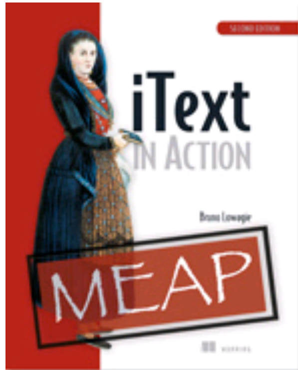
MANNING PUBLICATIONS

The Phrase Object: A List of Chunks with Leading

Article based on



iText in Action, Second Edition
EARLY ACCESS EDITION

Bruno Lowagie
MEAP Release: November 2009
Softbound print: Fall 2010 | 600 pages
ISBN: 9781935182610

This article is taken from the book iText in Action, Second Edition*. The author discusses the Phrase object. In particular, he shows how to create constants for different fonts and set the leading. He also explains how to substitute and embed fonts.*

Get **35% off** any version of *iText in Action, 2nd Ed.*, with the checkout code **fcc35**. Offer is only valid through www.manning.com.

When I created iText, I chose the word *chunk* for the atomic text element because of its first definition in my dictionary, "a solid piece."

A `Chunk` isn't aware of the space that is needed between two lines. That's why we set the leading. The word leading is pronounced as *ledding*. It's derived from the word lead (the metal). When type was set by hand for printing presses, strips of lead were placed between lines of type to add space. The word originally referred to the thickness of these strips of lead that were placed between the lines. The PDF Reference redefined the word as "the vertical distance between the baselines of adjacent lines of text" (IS0-32000; section 9.3.5).

A phrase, on the other hand, is defined as "a string of words." It isn't solid; it's a composed object. Translated to iText and Java, a `Phrase` is an `ArrayList` of `Chunk` objects.

A phrase with different fonts

Listing 1 shows how to create methods that compose `Phrase` objects using different `Chunk`s. Usually, you'll create constants for the different `Font`s you'll use.

Listing 1 DirectorPhrases1.java

```
public static final Font BOLD_UNDERLINED =                    A
  new Font(Font.TIMES_ROMAN, 12, Font.BOLD | Font.UNDERLINE); A
public static final Font NORMAL =                             A
```

```
    new Font(Font.TIMES_ROMAN, 12);                                        A

  public Phrase createDirectorPhrase(ResultSet rs)
    throws UnsupportedEncodingException, SQLException {
    Phrase director = new Phrase();                                        B
    director.add(new Chunk(                                                C
      new String(rs.getBytes("name"), "UTF-8"), BOLD_UNDERLINED));         C
    director.add(new Chunk(",", BOLD_UNDERLINED));                         C
    director.add(new Chunk(" ", NORMAL));                                  C
    director.add(                                                          C
      new Chunk(new String(rs.getBytes("given_name"), "UTF-8"), NORMAL));  C
    return director;
  }
```

A Creates different `Font` objects
B Creates a `Phrase` object
C Adds `Chunk`s to the `Phrase`

In this example, we're going to list 80 directors from a movie database. The method `createDirectorPhrase()` produces the `Phrase` exactly the way we want it. It's good practice to create a factory class containing different `createObject()` methods if you need to create `Chunk`, `Phrase`, or other objects in a standardized way.

The leading of a phrase

The method `createDirectorPhrase()` is used in listing 2. In this listing, we're showing five steps in the PDF creation process.

Listing 2 DirectorPhrases1.java

```
  Document document = new Document();                                     1
  PdfWriter.getInstance(document, new FileOutputStream(filename));         2
  document.open();                                                         3
  DatabaseConnection connection = new HsqldbConnection("filmfestival");
  Statement stm = connection.createStatement();
  ResultSet rs = stm.executeQuery("SELECT name, given_name"
    + "FROM film_director ORDER BY name, given_name");
  while (rs.next()) {                                                      4
    document.add(createDirectorPhrase(rs));                                4
    document.add(Chunk.NEWLINE);                                           4
  }                                                                        4
  stm.close();
  connection.close();
  document.close();                                                        5
```

1 Step 1: Create the `Document`
2 Step 2: Get an instance of `PdfWriter`
3 Step 3: Open the `Document`
4 Step 4: Add content
5 Step 5: Close the `Document`

Observe that we use the default leading in step #2.

