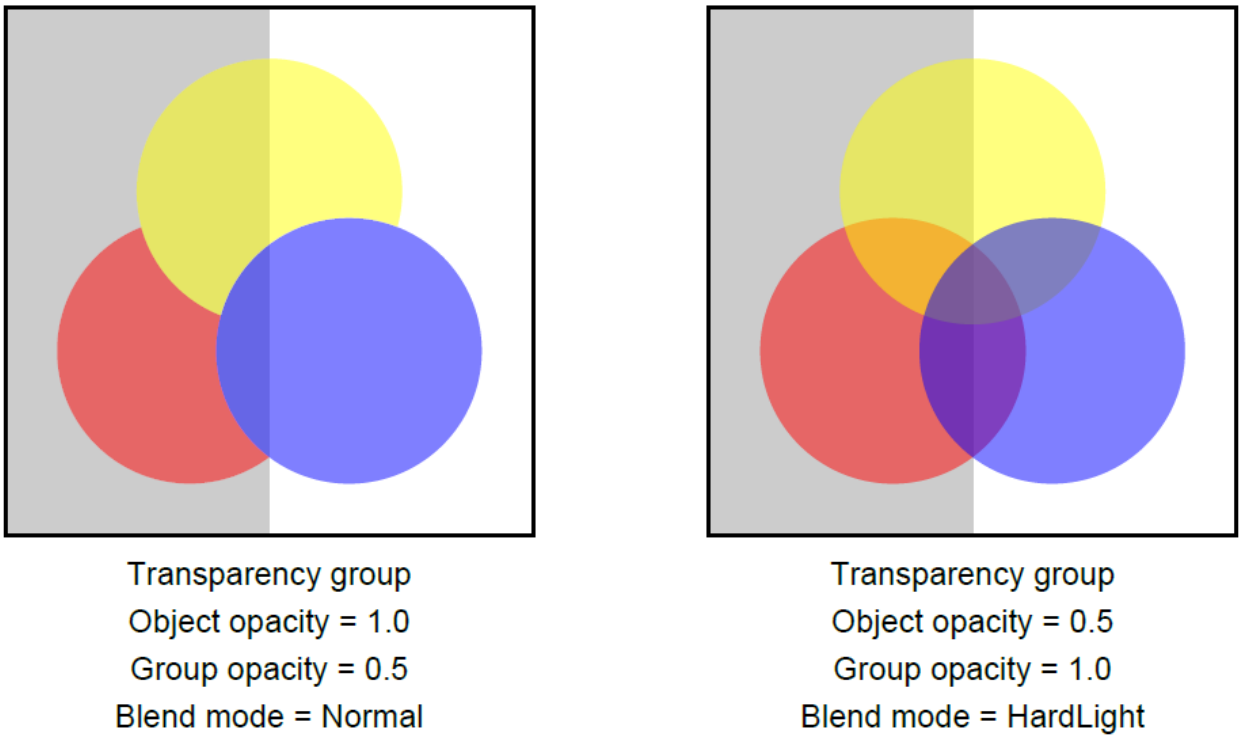


Figure 4.31: Transparency groups

Code sample 4.39 demonstrates that the transparency groups were defined using a Form XObject.

Code sample 4.39: C0415_TransparencyGroups

```
1  cb.saveState();
2  PdfTemplate tp = cb.createTemplate(200, 200);
3  PdfTransparencyGroup group = new PdfTransparencyGroup();
4  tp.setGroup(group);
5  drawCircles(0, 0, tp);
6  cb.setGState(gs1);
7  cb.addTemplate(tp, gap, 500 - 200 - gap);
8  cb.restoreState();
9  cb.saveState();
10 tp = cb.createTemplate(200, 200);
11 tp.setGroup(group);
12 PdfGState gs2 = new PdfGState();
13 gs2.setFillOpacity(0.5f);
14 gs2.setBlendMode(PdfGState.BM_HARDLIGHT);
15 tp.setGState(gs2);
16 drawCircles(0, 0, tp);
17 cb.addTemplate(tp, 200 + 2 * gap, 500 - 200 - gap);
18 cb.restoreState();
```

In line 1 to 8, we create a `PdfTemplate` object on which we draw the circles. We define a `PdfTransparencyGroup`

and we use the `setGroup()` method to indicate that all objects of the Form XObject belong to this group. We change the general graphics state stat by reusing the `gs1` object from code sample 4.38.

In line 9 to 18, we create another `PdfTemplate` and another `PdfGState` introducing a different blend mode (HardLight). This time, we use the `setGState()` method on the level of the Form XObject instead of on the general graphics state. This explains the difference in result shown in figure 4.31.

4.2.8.3 Isolation and knockout

The `PdfTransparencyGroup` class has two methods: `setIsolated()` and `setKnockout()`. Both methods expect a Boolean value as parameter. Figure 4.32 shows all possible combinations.

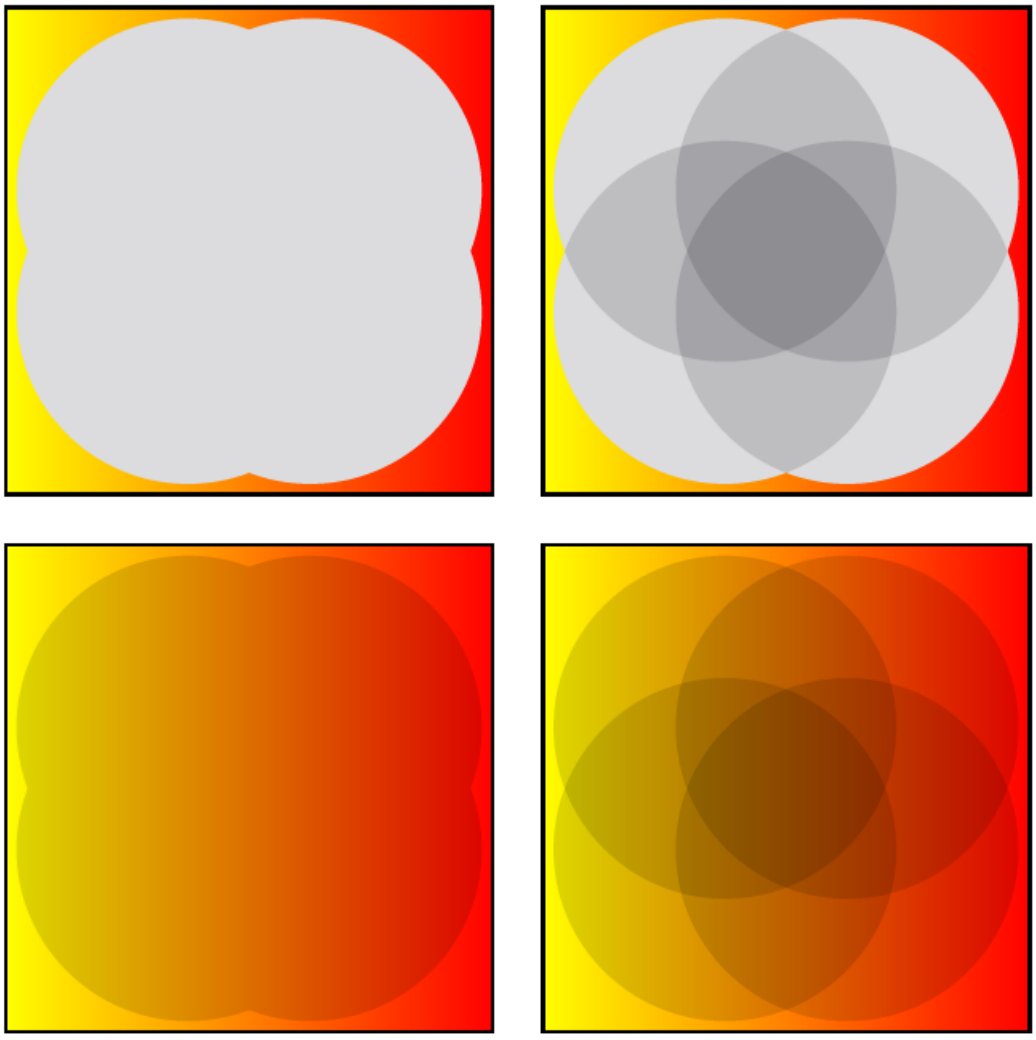


Figure 4.32: Isolation and knockout

The code to draw the four figures is identical. The backdrop is a square with an axial shading going from yellow to red. Four circles are drawn against this backdrop. All the circles have the same CMYK color: C, M and Y are set to 0 and K to 0.15. The opacity is 1 and the blend mode is *multiply*; the only difference is the *isolation* and the *knockout* mode.

- *Isolation*—For the two upper squares, the group is isolated: it doesn't interact with the backdrop. For the two lower squares, the group is nonisolated: the group composites with the backdrop.
- *Knockout*—For the squares at the left, knockout is set to `true`: the circles don't composite with each other. For the two on the right, it's set to `false`: they composite with each other.

Listing 4.40 shows how the upper-right figure was drawn. The other figures are created by changing the Boolean parameters in line 4 and 5.

Code sample 4.40: C0416_IsolationKnockout

```
1  PdfTemplate tp = cb.createTemplate(200, 200);
2  pictureCircles(0, 0, tp);
3  PdfTransparencyGroup group = new PdfTransparencyGroup();
4  group.setIsolated(true);
5  group.setKnockout(true);
6  tp.setGroup(group);
7  cb.addTemplate(tp, 50 + gap, 500);
```

The graphics state when drawing the circles was defined like this:

```
PdfGState gs = new PdfGState();
gs.setBlendMode(PdfGState.BM_MULTIPLY);
gs.setFillOpacity(1f);
cb.setGState(gs);
```

The PDF reference defines many other blend modes apart from *multiply*. You can find these blend modes in the `PdfGState` class. They all start with the prefix `BM_`. Feel free to experiment with some other values to find out how they are different.

We'll conclude this chapter by applying transparency to images.

4.2.9 Masking and clipping images

In section 4.2.6.2, we briefly discussed images, and we introduced the `Image` object without going into much detail. In this section, we'll introduce some concepts that are related to transparency. Let's start with the concept of image masks.

4.2.9.1 Hard masks and soft masks

In figure 4.33, we see an image of which parts are made fully transparent using a hard mask.

Figure 4.33: Hard image mask

In figure 4.34, we see an image that is gradually made transparent using a soft mask. The left side of the image is made completely transparent; the right side is completely opaque.

Figure 4.34: Soft image mask

If we take a look inside, we see the syntax for the hard image mask to the left and the syntax for the soft image mask to the right.

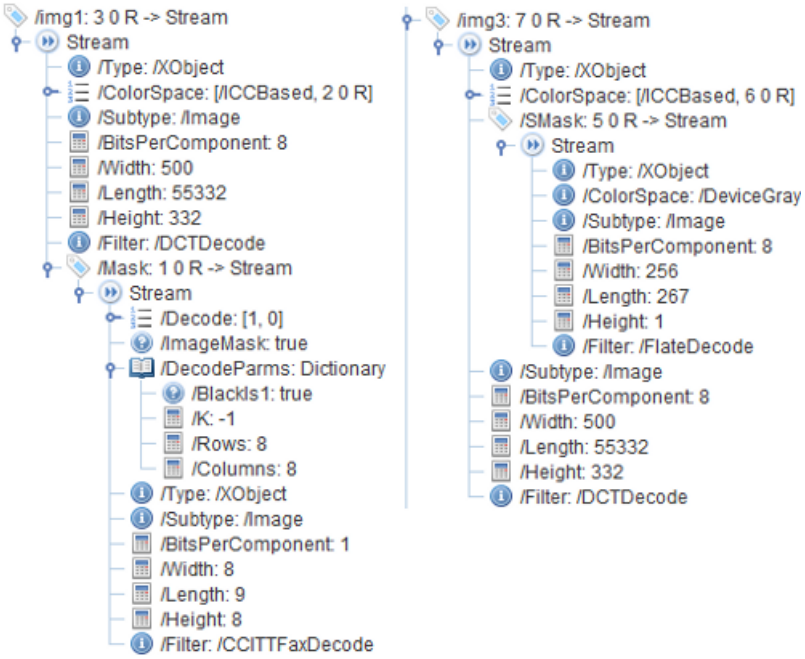


Figure 4.35: Hard and soft image masks: the syntax

To the right, we have a JPEG image (`/DCTDecode` filter) of which the stream dictionary has a `/Mask` entry that refers to an image XObject. This is called *stencil masking*. The value of the `/Mask` entry is an image of which the `/ImageMask` value is `true`. This image should be a monochrome image (the `/BitsPerComponent` value is `1`) that is treated as a stencil mark that is partly opaque and partly transparent. The number of pixels of the mask can be different from the number of pixels of the image it is masking. In our example, the JPEG measures 500 by 332 pixels, whereas the mask only measures 8 by 8 pixels.

To the left, we have a JPEG image of which the stream dictionary has an `/SMask` entry that refers to an image XObject of which the colorspace is `/DeviceGray`. The gray value of each pixel determines the opacity of the pixels that are being masked.

Let's take a look at the code that was used to produce the masked images shown in figures 4.33 and 4.34.

Code sample 4.41: C0417_ImageMask

```
1  public Image getImageHardMask() throws DocumentException, IOException {
2      byte circledata[] = { (byte) 0x3c, (byte) 0x7e, (byte) 0xff,
3              (byte) 0xff, (byte) 0xff, (byte) 0xff, (byte) 0x7e,
4              (byte) 0x3c };
5      Image mask = Image.getInstance(8, 8, 1, 1, circledata);
6      mask.makeMask();
7      mask.setInverted(true);
8      Image img = Image.getInstance(RESOURCE);
9      img.setImageMask(mask);
10     return img;
11 }
12 public Image getImageSoftMask() throws DocumentException, IOException {
```

```
13      byte gradient[] = new byte[256];
14      for (int i = 0; i < 256; i++)
15          gradient[i] = (byte) i;
16      Image mask = Image.getInstance(256, 1, 1, 8, gradient);
17      mask.makeMask();
18      Image img = Image.getInstance(RESOURCE);
19      img.setImageMask(mask);
20      return img;
21  }
```

Looking at listing 4.41, we see that we create images using raw bytes in lines 5 and 16.

