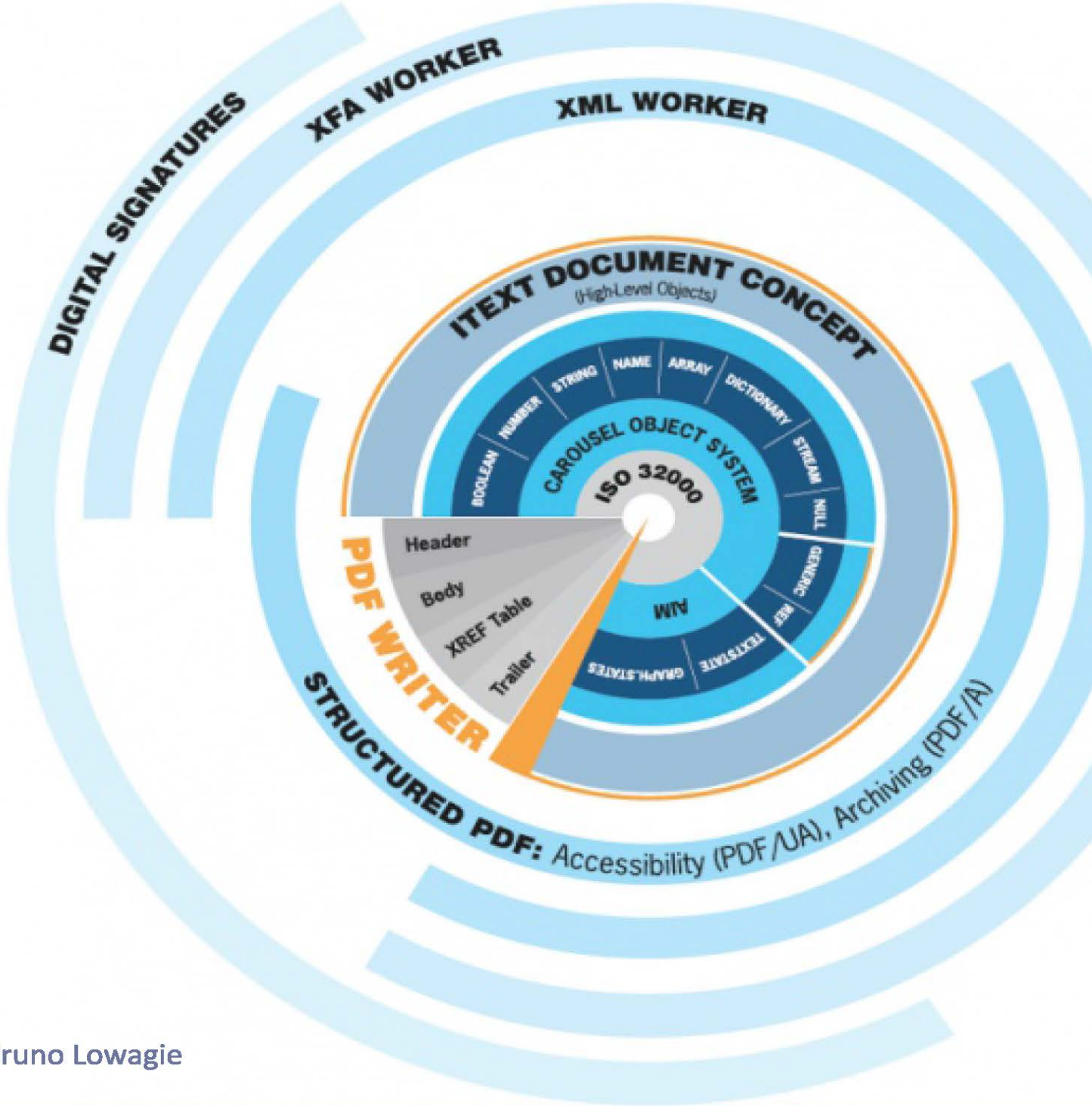
The ABC of PDF with iText

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 2014-05-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

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 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.

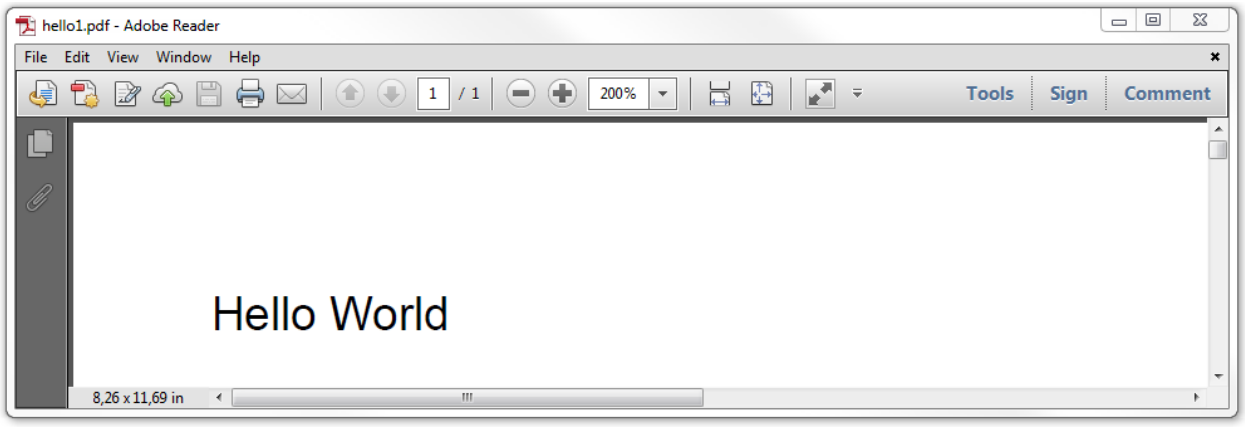


Figure 2.1: Hello World

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` keyword in 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.