In the first case, we create a byte array with a specific pattern, and we use the `getInstance()` method that accepts a width and a height (8 by 8), the number of components and the number of bits per component. We have one component of which the value can be either 1 or 0—this is a black and white image. the final parameter is the byte array. Note that we use the ´setInverted()´ method. This method defines which color needs to be used as stencil. In this case, we make sure the white part of the stencil is the part that will be made transparent.

In the second case, we have an image of 256 by 1 pixels. We still have one component, with 8 bits per component (a value between 0 and 255)—this is a gray color image. The bytes we pass as data consist of a gradient that varies between 0 and 255. Note that iText doesn't really require you to define whether or not the mask is a hard mask or a soft mask. This is determined by the nature of the image that is being used as mask.

Figure 4.36 shows two other cases in which the `/Mask` entry is used.

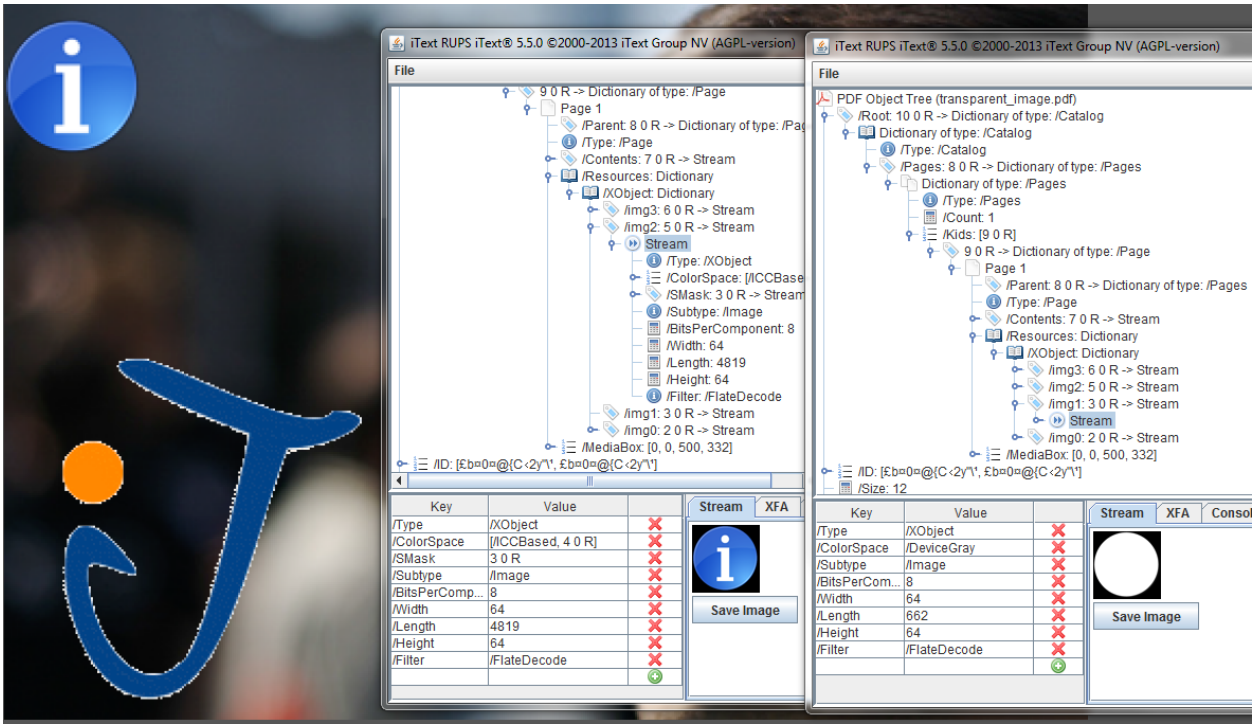


Figure 4.36: transparent images

In the upper-left corner, we see a circular image that was originally a transparent PNG image. PNG isn't supported in PDF, let alone transparent PNG files. When adding such a PNG to a document, iText creates an opaque bitmap (see object 5) as well as a mask for this image.

In the lower-left corner, we see a JPEG image that is also partly transparent. Figure 4.37 shows the corresponding syntax.

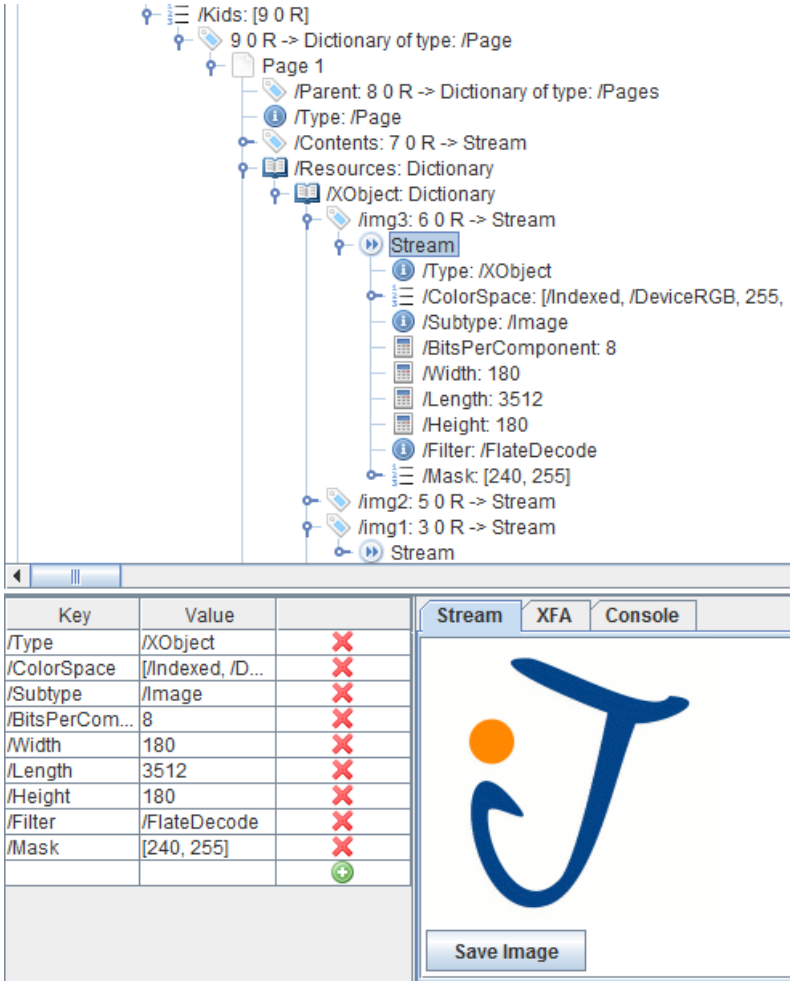


Figure 4.37: Color key masking

In this case, we have an image with an indexed color space (values from 0 to 255) and now we define the `/Mask` as an array of pairs that represent color ranges. In our case, we have a single pair ranging from color value 240 to color value 255. These colors will be transparent. Note that the result isn't always very nice, especially when applied to images with a lossy compression.

Code sample 4.42 shows how both images were added to the PDF.

Code sample 4.42: C0418_TransparentImage

```
Image img2 = Image.getInstance(RESOURCE2);
img2.setAbsolutePosition(0, 260);
document.add(img2);
Image img3 = Image.getInstance(RESOURCE3);
img3.setTransparency(new int[]{ 0xF0, 0xFF });
img3.setAbsolutePosition(0, 0);
document.add(img3);
```

The path named `RESOURCE2` refers to a transparent PNG image. This PNG is implicitly converted to two images by iText. The color key masking for `RESOURCE3` is defined using the `setTransparency()` method.

In the final section of this chapter, we'll look at another way to make part of an image transparent: we'll clip the image.

4.2.9.2 Clipping images

In figure 4.38, you see a picture of my wife and me at the film festival in Ghent. To the right, you see the same file opened in iText RUPS. When looking at the image using RUPS, you see that the original image stored in the PDF is larger than expected. You can see that we're standing at a desk.

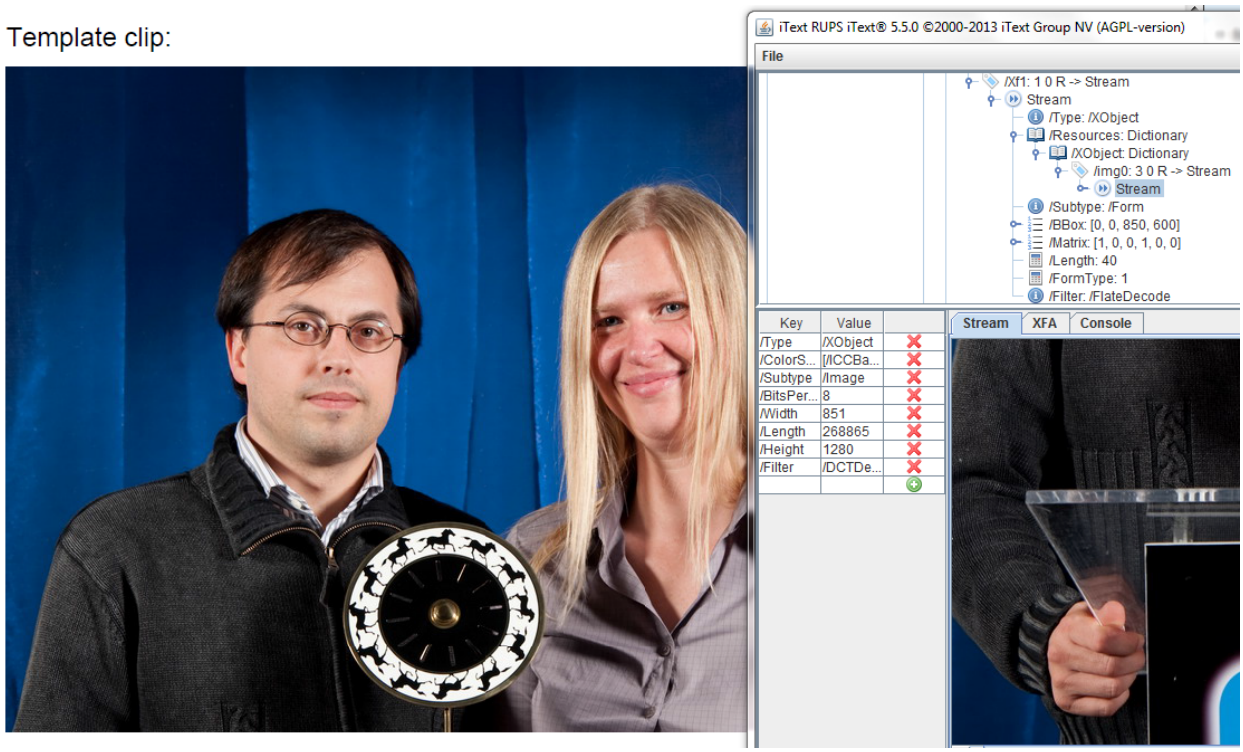


Figure 4.38: Template clipping

Looking more closely, we see that the image consists of 851 by 1280 pixels. It's a resource of a Form XObject (`/Xf1`) with a bounding box of 850 by 600 user units. This bounding box clips the image. Code sample 4.43 shows how it's done.

Code sample 4.43: C0419_TemplateClip

```
Image img = Image.getInstance(RESOURCE);
float w = img.getScaledWidth();
float h = img.getScaledHeight();
PdfTemplate t = writer.getDirectContent().createTemplate(850, 600);
t.addImage(img, w, 0, 0, h, 0, -600);
Image clipped = Image.getInstance(t);
clipped.scalePercent(50);
document.add(new Paragraph("Template clip:"));
document.add(clipped);
```

What happens with the image in the template is true for all the objects you add to the direct content. Everything that is added outside the boundaries of a `PdfTemplate` of a page will be present in the PDF, but you won't see it in a PDF viewer.

iText may change the way an image is compressed, but it doesn't remove pixels. In the case of code sample 4.40, the complete picture will be in the PDF file, but it won't be visible when looking at the PDF document.

If you need to clip an image using a shape that is different from a rectangle, you need to use a clipping path. This is shown in figure 4.39.

Figure 4.39: Clipping path

We don't need any new functionality to achieve this, we can use the `newPath()` method that was introduced in section 4.2.2. See code sample 4.44.

Code sample 4.44: C0419_TemplateClip

```
t = writer.getDirectContent().createTemplate(850, 600);
t.ellipse(0, 0, 850, 600);
t.clip();
t.newPath();
t.addImage(img, w, 0, 0, h, 0, -600);
```

FIgure 4.40 shows the result of the final example of this chapter. If you look closely, you see that the edges are a gradient similar to what we had when we discussed soft mask images. In this case however, we are using a soft mask dictionary.

Figure 4.40: Transparent overlay

The code to create this soft mask is a tad more complex.

Code sample 4.45: C0419_TemplateClip

```
Image img = Image.getInstance(RESOURCE);
float w = img.getScaledWidth();
float h = img.getScaledHeight();
canvas.ellipse(1, 1, 848, 598);
canvas.clip();
canvas.newPath();
canvas.addImage(img, w, 0, 0, h, 0, -600);
PdfTemplate t2 = writer.getDirectContent().createTemplate(850, 600);
PdfTransparencyGroup transGroup = new PdfTransparencyGroup();
```

```
10  transGroup.put(PdfName.CS, PdfName.DEVICEGRAY);
11  transGroup.setIsolated(true);
12  transGroup.setKnockout(false);
13  t2.setGroup(transGroup);
14  int gradationStep = 30;
15  float[] gradationRatioList = new float[gradationStep];
16  for(int i = 0; i < gradationStep; i++) {
17    gradationRatioList[i] = 1 - (float)Math.sin(Math.toRadians(90.0f/gradationStep*(i + 1)));
18  }
19  for(int i = 1; i < gradationStep + 1; i++) {
20      t2.setLineWidth(5 * (gradationStep + 1 - i));
21      t2.setGrayStroke(gradationRatioList[gradationStep - i]);
22      t2.ellipse(0, 0, 850, 600);
23      t2.stroke();
24  }
25  PdfDictionary maskDict = new PdfDictionary();
26  maskDict.put(PdfName.TYPE, PdfName.MASK );
27  maskDict.put(PdfName.S, new PdfName("Luminosity"));
28  maskDict.put(new PdfName("G"), t2.getIndirectReference());
29  PdfGState gState = new PdfGState();
30  gState.put(PdfName.SMASK, maskDict);
31  canvas.setGState(gState);
32  canvas.addTemplate(t2, 0, 0);
```

Let's take a closer look at what happens in code sample 4.45:

- In line 1 to 7, we create an image and we add it to the canvas after defining a clipping path. If we stopped here, we'd have the same result as in figure 4.39.
- In line 8 to 13, we create a Form XObject and we define a transparency group for that `PdfTemplate`.
- In line 14 to 14, we draw 30 identical ellipses with different border widths and different border colors—30 shades of gray.
- In line 25 to 28, we create a soft mask dictionary for the Form XObject with the ellipses.
- In line 29 to 31, we create a graphics state dictionary with an `/SMask` entry and we change the state.
- When we add the Form XObject with the ellipses in line 32, the `PdfTemplate` acts as a transparent overlay for the image.

With this example, we conclude chapter 4.

4.3 Summary

In this chapter, we've taken a closer look at the first part of the Adobe Imaging Model, more specifically at the syntax that allows you to construct and paint paths, to introduce colors, and to change, save and restore the graphics state. We've created external objects, Form XObjects as well as image XObjects, and when discussing the graphics state dictionary, we've focused on transparency and we've applied this to images.

In the next chapter, we'll focus on text state.

5. Text State

In section 4.1.3, we discovered that there are 5 types of graphics objects in PDF. We've already discussed path objects, external objects, inline image objects, and shading objects in chapter 4. We've saved text objects for this chapter.

5.1 Text objects

We started chapter 4 with the following snippet of PDF syntax:

```
BT
36 788 Td
/F1 12 Tf
(Hello World )Tj
ET
q
0 0 m
595 842 l
S
Q
```

The part between the `BT` and `ET` operators is a text object. Table 5.1 shows the corresponding iText methods.

Table 5.1: Text object operators

PDF	iText	Description
`BT`	`beginText()`	Begins a text object. Initializes the text matrix, text line matrix and identity matrix.
`ET`	`endText()`	Ends a text object, discards the text matrix.

There are specific rules for text objects. Inside a `BT`/`ET` sequence, it is allowed:

- to change color (using the operators listed in table 4.5),
- to use general graphics state operators (listed in table 4.8),
- to use text state, text positioning and text showing operators (as will be discussed in this chapter), and
- to use marked content operators (as will be discussed in the next chapter).

It is not allowed to use any other operator, e.g. you are not allowed to construct, stroke or fill paths inside a `BT`/`ET` sequence.

It is not allowed to nest text objects. When discussing the graphics state stack, we nested `saveState()`/`restoreState()` sequences. With text objects, a second `BT` is forbidden before an `ET`.

The color of text is determined by using the graphics state operators to change the fill and the stroke color. By default, glyphs will be drawn using the fill color. The default can be changed by using a text state operator that changes the rendering mode.

5.1.1 Text state operators

The text state is a subset of the graphics state. The available text state operators are listed in table 5.2.

Table 5.2: Text state operators

PDF	iText	Parameters	Description
`Tf`	`setFontAndSize`	`(font, size)`	Sets the text font (a `BaseFont` object) and size.
`Tc`	`setCharacterSpacing`	`(charSpace)`	Sets the character spacing (initially `0`).
`Tw`	`setWordSpacing`	`(wordSpace)`	Sets the word spacing (initially `0`).
`Tz`	`setHorizontalScaling`	`(scale)`	Sets the horizontal scaling (initially `100`).
`TL`	`setLeading`	`(leading)`	Sets the leading (initially `0`).
`Ts`	`setTextRise`	`(rise)`	Sets the text rise (initially `0`).
`Tr`	`setTextRenderingMode`	`(render)`	Specifies a rendering mode (a combination of stroking and filling). By default, glyphs are filled.

We can't take a look at any examples yet, because we don't know anything about operators to position and to show text yet.

5.1.2 Text-positioning operators

A glyph is a graphical shape and it's subject to all graphical manipulations, such as coordinate transformations defined by the CTM, but there are also three matrices for text that are valid inside a text object:

- *The text matrix*—This matrix is updated by the text-positioning and text-showing operators listed in tables 5.3 and 5.4.
- *The text-line matrix*—This captures the value of the text matrix at the beginning of a line of text.
- *The text-rendering matrix*—This is an intermediate result that combines the effects of text state parameters, the text matrix and the CTM.

Table 5.3 lists the available text-positioning operators.

Table 5.3: Text-positioning operators

PDF	iText	Parameters	Description
`Td`	`moveText`	`(tx, ty)`	Moves the text to the start of the next line, offset from the start of the current line by `(tx, ty)`.
`TD`	`moveTextWithLeading`	`(tx, ty)`	Same as `moveText()` but sets the leading to `-ty`.
`Tm`	`setTextMatrix`	`(a,b,c,d,e,f)` / `(e,f)`	Sets the text matrix and the text-line matrix. The parameters `a`, `b`, `c`, `d`, `e`, and `f` are the elements of a matrix that will replace the current text matrix.
`T*`	`newlineText`	`()`	Moves to the start of the next line (depending on the current value of the leading.

The value of the matrix parameters isn't persisted from one text object to another. Every new text object, starts with a new text, text-line and text-rendering matrix.

5.1.3 Text-showing operators

We conclude the overview of text-related operators with the text-showing operators. See table 5.4.

Table 5.4: Text-showing operators

PDF	iText	Parameters	Description
`Tj`	`showText`	`(string)`	Shows a text string.
`'`	`newlineShowText`	`(string)`	Moves to the next line, and shows a text string.
`"`	`newlineShowText`	`(aw, ac, string)`	Moves to the next line, and shows a text string using `aw` as word spacing and `ac` as character spacing.
`TJ`	`showText`	`(textarray)`	Shows one or more text strings, allowing individual glyph positioning.

Now that we've been introduced to all the available text operators, let's take a look at some examples.

5.1.4 Text operators in action

In the first example, we changed the text state a couple of times before adding the words "Hello World". This is shown in figure 5.1.

Figure 5.1: Text state operators

We already know from the first example in the previous chapter how the first "Hello World" was added. This is shown in code sample 5.1.

Code sample 5.1: C0501_TextState

```
canvas.beginText();
canvas.moveText(36, 788);
canvas.setFontAndSize(BaseFont.createFont(), 12);
canvas.showText("Hello World ");
canvas.endText();
```

In code sample 5.2, we try some more text state operators. With the `setCharacterSpacing()` method, we increase the space between the characters with 3 user units. With the `setWordSpacing()` method, we increase the space between the words with 30 user units. With the `setHorizontalScaling()` method, we scale the words to 150% of their original width. Finally, we add the word "Hello" followed by the word "World" with a text rise of 4 user units.

Code sample 5.2: C0501_TextState

```
canvas.beginText();
canvas.moveText(36, 760);
canvas.setCharacterSpacing(3);
canvas.showText("Hello World ");
canvas.setCharacterSpacing(0);
canvas.setLeading(16);
canvas.newlineText();
canvas.setWordSpacing(30);
canvas.showText("Hello World ");
canvas.setWordSpacing(0);
canvas.setHorizontalScaling(150);
canvas.newlineShowText("Hello World ");
```

```
canvas.setHorizontalScaling(100);
canvas.setLeading(24);
canvas.newlineShowText("Hello ");
canvas.setTextRise(4);
canvas.showText("World ");
```

Figure 5.2 demonstrates the different parameters we can use for the `setTextRenderingMode()` method.

Figure 5.2: Text rendering mode

Let's start with the first four lines, of which only three are visible. These are added to the document using the code from code sample 5.3.

Code sample 5.3: C0501_TextState

```
canvas.setColorFill(BaseColor.BLUE);
canvas.setLineWidth(0.3f);
canvas.setColorStroke(BaseColor.RED);
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_INVISIBLE);
canvas.newlineShowText("Hello World (invisible)");
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_STROKE);
canvas.newlineShowText("Hello World (stroke)");
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_FILL);
canvas.newlineShowText("HelloWorld (fill)");
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_FILL_STROKE);
canvas.newlineShowText("HelloWorld (fill and stroke)");
canvas.endText();
```

The first line that is drawn is invisible. You can only see what is written if you select the text and copy it into a text editor. This is a way to add text to a document that can be seen by a machine when parsing a

document, but not by a human being when reading the document in a PDF viewer. In the second line (the first visible line), the outlines of every glyph is drawn using the stroke color (red). The next line shows the default behavior. The fill color is blue and that's the color that is used to draw text. There's also a line where we fill and stroke the text. You see the outlines of the text in red and the glyphs are filled in blue.

Table 5.5 shows an overview of all the possible parameters. The first column shows the value of the operand for the `Tr` operator in PDF. The second column shows the value that is used in iText.

Table 5.5: Overview of the text rendering mode values

PDF	Rendering mode	Description
0	`TEXT_RENDER_MODE_FILL`	This is the default: glyphs are shapes that are filled.
1	`TEXT_RENDER_MODE_STROKE`	With this mode, the paths of the glyphs are stroked, not filled.
2	`TEXT_RENDER_MODE_FILL_STROKE`	Glyphs are filled first, then stroked.
3	`TEXT_RENDER_MODE_INVISIBLE`	Glyphs are neither filled nor stroked. Text added using this rendering mode is invisible, but it can be selected and copied.
4	`TEXT_RENDER_MODE_FILL_CLIP`	Fill text and add text to path for clipping.
5	`TEXT_RENDER_MODE_STROKE_CLIP`	Stroke text and add text to path for clipping.
6	`TEXT_RENDER_MODE_FILL_STROKE_CLIP`	Fill and stroke text and add text to path for clipping.
7	`TEXT_RENDER_MODE_CLIP`	Add text to path for clipping.

We've used the parameters ending with `_CLIP` for the final four lines in figure 4.2. In code sample 5.4, we show the text, and then we draw a green rectangle that should normally cover the upper half of the text. However, the text is used as a clipping path, which explains why we don't see any rectangle. We just see the text, even half of the text that is invisible.

Code sample 5.4: C0501_TextState

```
canvas.setColorFill(BaseColor.GREEN);
canvas.saveState();
canvas.beginText();
canvas.setTextMatrix(36, 624);
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_CLIP);
canvas.showText("Hello World (clip)");
canvas.endText();
canvas.rectangle(36, 628, 236, 634);
canvas.fill();
canvas.restoreState();
canvas.saveState();
canvas.beginText();
canvas.setTextMatrix(36, 608);
```

```
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_STROKE_CLIP);
canvas.showText("Hello World (stroke clip)");
canvas.endText();
canvas.rectangle(36, 612, 236, 618);
canvas.fill();
canvas.restoreState();
canvas.saveState();
canvas.beginText();
canvas.setTextMatrix(36, 592);
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_FILL_CLIP);
canvas.showText("HelloWorld (fill clip)");
canvas.endText();
canvas.rectangle(36, 596, 236, 602);
canvas.fill();
canvas.restoreState();
canvas.saveState();
canvas.beginText();
canvas.setTextMatrix(36, 576);
canvas.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_FILL_STROKE_CLIP);
canvas.showText("HelloWorld (fill and stroke clip)");
canvas.endText();
canvas.rectangle(36, 580, 236, 586);
canvas.fill();
canvas.restoreState();
```

Figure 5.3 shows us some text positioning examples.

Figure 5.3: Text positioning

In code sample 5.5, we move the text to a specific coordinate using the `moveText()` method. We show the text "Hello World" twice. These words are added on the same line. Then we move down 16 user units using the `moveTextWithLeading()` method. As we're using the relative Y-value -16, the leading is set to 16 user units.

We add the text that is shown on the second line. Using the `newlineText()` method will once more move the current position down with 16 user units. We add a third line.

We can change the text matrix to a new absolute value using the `setTextMatrix()` method. The text we add is shown on the fourth line. We change the text matrix once more, introducing more values. Due to the new text matrix, the text we add in the last `showText()` line is scaled with a factor 2 and slightly skewed.

Code sample 5.5: C0502_TextState

```
canvas.beginText();
canvas.moveText(36, 788);
canvas.setFontAndSize(BaseFont.createFont(), 12);
canvas.showText("Hello World ");
canvas.showText("Hello World ");
canvas.moveTextWithLeading(0, -16);
canvas.showText("Hello World ");
canvas.newlineText();
canvas.showText("Hello World ");
canvas.setTextMatrix(72, 740);
canvas.showText("Hello World ");
canvas.setTextMatrix(2, 0, 1, 2, 36, 710);
canvas.showText("Hello World ");
canvas.endText();
```

In figure 5.4, we try some text showing operators.

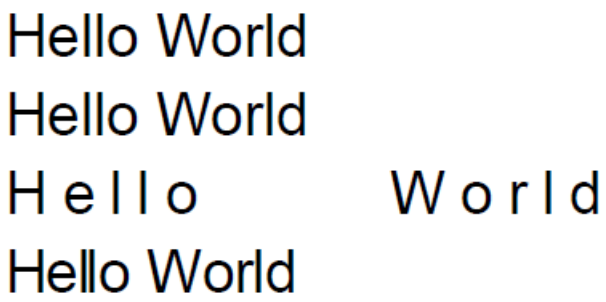


Figure 5.4: Text showing

We've already used the `showText()` and the `newlineText()` methods in previous examples. We now also use the `newlineShowText()` method that changes the text state. We use it once to show Hello Text using the current state regarding word and character spacing, and we use it once to introduce a word spacing of 30 units and a character spacing of 3 units. Finally, we take a look at the `showText()` method that accepts a `PdfTextArray` object. We can use this method to fine-tune the distance between different parts of a line. In this case, we move "el" 45 glyph units closer to "H", we move the two "l" glyphs 85 glyph units closer to each other. We don't use a space character to separate the two words "Hello" and "World". Instead, we introduce a gap of 250 units in glyph space. Finally, we move "ld" 35 glyph units closer to "Wor". Using a text array is common in high-end PDF tools that require a high typography-quality.

Code sample 5.5: C0502_TextState

```
canvas.beginText();
canvas.moveText(216, 788);
canvas.showText("Hello World ");
canvas.setLeading(16);
canvas.newlineShowText("Hello World ");
canvas.newlineShowText(30, 3, "Hello World ");
canvas.setCharacterSpacing(0);
canvas.setWordSpacing(0);
canvas.newlineText();
PdfTextArray array = new PdfTextArray("H");
array.add(45);
array.add("el");
array.add(85);
array.add("lo");
array.add(-250);
array.add("Wor");
array.add(35);
array.add("ld ");
canvas.showText(array);
canvas.endText();
```

All the examples we've seen so far are marked with an all-caps warning: "THIS IS NOT THE USUAL WAY TO ADD TEXT; THIS IS THE HARD WAY!!!"

iText provides much easier ways to add text to a document, but that's outside the scope of this book. We'll only take a look at a couple of the convenience methods that can be used when adding text at absolute positions.

5.1.5 Convenience methods

The result we got when we tried to move glyphs closer to each other using a self-made `PdfTextArray` wasn't that successful. It's not something you're supposed to do manually. The `PdfContentByte` class has a static `getKernArray()` method that allows you to create the `PdfTextArray` automatically based on the *kerning* info that is available in the font program. Let's take a close look at figure 5.5.

Figure 5.5: Text with and without kerning

The first line is added the same way, we've added text before, using `canvas.showText("Hello World ");` For the second and third line, we introduced kerning. You may not see the difference with the naked eye, but when we calculate the widths without and with kerning, we get a difference of 0.66 point.

Code sample 5.6 shows two different ways to use kerning.

Code sample 5.6: C0503_TextState

```
BaseFont bf = BaseFont.createFont();
canvas.setFontAndSize(bf, 12);
canvas.setLeading(16);
canvas.showText("Hello World ");
canvas.newlineText();
PdfTextArray array = PdfContentByte.getKernArray("Hello World ", bf);
canvas.showText(array);
canvas.newlineText();
canvas.showTextKerned("Hello World ");
canvas.newlineText();
canvas.showText(String.format("Kerned: %s; not kerned: %s",
        canvas.getEffectiveStringWidth("Hello World ", true),
        canvas.getEffectiveStringWidth("Hello World ", false)));
```

Instead of using the `showText()` method passing a `PdfTextArray` created with the `getKernArray()` method, we can also use the `showTextKerned()` method. The result is identical. The `showTextKerned()` method uses the `getKernedArray()` internally. If you'd use RUPS to look inside the PDF, you'd find the following syntax:

```
[(Hello ), 40, (W), 30, (or), -15, (ld )] TJ
```

The word "Hello" isn't optimized, but we save 40 units in glyph space between the space character and the "W". The word "World" is split into three pieces, saving 30 units between "W" and "or", but introducing an extra 15 units between "or" and "ld".

With the `getEffectiveStringWidth()` method, we can get the effective width of a `String` using the current font and font size that is active in the `PdfContentByte` object. The Boolean parameter indicates whether you want the width using kerning (`true`) or the width without taking into account the kerning (`false`). Looking at figure 5.5, we see that the kerned string measures 64.68 user units whereas the string without kerning measures 65.340004 user units.

How does glyph space relate to user space?

We already know that one user unit corresponds with one point by default. These measurements are done in *user space*. Glyphs are measured in *glyph space*. Thousand units in glyph space correspond with one unit in *text space*. The conversion from *text space* to user space is done through the *text matrix*.

An example will help us understand. When we kerned the words "Hello World ", we gained 55 units in glyph space (40 + 30 - 15). This is 0.055 units in text space. We used a font size of 12 points, hence we've gained 12 x 0.055 or 0.66 points. If we ignore the rounding errors, that's the difference between the effective width of the non-kerned and the kerned text string: 65.34 - 64.68.

Figure 5.5 also shows a couple of "Hello World" text snippets that are rotated. This could be achieved by calculating a text matrix, but in this case, we used the `showTextAligned()` and the `showTextAlignedKerned()` methods as shown in code sample 5.7.

Code sample 5.6: C0503_TextState

```
1 canvas.showTextAligned(Element.ALIGN_CENTER, "Hello World ", 144, 790, 30);
2 canvas.showTextAlignedKerned(Element.ALIGN_CENTER, "Hello World ", 144, 770, 30);
```

Using these methods is much easier than having to define a text matrix. In this case, we define a coordinate, for instance `(114, 790)` and an angle in degrees, for instance `30`. We tell iText to align the text in such a way that the coordinate is at the center of the baseline of the text.

Possible values are:

- `Element.ALIGN_LEFT`— aligns the text, so that the coordinate is to the left of the text,
- `Element.ALIGN_CENTER`— aligns the text, so that the coordinate is at the center of the text, and
- `Element.ALIGN_RIGHT`— aligns the text, so that the coordinate is to the right of the text.

This is still a pretty low level approach, we'll discuss more convenient ways to add text in the book "Create your PDFs with iText[1]".

We have listed all the possible text state operators and we've made some simple examples demonstrating the difference between the different iText methods involving text state, but we've overlooked one important aspect. So far, we've always used `BaseFont.createFont()` to create a `BaseFont` object. This `createFont()` method introduces the default font, which is the Standard Type 1 font Helvetica. In the next section, we'll discover how we can introduce other font types.

5.2 Introducing fonts

The very first versions of Adobe Reader, at that time known as Acrobat Reader, shipped with 14 so-called *Base 14 fonts*. The rationale behind these fonts was that you never had to embed these fonts into a PDF file as you could always expect them to be present in the viewer.

Today, these fonts are no longer part of the viewer. The terminology has also changed. We now call these fonts the *Standard Type 1 fonts*: (1) Courier, (2) Courier-Bold, (3) Courier-Oblique, (4) Courier-BoldOblique, (5) Helvetica, (6) Helvetica-Bold, (7) Helvetica-Oblique, (8) Helvetica-BoldOblique, (9) Times-Roman, (10) Times-Bold, (11) Times-Italic, (12) Times-BoldItalic, (13) Symbol, and (14) ZapfDingbats.

Each viewer is supposed to have access to the 14 Standard Type 1 fonts on the OS, or to a font that is very similar. For instance: on Windows, Helvetica will be substituted by Arial.

These fonts are useful if you want to keep the file size of the PDF document small, but in general it is recommended to embed (subsets of) fonts. As a matter of fact, embedding fonts can be mandatory in some use cases, for instance when you're creating PDF/A documents (A stands for Archiving).

To embed a font, we need a font program.

[1] https://leanpub.com/itext_pdfcreate

5.2.1 Font programs

Table 5.6 lists the extensions of the files that contain font metrics or a font program, or both.

Table 5.6: Font files and their extensions

Font Type	Extension	Description
Type 1	.afm, .pfm, .pfb	A Type 1 font is composed of two files: one containing the metrics (.afm or .pfm) and one containing the mathematical descriptions for each character (.pfb).
TrueType	.ttf	A font based on a specification developed by Apple to compete with Adobe's type 1 fonts
OpenType	.otf, .ttf, .ttc	A cross-platform font file format based on Unicode. OpenType font files containing Type 1 outlines have an .otf extension. Filenames of OpenType fonts containing TrueType data have a .ttf or .ttc extension. The .ttc extension is used for TrueType Collections.

Type 1 was originally a proprietary specification owned by Adobe, but after Apple introduced TrueType as a competitor, the specification was published, and third party manufacturers were allowed to create Type 1 fonts, provided they adhered to the specification. In 1991, Microsoft started using TrueType as its standard font and for a long time, TrueType was the most common font on both Mac OS and MS Windows systems. Unfortunately, Apple as well as Microsoft added their own proprietary extensions, and soon they had their own versions and interpretations of (what once was) the standard. When looking at a commercial font, you had to be careful to buy a font that could be used on your system. A TrueType font for Windows didn't necessarily work on a Mac, and vice versa. To resolve the platform dependency of TrueType fonts, Microsoft started developing a new format. Microsoft was joined by Adobe, and support for Adobe's Type 1 fonts was added. In 1996, a new font format was born: OpenType fonts. The glyphs in an OpenType font can be defined using either TrueType or Type 1 technology.

This is the history of fonts in a nutshell. There's nothing to worry about: fonts inside a PDF, no matter of which type, can be viewed on any platform. Let's examine fonts from a PDF perspective.

5.2.2 Fonts inside a PDF

Fonts are stored in a dictionary of type `/Font` and the `/Subtype` entry indicates how the font is stored inside the PDF. Table 5.7 shows the different options for the `/Subtype` value.

Table 5.7: Subtype values for fonts

Subtype	Description
`/Type1`	A font that defines glyph shapes using Type 1 font technology
`/Type3`	A font that defines glyphs with streams of PDF graphics operators
`/TrueType`	A font based on the TrueType font format
`/Type0`	A composite font—a font composed of glyphs from a descendant CIDFont

Table 5.7: Subtype values for fonts

Subtype	Description
/CIDTypeType0	A CIDFont whose glyph descriptions are based on the Compact Font Format (CFF)
/CIDTypeType2	A CIDFont whose glyph descriptions are based on TrueType font technology

Table 5.7 in this book corresponds to Table 110 in ISO-32000-1, omitting the subtype /MMType1. Multiple Master fonts have been discontinued.

Multiple Master (MMType1) fonts can be present in a PDF document, and iText can deal with PDFs containing MMType1 fonts, but there's no support for MMType1 in the context of creating documents.

Fonts in PDF are a complex matter. Instead of diving into the theory of fonts, we'll take a look at some examples to see how section 5.2.1 and section 5.2.2 relate to each other.

5.3 Using fonts in PDF

Let's start by making a distinction between two groups of fonts. If the font dictionary has a /Subtype entry with value /Type1, /Type3 and /TrueType, the font is stored inside the PDF as a *simple* font. This means that each glyph corresponds with a single-byte character. A Type 0 font is called a *composite* font. It obtains its glyphs from a font-like object called a CIDFont, but let's start with simple fonts.

5.3.1 Simple fonts

Content that needs to be rendered using a simple font is stored in the content stream as a sequence of single byte characters. In a simple font, we can define 256 glyphs. These glyphs are represented by characters with values ranging from 0 to 255.

The mapping between the characters and the glyphs is called the *character encoding*. A Type 1 font can have a special built-in encoding, as is the case for Symbol and ZapfDingbats. With other fonts, multiple encodings may be available. For instance, the glyph known as *dagger* (†) corresponds with (char) 134 in the encoding known as WinAnsi, aka Western European Latin (code page 1252), a superset of Latin 1 (ISO-8859-1). The same dagger glyph corresponds to different character values in the Adobe Standard encoding (178), MacRoman encoding (160), and PDF Doc Encoding (129).

Figure 5.6 shows a PDF with five lines of text. If we look at the Fonts tab in the Document Properties dialog, we see a list of five fonts.

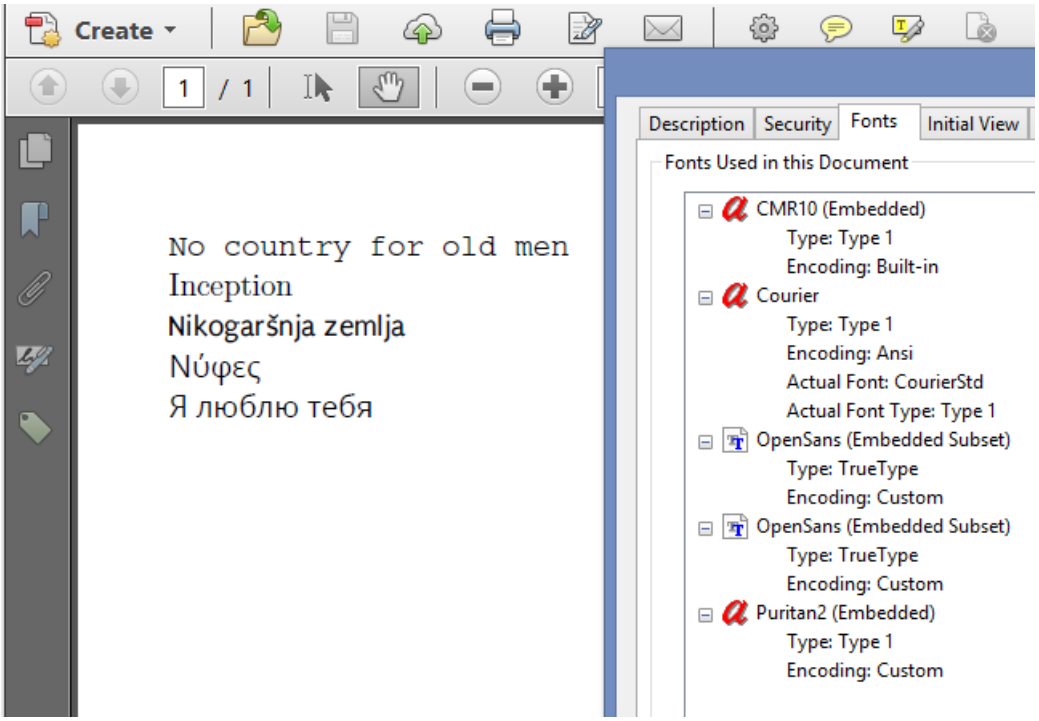


Figure 5.6: Simple fonts

Let's examine the content on this screen shot line by line:

- The first line ("No Country for old men") is written in Courier, using the Windows code page (ANSI encoding). In our code, we defined the standard type 1 font Courier, but we didn't embed the font into the PDF. Instead of the Courier font we expected, Acrobat used CourierStd, a Type1 font that is very similar (if not identical) to Courier.
- The second line ("Inception") is written using the Type 1 font Computer Modern Regular (CMR10). This font was embedded into the PDF and has a single built-in encoding.
- The third line (a text in a Central European language) is written using an OpenType font with Type 1 outlines called Puritan. We used Code Page 1250 which is the encoding used for Central European and Eastern European languages that use Latin script, but that involve some special characters that can't be found in Latin-1. This custom set of glyphs is fully embedded into the PDF.
- The fourth line (a text in Greek) is written using an OpenType font with TrueType outlines called OpenSans. We used Code Page 1253 used to write Modern Greek. Only a subset of this font is embedded, containing only those characters that are used in the text.
- The fifth line (a text in Russian) is also written using OpenSans, but now we used Code Page 1251 that covers languages that use the Cyrillic alphabet. OpenSans is mentioned twice in the fonts tab because there are two sets of OpenSans in the PDF using a different custom encoding.

Sample 5.7 shows the code that was used to create this PDF.

Code sample 5.7: C0504_SimpleFonts

```
String TYPE1 = "resources/fonts/cmr10.afm";
String OT_T1 = "resources/fonts/Puritan2.otf";
String OT_TT = "resources/fonts/OpenSans-Regular.ttf";
canvas.beginText();
canvas.moveText(36, 806);
canvas.setLeading(16);
BaseFont bf;
bf = BaseFont.createFont(BaseFont.COURIER, BaseFont.WINANSI, BaseFont.NOT_EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("No country for old men");
bf = BaseFont.createFont(TYPE1, BaseFont.WINANSI, BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("Inception");
bf = BaseFont.createFont(OT_T1, BaseFont.CP1250, BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("Nikogar\u0161nja zemlja");
bf = BaseFont.createFont(OT_TT, "CP1253", BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("\u039d\u03cd\u03c6\u03b5\u03c2");
bf = BaseFont.createFont(OT_TT, "CP1251", BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("\u042f \u043b\u044e\u0431\u043b\u044e \u0442\u0435\u0431\u044f");
canvas.endText();
```

Now let's look inside the PDF.

5.3.1.1 The font is not embedded

The font courier is defined like this:

```
1 0 obj
<</BaseFont/Courier/Type/Font/Encoding/WinAnsiEncoding/Subtype/Type1>>
endobj
```

This is a simple PDF dictionary with four entries:

1. The `/Type` is `/Font`,
2. The `/Subtype` is `/Type1`,
3. The `/BaseFont` is `Courier`, and
4. The `/Encoding` is `/WinAnsiEncoding`.

When we look at the content stream of the page, we see:

```
/F1 12 Tf
(No country for old men) '
```

We recognize the name /F1 in the /Resources of the page dictionary:

```
/Resources<</Font<</F1 1 0 R/F2 2 0 R/F3 3 0 R/F4 4 0 R/F5 5 0 R>>>>
```

The page resources consist of a /Font entry, which in turn has five entries, one for each font that is used on the page. The name /F1 refers to object 1 in the document. This is pretty straightforward, but as already mentioned, it is often better to embed the font, because your operating system won't always be able to find a font that resembles the font you selected.

5.3.1.2 The font is embedded

Figure 5.7 shows how the font Computer Modern Regular (CMR10) is stored inside our sample PDF. We recognize the /Type, /Subtype and /BaseFont entry. There is no /Encoding entry, but there's a /FirstChar, /LastChar, /Widths and /FontDescriptor entry.

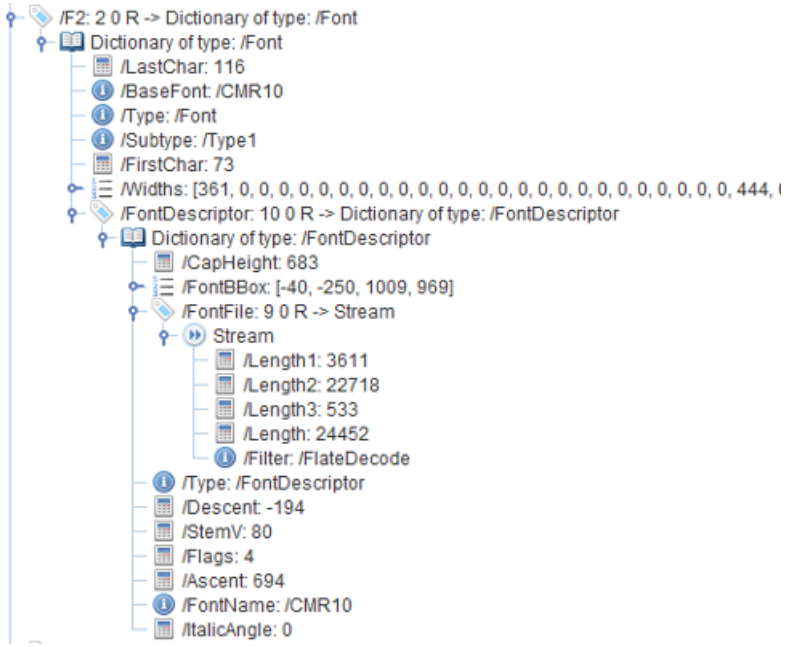


Figure 5.7: Type 1 font with built-in encoding

We are using this font to write the word "Inception". We only need 8 different glyphs to write this word, the first glyph being 'I' (corresponding with character 73) and the last one being 't' (corresponding with character 116). In the /Widths array, we define the width of each glyph, starting with 'I' and ending with 't'. We are not interested in the widths of the glyphs we don't use, hence we can define these widths as 0.

The value of the /FontDescriptor entry is another dictionary. It contains more info about the font metrics. It also contains a /FontFile entry. The value of this entry is a stream containing the font program.

A font descriptor of an embedded font has a /FontFile entry, a /FontFile2 entry, or a /FontFile3 entry. Table 5.8 explains the difference.

Table 5.8: Possible font file entries for the font descriptor

Entry key	Description
`/FontFile`	The stream refers to a Type 1 font program in the original (non-compact) format. As usual, the `/Length` parameter shows the length of the compressed stream. There are 3 extra length values that give us information about the decoded stream: `/Length1` shows the length in bytes of the clear-text portion of the Type 1 font program. `/Length2` shows the length in bytes of the encrypted portion of the Type 1 font program. `/Length3` shows the length in bytes of the fixed-content portion.
`/FontFile2`	The stream refers to a TrueType font program. Again, the value of the `/Length` parameter corresponds with the length of the compressed stream, but there's an extra `/Length2` entry that shows the length of the decoded TrueType font program.
`/FontFile3`	The stream refers to a font program represented in the Compact Font Format. The `/Subtype` entry further specifies the type of font. Possible values for `/Subtype` are: - `/Type1C`: the font file is a Type 1 compact font. - `/CIDFontType0C`: the font file is a Type 0 compact CIDFont. - `/OpenType`: the font is an OpenType font.

The way the font is stored in a PDF by iText depends on the type of font program that was provided, but also on the encoding that was used. For instance, we have used the font program `Puritan2.otf`, an OpenType font with Type 1 outlines, but we stored it inside the PDF as a Type 1 font. This is shown in figure 5.8.

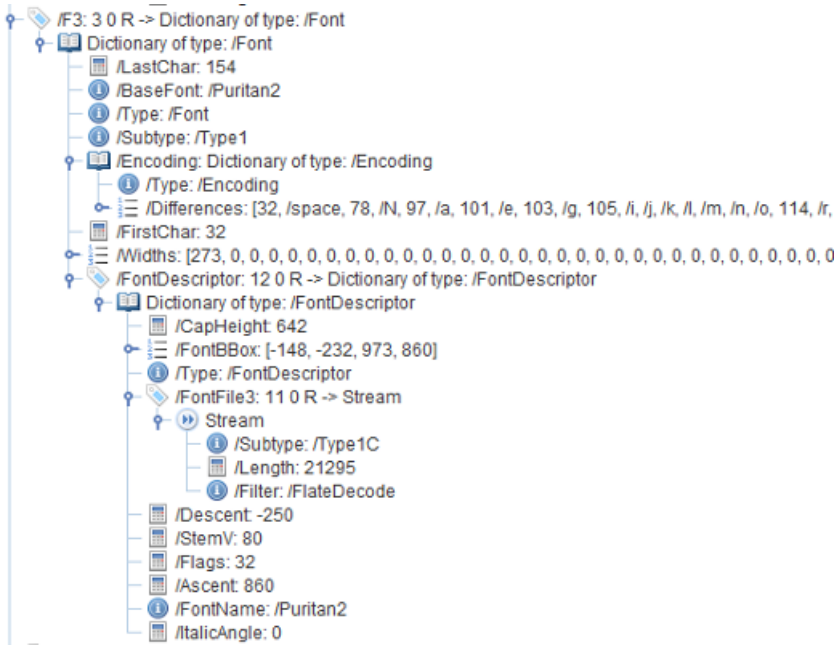


Figure 5.8: Type 1 font

In the `/FontDescriptor` entry, we recognize a `/FontFile3` stream with `/Subtype` equal to `/Type1C`. The font is stored as a Type 1 Compact font. In the next section, we'll use the same font program without a different

encoding. We'll discover that the font will be embedded in a totally different way.

5.3.1.3 Encoding

If we look at the font dictionary, we also see an entry for the `/Encoding`. This entry is a dictionary that contains a `/Differences` array. This array describes the differences from the encoding specified by `/BaseEncoding` or, if `/BaseEncoding` is absent from a default base encoding. The numbers in this array are the first index in a sequence of character codes to be changed. In figure 5.8, we start with the index 23 for the character code `/space`. The index 105 is the first index for the sequence of character codes `/i`, `/j`, `/k`,... These differences reflect the glyphs we use in code page 1250.

We see a similar *custom* encoding in figure 5.9 (code page 1253) and figure 5.10 (code page 1251). In both cases the font OpenSans-Regular is used.

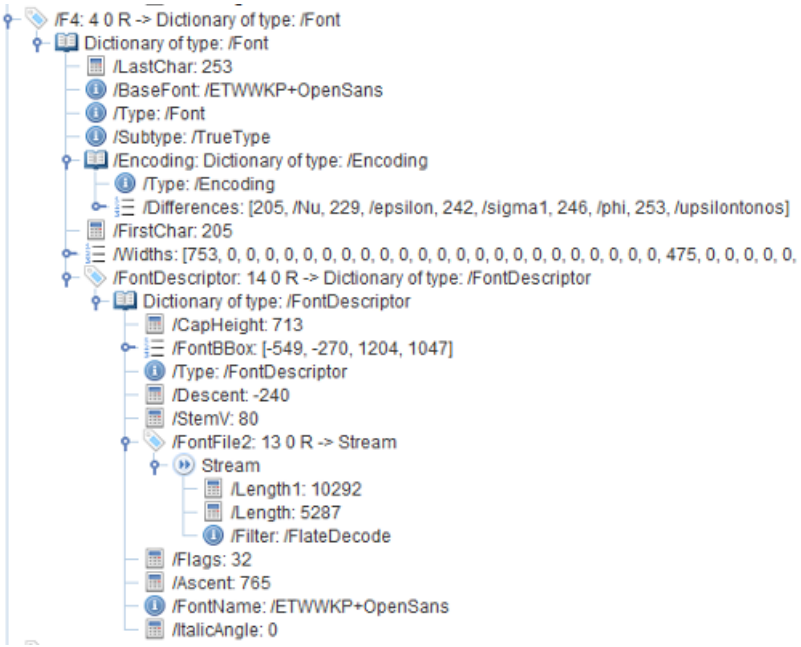


Figure 5.9: OpenSans subset for Greek characters

This is the font that is used for the greek text. The content stream for the snippet of Greek text looks like this:

```
/F4 12 Tf
(Íýöåò) '
```

The Í character corresponds with Unicode 205. In the custom encoding shown in figure 5.9, it is used for the greek capital Nu. The ò character corresponds with Unicode 242. In the custom encoding shown in figure 5.9, it is used for a glyph representing the greek letter sigma. And so on.

Figure 5.10 shows how the same font, stored using a different encoding.

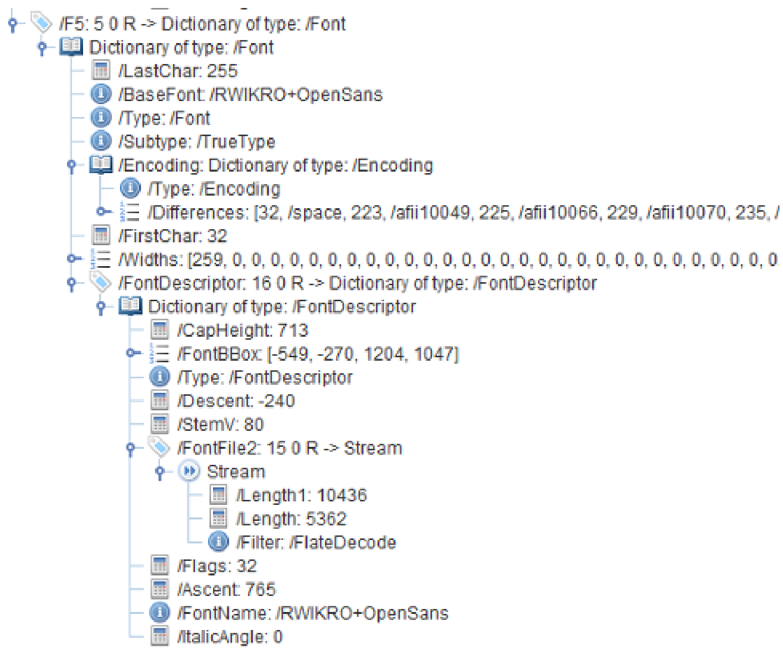


Figure 5.10: OpenSans subset for Russian characters

The content stream for the snippet of Russian text looks like this:

```
/F5 12 Tf
(ß ëþáëþ òåáÿ) '
```

Whereas the ò in the Greek example corresponded with sigma, it now corresponds with `afii10084`. AFII stands for the Association for Font Information Interchange, and AFII has defined an id for a large set of characters from different languages. The AFII notation is different from Unicode in the sense that AFII was designed for textual entities, whereas Unicode was designed for graphic entities. The AFII notation has been replaced with Unicode in many cases, but you may still find references to it in PDF.

5.3.1.4 Font subsets

When we look at the font descriptor, we see a `/FontFile2` entry: the font is embedded as a TrueType font. There is something odd about the `/FontName` entry in the font descriptor dictionary. In figure 5.9, we see `/ETWWKP+OpenSans`. In figure 5.10, we see `/RWIKRO+OpenSans`. The actual font name is OpenSans, but we are using two different subsets of the font. The distinction between the subsets is made by prefixing the name with a tag followed by a plus sign. The tag consists of six upper case letters that can be chosen randomly, but that need to be unique for each different subset within the PDF file. When creating a PDF using iText, the subset will only contain glyph descriptions for the characters that are used in the document.

> If we compare the original length of font files, we see that the OpenSans fonts take about 10K bytes. The Type 1 fonts were more than double in byte-size. This is caused by the fact that Type 1 fonts can't be sub-setted. We only need a handful of glyphs and we only define the widths and the encoding for the glyphs we use, but we can't store a reduced version of the font program.

5.3.1.5 Available encodings

Once you start experimenting with code sample 5.7, for instance by trying to render the Cyrillic characters using the font Courier, you'll notice that the Russian `String` isn't rendered. That's because Codepage 1251 isn't supported in Courier. The Standard Type1 font Courier doesn't know anything about Cyrillic characters. In code sample 5.8, we ask iText which encodings are supported in Courier, Computer Modern, Puritan and OpenSans.

Code sample 5.8: C0505_SupportedEncoding

```
public static final String TYPE1 = "resources/fonts/cmr10.afm";
public static final String OT_T1 = "resources/fonts/Puritan2.otf";
public static final String OT_TT = "resources/fonts/OpenSans-Regular.ttf";
public static void main(String[] args) throws DocumentException, IOException {
    C0505_SupportedEncoding app = new C0505_SupportedEncoding();
    app.listEncodings(BaseFont.createFont(
        BaseFont.COURIER, BaseFont.WINANSI, BaseFont.NOT_EMBEDDED));
    app.listEncodings(BaseFont.createFont(TYPE1, BaseFont.WINANSI, BaseFont.NOT_EMBEDDED));
    app.listEncodings(BaseFont.createFont(OT_T1, BaseFont.WINANSI, BaseFont.NOT_EMBEDDED));
    app.listEncodings(BaseFont.createFont(OT_TT, BaseFont.WINANSI, BaseFont.NOT_EMBEDDED));
}
public void listEncodings(BaseFont bf) {
    System.out.println(bf.getPostscriptFontName());
    String[] encoding = bf.getCodePagesSupported();
    for (String enc : encoding) {
        System.out.print('\t');
        System.out.println(enc);
    }
}
```

When we run this small example and look at the `System.out`, we see the following overview:

```
Courier
CMR10
Puritan2
    1252 Latin 1
    Macintosh Character Set (US Roman)
    Symbol Character Set
    865 MS-DOS Nordic
    863 MS-DOS Canadian French
    861 MS-DOS Icelandic
    860 MS-DOS Portuguese
OpenSans
    1252 Latin 1
    1250 Latin 2: Eastern Europe
```

```
1251 Cyrillic
1253 Greek
1254 Turkish
1257 Windows Baltic
1258 Vietnamese
Macintosh Character Set (US Roman)
```

Only Puritan and OpenSans offer the possibility to use different encodings to create a simple font.

Those are also the fonts that allow the use of `Identity-H` and `Identity-V`. When you see these `Identity` encodings, you are looking at text that uses Unicode. In that case, you are dealing with a composite font.

5.3.2 Composite fonts

A composite font obtains its glyphs from a font-like object called a CIDFont. A composite font is represented by a font dictionary with subtype `/Type0`. The Type 0 font is know as the *root font*, and its associated CIDFont is called its *descendant*.

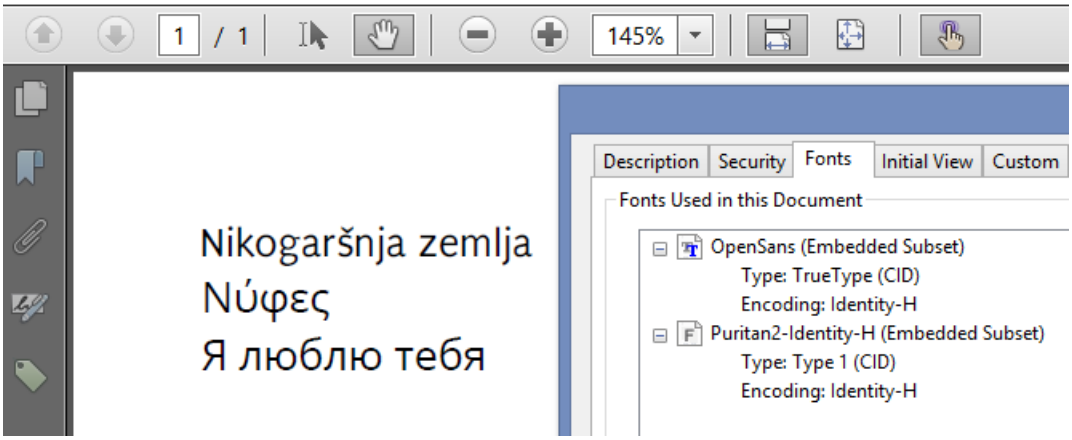


Figure 5.11: Composite fonts seen from the outside

In figure 5.11, we use two of the fonts we already used in figure 5.6, but instead of introducing them as a simple font, we now use them as a composite font. The encoding is no longer *custom*, but *Identity-H*. We don't reuse the standard Type 1 font Courier, nor the Computer Modern font as they can't be used as composite fonts.

If you compare code sample 5.9 with code sample 5.7, you'll notice only one major difference: we now use `BaseFont.IDENTITY_H` instead of a custom encoding.

Code sample 5.9: C0506_CompositeFonts

```
String OT_T1 = "resources/fonts/Puritan2.otf";
String OT_TT = "resources/fonts/OpenSans-Regular.ttf";
canvas.beginText();
canvas.moveText(36, 806);
canvas.setLeading(16);
BaseFont bf;
bf = BaseFont.createFont(OT_T1, BaseFont.IDENTITY_H, BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("Nikogar\u0161nja zemlja");
bf = BaseFont.createFont(OT_TT, BaseFont.IDENTITY_H, BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("\u039d\u03cd\u03c6\u03b5\u03c2");
bf = BaseFont.createFont(OT_TT, BaseFont.IDENTITY_H, BaseFont.EMBEDDED);
canvas.setFontAndSize(bf, 12);
canvas.newlineShowText("\u042f \u043b\u044e\u0431\u043b\u044e \u0442\u0435\u0431\u044f");
canvas.endText();
```

Let's take a look inside the PDF document. Figure 5.12 shows a snippet of the content stream.

```
BT
36 806 Td
16 TL
/F1 12 Tf
( / J L P H B S Å O K B  [ F N M K B)'
/F2 12 Tf
( k š “ ,  )'
/F2 12 Tf
( É  Õ è Ë Õ è  Ü Ï Ë é)'
ET
```

Figure 5.12: Composite fonts seen from the inside

Every glyph is now represented by two characters, which is different from what we saw in section 5.3.1.3.

We can now compare figure 5.13 with figure 5.8, and figure 5.14 with figure 5.9.

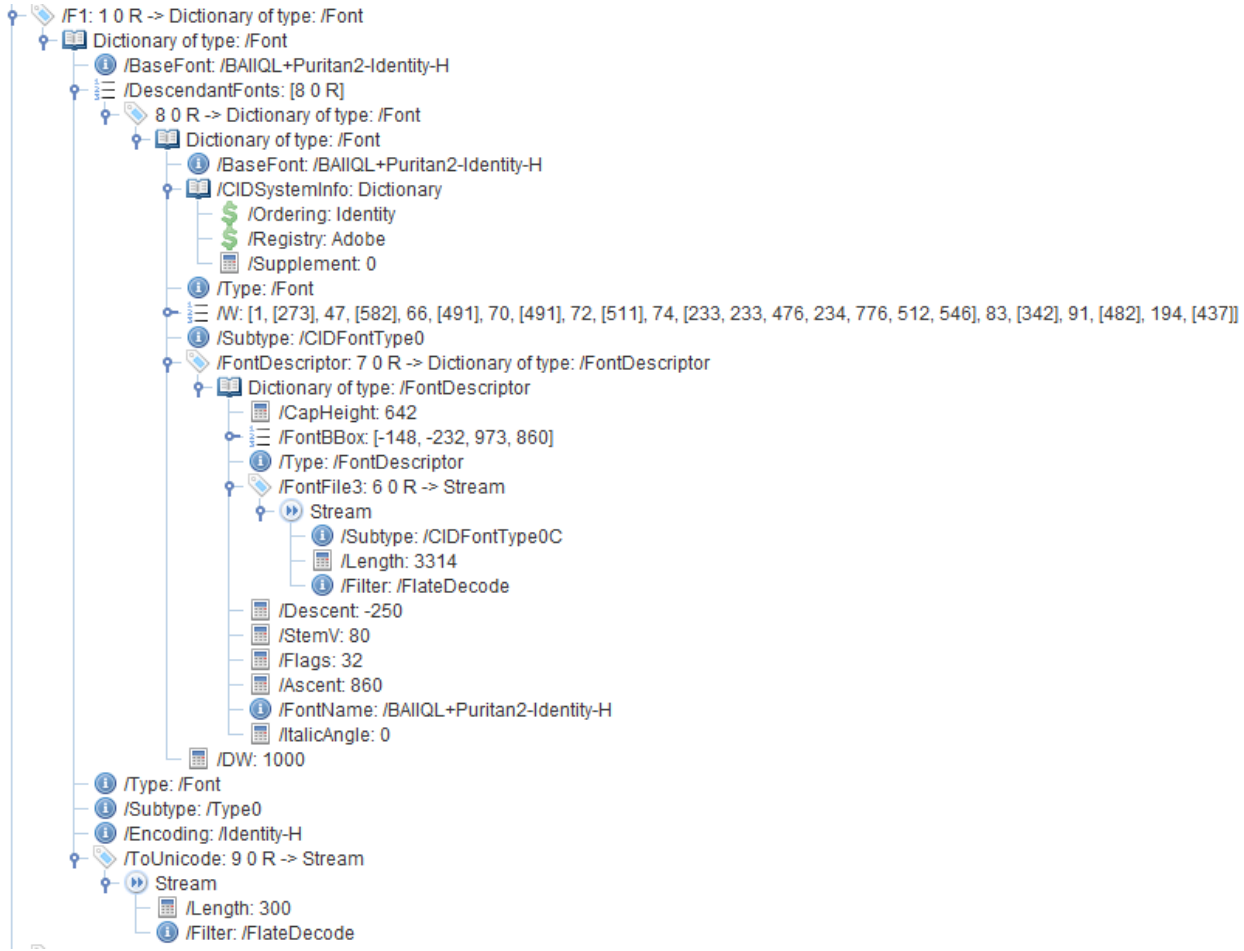


Figure 5.13: Puritan as a composite font

For the Puritan font, we have a dictionary of type `/Font` if which the `/Subtype` is `/Type0` and the `/Encoding` is `/Identity-H`. The `/ToUnicode` entry is very important: it maps every character code that is used in this font to its corresponding Unicode value. This `/ToUnicode` stream is called a *CMap*. Such a CMap is similar to the `/Encoding` entry we encountered when we discussed simple fonts. It maps character codes to character selectors. These character selectors are the CIDs (Character Identifiers) of a CIDFont.

The `DescendantFonts` entry is an array containing references to the descendant fonts that define the Type0 font. In PostScript, this array can contain multiple fonts. In PDF, this array can only contain one value: a single CIDFont. In this case, we have a CIDFont of which the `/Subtype` is `/CIDTypeType0`. In the font descriptor, we see a `/FontFile3` (Compact Font Format) entry of which the `/Subtype` is `/CIDFontType0C`.

The OpenSans font is no longer used as a simple font with `/SubType /TrueType`, but as a `/Type0` font with a descendant `CIDFontType2`. The font descriptor has a `/FontFile2` (TrueType font program). See figure 5.14.

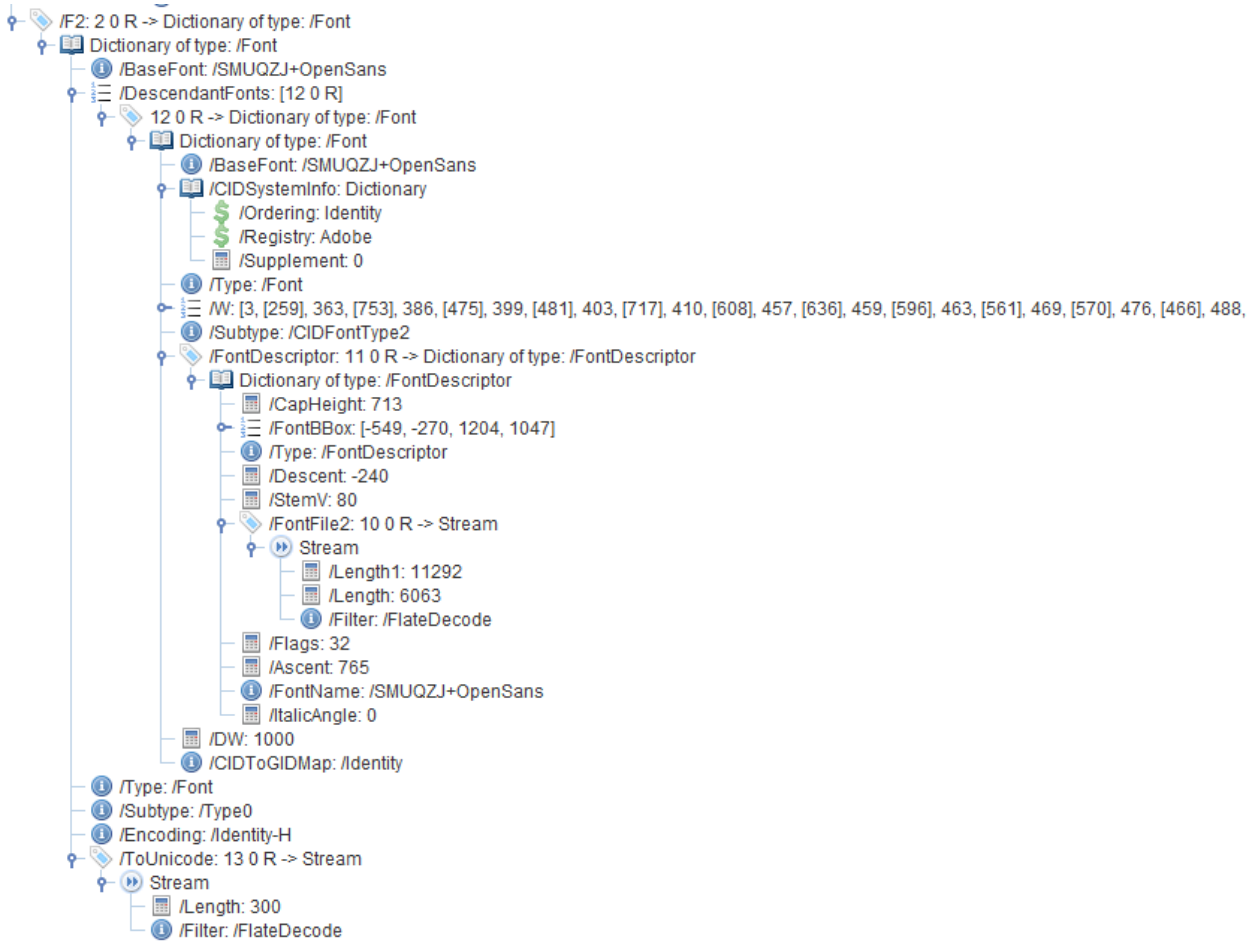


Figure 5.14: OpenSans as a composite font

5.4 Using fonts in iText

Looking back at the examples in the previous section, you see something magical going on. We take a single font file, e.g. `OpenSans-Regular.ttf` and by using different parameters for the `createFont()` method, iText gives us a `BaseFont` object that results in a completely different type of font when we look under the hood. If you look at the `BaseFont` class, you'll notice that it is defined as an abstract class.

Let's take a look at the different `BaseFont` implementations that are available in iText. This will also allow us to discuss Type3 fonts and fonts with CMaps in a context that is different from the `/ToUnicode` entry.

5.4.1 Overview of the BaseFont implementations

Table 5.9 lists a series of iText classes that are used when creating a `BaseFont` object. Together these classes cover all the font types listed in table 5.6, as well as all the font subtypes listed in table 5.7.

Table 5.9: iText BaseFont classes

Class name	Description
`Type1Font`	You'll get a `Type1Font` instance if you create a standard type 1 font, or if you pass an `.afm` or `.pfm` file. Standard Type 1 fonts are *never* embedded. For other Type 1 fonts, it depends on the value of the `embedded` parameter and the presence of a `.pfb` file whether or not the font will be embedded by iText.
`TrueTypeFont`	In spite of its name, this class isn't only used for TrueType fonts (`.ttf`), but also for OpenType fonts with TrueType (`.ttf`) or Type1 (`.otf`) outlines. This class will create a font of subtype `/TrueType` or `/Type1` in a PDF document.
`TrueTypeFontUnicode`	Files with extension `.ttf` or `.otf` can also result in this subclass of `TrueTypeFont` if you use them to create a composite font. So will files with extension `.ttc`. Inside the PDF, you'll find the subtype `/Type0` along with `/CIDFontType2` (`.ttf` and `.ttc` files) or `/CIDFontType0` (`.otf` files). Contrary to its superclass, `TrueTypeFontUnicode` ignores the `embedded` parameter. iText will *always* embed a *subset* of the font.
`CFFFont`	OpenType fonts with extension `.otf` use the Compact Font Format (CFF). `CFFFont` is *not* a subclass of `BaseFont`. Creating a font using an `.otf` file results in an instance of `TrueTypeFont`, but it's the `CFFFont` class that does the work.
`Type3Font`	Type3 fonts are special. They don't come in files, but you need to create them using PDF syntax. Type3 fonts are *always* embedded.
`CJKFont`	This is a special class for Chinese, Japanese, and Korean fonts for which the metrics files are shipped in a separate JAR. Using a CJK font results in a Type 0 font; the font is never embedded.

You don't need to address classes such as `Type1Font` or `TrueTypeFont` directly; just as you used the `Image` class to make iText select the correct image type, you can let `BaseFont` decide which font class applies, except for one very special type of font: Type3.

5.4.2 A Type3 font example

When we created a new iText logo in 2014, we decided to use the brand name as the basis for the graphics.

Figure 5.15: iText logo

This logo was created by a graphical designer, but we thought it would be nice if we could use this logo in documents using a font. As we only need four glyphs: I, T, E, and X, and as two of these glyphs (I and E) need to be rendered in orange, whereas the other two (X and T) need to be rendered in blue, it makes sense to introduce a Type3 font consisting of nothing but these four glyphs. Type3 fonts are user-defined fonts of which the glyphs are drawn using PDF syntax. They can also contain color information.

Code sample 5.10 shows how the font is created.

Code sample 5.10: C0507_Type3Font

```
1  Type3Font t3 = new Type3Font(writer, true);
2  PdfContentByte i = t3.defineGlyph('I', 700, 0, 0, 1200, 600);
3  i.setColorStroke(new BaseColor(0xf9, 0x9d, 0x25));
4  i.setLineWidth(linewidth);
5  i.setLineCap(PdfContentByte.LINE_CAP_ROUND);
6  i.moveTo(600, 36);
7  i.lineTo(600, 564);
8  i.stroke();
9  PdfContentByte t = t3.defineGlyph('T', 1170, 0, 0, 1200, 600);
10 t.setColorStroke(new BaseColor(0x08, 0x49, 0x75));
11 t.setLineWidth(linewidth);
12 t.setLineCap(PdfContentByte.LINE_CAP_ROUND);
13 t.moveTo(144, 564);
14 t.lineTo(1056, 564);
15 t.moveTo(600, 36);
16 t.lineTo(600, 564);
17 t.stroke();
18 PdfContentByte e = t3.defineGlyph('E', 1150, 0, 0, 1200, 600);
19 e.setColorStroke(new BaseColor(0xf8, 0x9b, 0x22));
20 e.setLineWidth(linewidth);
21 e.setLineCap(PdfContentByte.LINE_CAP_ROUND);
22 e.moveTo(144, 36);
23 e.lineTo(1056, 36);
24 e.moveTo(144, 300);
25 e.lineTo(1056, 300);
26 e.moveTo(144, 564);
27 e.lineTo(1056, 564);
28 e.stroke();
29 PdfContentByte x = t3.defineGlyph('X', 1160, 0, 0, 1200, 600);
30 x.setColorStroke(new BaseColor(0x10, 0x46, 0x75));
31 x.setLineWidth(linewidth);
32 x.setLineCap(PdfContentByte.LINE_CAP_ROUND);
33 x.moveTo(144, 36);
34 x.lineTo(1056, 564);
35 x.moveTo(144, 564);
36 x.lineTo(1056, 36);
37 x.stroke();
```

In line 1, we create a `BaseFont` instance. This is the only type of font for which we use a specific constructor instead of using the `createFont()` method. We pass an instance of `PdfWriter` to which the `Type3Font` will write the description of each glyph. The `Boolean` parameter indicates whether or not we want to define the color at the level of the glyph.

In this case, we pass `true`, which means that we want want to create colored glyphs. If we pass `false`, we are not allowed to use color for the glyphs; instead we'll define the color by changing the overall fill (and stroke) color as explained in section 5.1.4.

Once we have a `Type3Font` instance, we can start defining glyphs using the `defineGlyph()` method. This method returns a `Type3Glyph` instance. This class extends the `PdfContentByte` class, which means that we can draw the glyph using the methods explained in chapter 4.

The `defineGlyph()` method expects the following parameters:

- `c`: the character to match this glyph.
- `wx`: the width of the glyph in glyph space.
- `llx`: the X coordinate of the lower-left corner of the glyph's bounding box.
- `lly`: the Y coordinate of the lower-left corner of the glyph's bounding box.
- `urx`: the X coordinate of the upper-right corner of the glyph's bounding box.
- `ury`: the Y coordinate of the upper-right corner of the glyph's bounding box.

In line 2 to 8 of code sample 5.10, we define the glyph that corresponds with the `'I'` character. In line 9 to 17, we define the `'T'` character. In line 18 to 28. we define the `'E'`. Finally, we define the `'X'` character in line 29 to 37.

Figure 5.16 shows what the font looks like when seen from the inside of the PDF document.

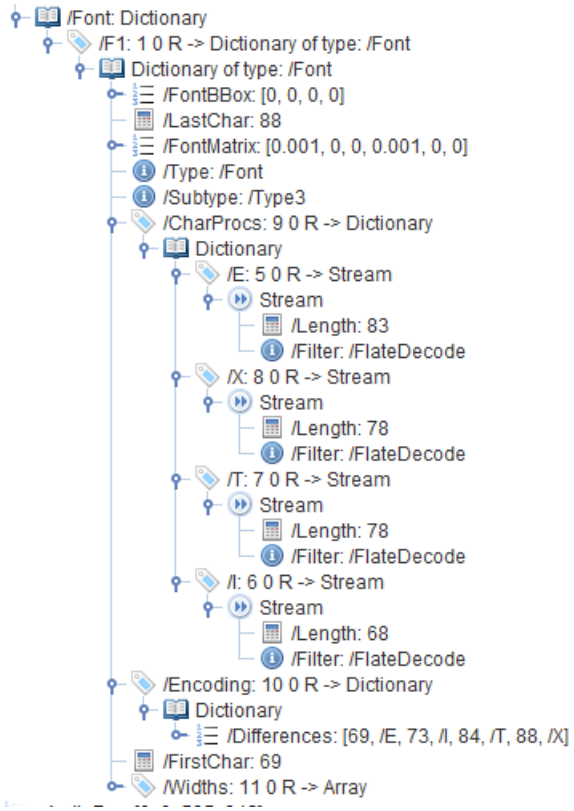


Figure 5.16: Type3 font

We have a font of subtype /Type3 defining four characters in the character value range from 69 ('E') to 88 ('X'). The /Encoding array maps four values in this range to four names /E, /I, /T, and /X. These four names correspond with keys in the /CharProcs dictionary. The value of each key is a stream that defines the glyph.

For instance, the /I key corresponds with the following content stream:

```
700 0 d0
0.97647 0.61569 0.1451 RG
125 w
1 J
600 36 m
600 564 l
S
```

The d0 operator sets width information and declares that the glyph description specifies both its shape and color. Alternatively, the d1 operator is used when you only define the shape, not the color.

In this case, we set the color using the RG operator, the width of the strokes using the w operator, and we use the J operator to define round caps. The actual glyph consists of a stroked line (S) between the coordinate defined by the m operator and the coordinate defined by the l operator. This is different from a "normal" font, where we define the outlines of the glyphs and then fill these outlines using a fill color.

The /T key corresponds with the following stream:

```
1170 0 d0
0.03137 0.28627 0.45882 RG
125 w
1 J
144 564 m
1056 564 l
600 36 m
600 564 l
S
```

In short: each line in the content stream of a glyph description will correspond with a line in your code. In this case, the previous snippet corresponds with lines 9 to 17 in code sample 5.10.

Code sample 5.11 shows how to use a BaseFont in iText.

Code sample 5.11: C0507_Type3Font

```
Font font = new Font(t3, 20);
Paragraph p = new Paragraph("ITEXT", font);
document.add(p);
p = new Paragraph(20, "I\nT\nE\nX\nT", font);
document.add(p);
```

We create a `Font` object, passing the `BaseFont` instance and a font size. Then we create `Paragraph` objects that use this font, and we add these objects to the `Document`. For instance: we add the strings `"ITEXT"` and `"I\nT\nE\nX\nT"`. Figure 5.17 shows the result.

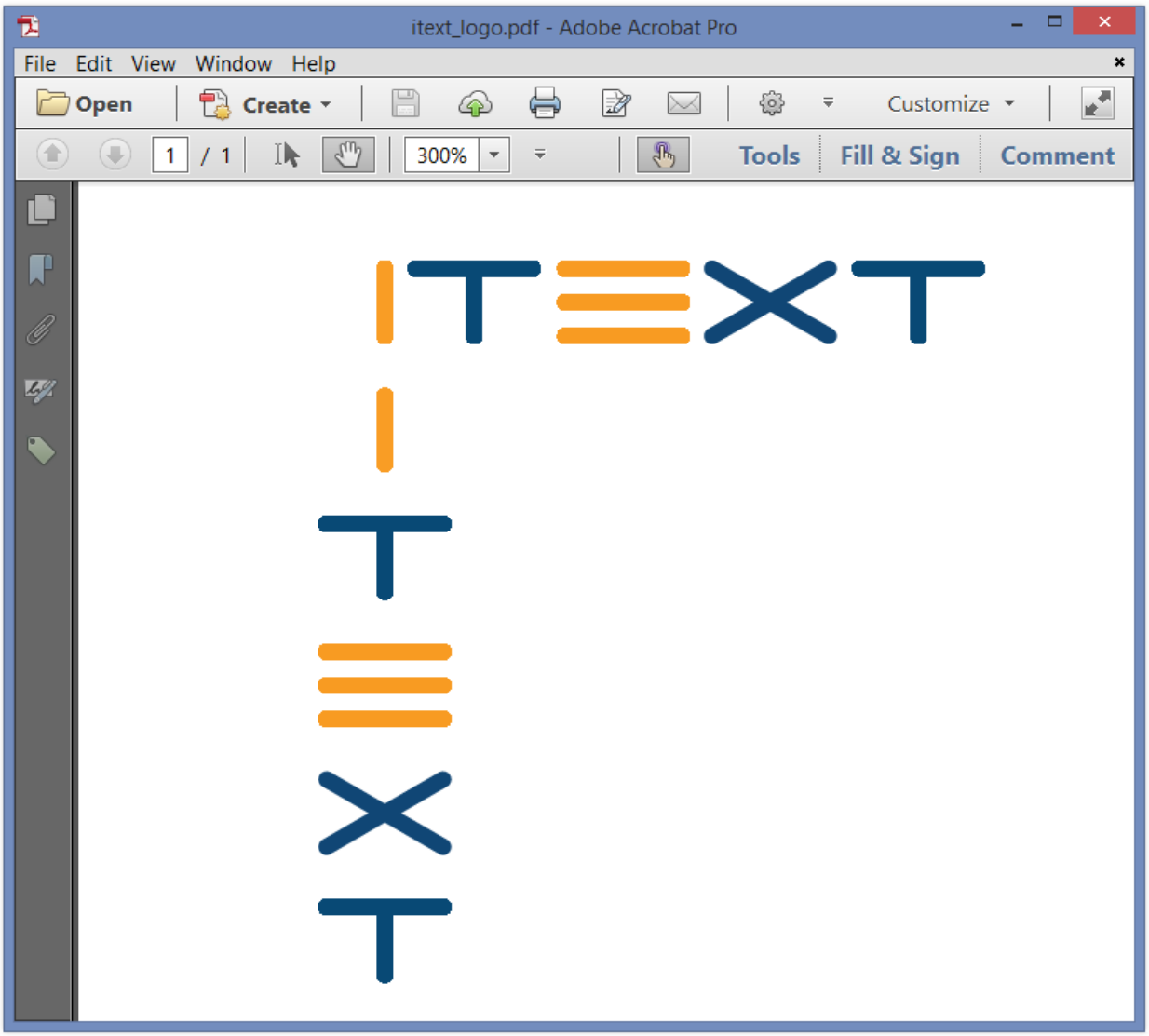


Figure 5.17: iText logo

Type3 fonts are always tricky, in the sense that they often produce odd results when trying to extract text from a PDF. In this case, we chose the characters that correspond with each glyph in such a way that we can easily recognize the actual text in the content stream:

```
BT
36 806 Td 0 -30 Td
/F1 20 Tf
(ITEXT) Tj
0 0 Td 0 -20 Td
(I) Tj 0 0 Td 0 -20 Td (T) Tj 0 0 Td 0 -20 Td (E)
Tj 0 0 Td 0 -20 Td (X) Tj 0 0 Td 0 -20 Td (T) Tj 0 0 Td
ET
```

This isn't always the case. We could easily have used the character `'a'` for the I glyph, `'b'` for the T glyph, `'c'` for the E glyph, and `'d'` for the X glyph. When you would extract the text from the PDF, you would then get `"abcdb"` instead of `"ITEXT"`. This is a common complaint from people who want to extract text from PDFs that use Type3 fonts, or when extracting text from a document with simple fonts with fonts that use a custom encoding or a wrong `/ToUnicode` table. In that case, you shouldn't blame the tool that extracts the content, but the tool that created it.

Let's conclude this chapter with an example that requires a `CJKFont`.

5.4.3 A CJKFont example

In figure 5.18, we list three movies showing their original title in Chinese, Japanese and Korean.

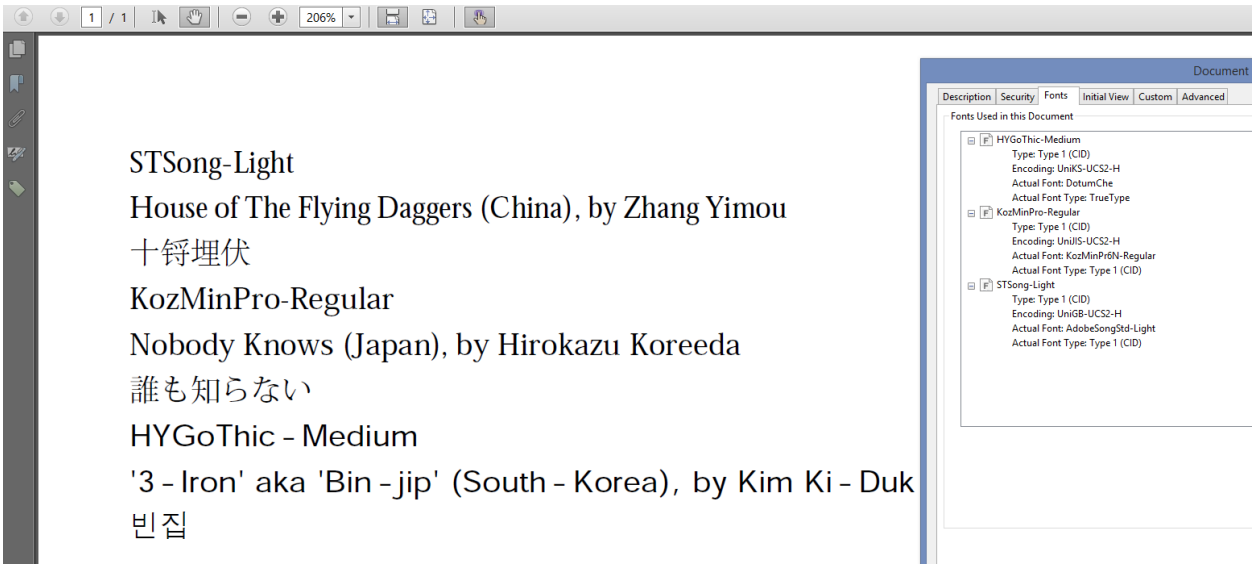


Figure 5.18: Chinese, Japanese and Korean fonts

We could have used an embedded font such as MS Arial Unicode to show these titles, but in this case, we used the so-called CJK fonts that don't need to be embedded.

If you open a file using these CJK fonts in Adobe Reader, and if the fonts aren't available, a dialog box will open. You'll be asked if you want to update the Reader. If you agree, the necessary font packs will be downloaded and installed.

To make this work, we don't need font programs that contain the drawing instructions for the glyphs, but we do need information about the font's properties and the encoding. This information can be found in files that are shipped in a separate jar: `itext-asian.jar`. You need to add this jar to your CLASSPATH if you want to try the code shown in listing 5.12.

Code sample 5.12: C0507_Type3Font

```
1  BaseFont bf = BaseFont.createFont("STSong-Light", "UniGB-UCS2-H", BaseFont.NOT_EMBEDDED);
2  Font font = new Font(bf, 12);
3  document.add(new Paragraph(bf.getPostscriptFontName(), font));
4  document.add(new Paragraph("House of The Flying Daggers (China), by Zhang Yimou", font));
5  document.add(new Paragraph("\u5341\u950a\u57cb\u4f0f", font));
6  bf = BaseFont.createFont("KozMinPro-Regular", "UniJIS-UCS2-H", BaseFont.NOT_EMBEDDED);
7  font = new Font(bf, 12);
8  document.add(new Paragraph(bf.getPostscriptFontName(), font));
9  document.add(new Paragraph("Nobody Knows (Japan), by Hirokazu Koreeda", font));
10 document.add(new Paragraph("\u8ab0\u3082\u77e5\u3089\u306a\u3044", font));
11 bf = BaseFont.createFont("HYGoThic-Medium", "UniKS-UCS2-H", BaseFont.NOT_EMBEDDED);
12 font = new Font(bf, 12);
13 document.add(new Paragraph(bf.getPostscriptFontName(), font));
14 document.add(new Paragraph("'3-Iron' aka 'Bin-jip' (South-Korea), by Kim Ki-Duk", font));
15 document.add(new Paragraph("\ube48\uc9d1", font));
```

In line 6 of this code snippet, iText will look in the `itext-asian.jar` for the `.properties` file that corresponds with the `fontname` we used. More specifically, iText will look for the file `KozMinPro-Regular.properties`. In this file, iText will find information about the font, for instance metrics such as the *ascent* and the *descent* of the glyphs.

iText will also search for the file used as the value for the `encoding`. The `UniJIS-UCS2-H` file contains a CMap that contains the Unicode (UCS-2) encoding for the Adobe-Japan1 character collection. We don't need to embed this CMap in the PDF, the way we did with the `/ToUnicode` CMap, because this is a predefined CMap. All PDF processors should support the predefined CMaps listed in the ISO standard for PDF.

Observe that the CMap files come in pairs: one for horizontal writing systems (ending in `-H`) and one for vertical writing systems (ending in `-V`).

Let's finish this chapter by looking what the `KozMinPro=Regular` font with encoding `UniJIS-UCS2-H` looks like when seen from this inside. This is shown in figure 5.18.

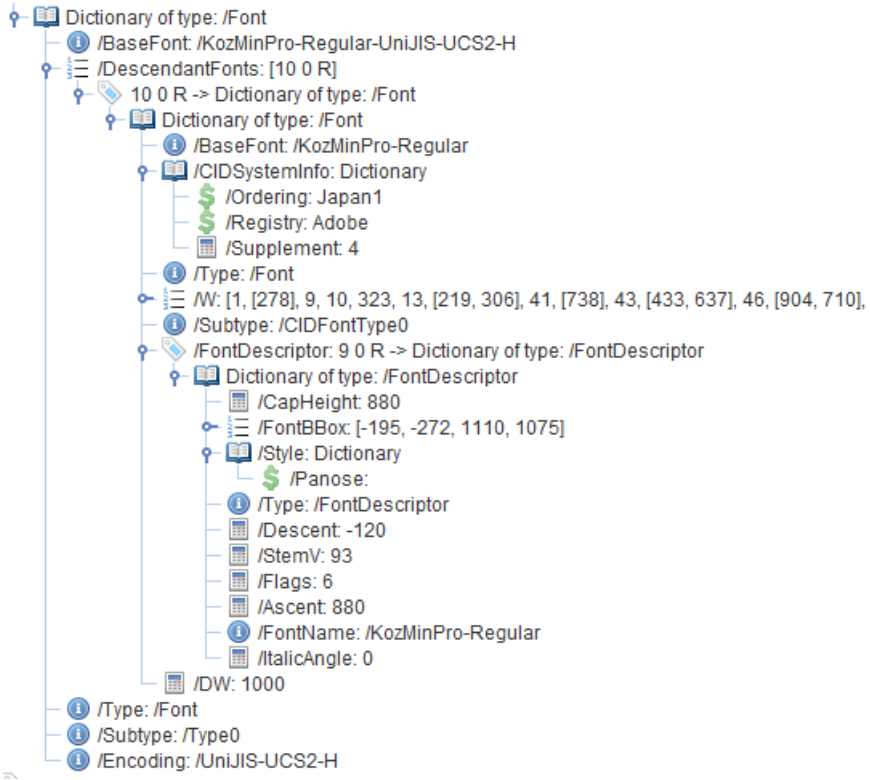


Figure 5.18: Japanese font

We see a `/Type0` font with a `/CIDFontType0` as descendant font. There is no font file, meaning that the font isn't embedded, but iText has taken some of the information from the files in `itext-asian.jar` for entries such as `/Descent`, `/Ascent`, `/W`, and so on. Without this information, it's not possible to create a valid CJK font.

5.5 Summary

This chapter about the Text State was an extension of the chapter about the Graphics State. We started by introducing a new series of PDF operators that can be used to change the text state, to position text and to show text.

We can not talk about text without talking about fonts, so we looked at the different flavors of font files, we looked at the way a font is stored inside a PDF, and we looked at how iText deals with fonts.

In the next chapter, we'll see a third series of PDF operators. Unlike the operators we discussed in the chapter about graphics state and the chapter about text state, these operators are not about drawing content on a page. Instead they are about adding attributes or specific characteristics to content that is visible or invisible on a page. We call them Marked Content operators.

6. Marked Content

III Part 3: Annotations and form fields

7. Annotations

8. Interactive forms