FAQ:

What is the default leading in iText? If you don't define a leading, iText looks at the font size of the `Phrase` or `Paragraph` that is added to the document, and multiplies it by 1.5. For instance: if you have a `Phrase` with a font of size 10, the default leading is 15. For the default font, with default size 12, the default leading is 18.

In the next example, we'll change the leading with the method `setLeading()`.

Database encoding versus the default charset used by the JVM

In listing 1, some `String`s were created using the encoding UTF-8 explicitly:

```
  new String(rs.getBytes("given_name"), "UTF-8")
```

That's because the database contains different names with special characters. If you look at the HSQL script `filmfestival.script`, you'll find `INSERT` statements like this:

```
INSERT INTO FILM_DIRECTOR VALUES(
  41,'I\u00c3\u00b1\u00c3\u00a1rritu','Alejandro Gonz\u00c3\u00a1lez')
```

That's the record for the director Alejandro González Iñárritu. The characters á, `(char) 226`, and ñ, `(char) 241`, can be stored as one byte using the ANSI character encoding, which is a superset of ISO 8859-1 a.k.a. Latin-1. HSQL stores them in UNICODE using multiple bytes per character. To make sure that the `String` is created correctly, I've used `ResultSet.getBytes()` instead of `ResultSet.getString()`.

This isn't always necessary. In most database systems, you can define the encoding for each table or for the whole database. The JVM uses the platform's default charset, for instance, in the constructor `new String(byte[] bytes)`.

FAQ:

Why is the data I retrieve from my database rendered as gibberish? This can be caused by an encoding mismatch. The records in your database are encoded using encoding X, but the `String` objects obtained from your `ResultSet` assume that they are encoded using your platform's charset Y. For instance: the name González could be rendered as GonzÃ¡lez if the UNICODE characters are interpreted as ANSI characters.

These encoding problems disappear as soon as you've created the PDF document correctly. One of the main reasons why people prefer PDF over any other document format is because PDF is a *portable* document format. A PDF document can be viewed and printed on any platform—UNIX, Macintosh, Windows, Linux, or Palm OS—regardless of the encoding of the character set that is used.

In theory, a PDF document should look the same on any of these platforms, using any viewer available on that platform. But, there's a caveat! If you have a close look at figure 1, you can see that this isn't always true.

Font substitution for non-embedded fonts

In figure 1, you could see that Helvetica was replaced by ArialMT. Figure 1 shows that the choice of the replacement font is completely up to the document viewer.

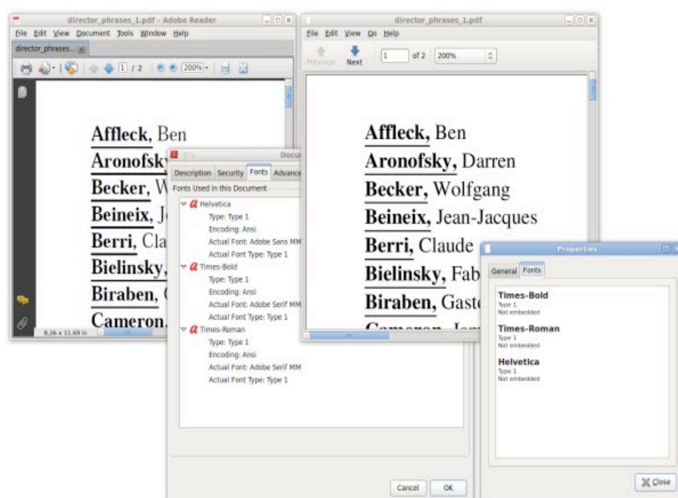


Figure 1 PDF file opened in Adobe Reader and Evince on Ubuntu

Adobe Reader on Ubuntu (see the left window) replaces Helvetica with Adobe Sans MM and Times-Roman with Adobe Serif MM. The MM refers to the fact that these are *Multiple Master* fonts. Wikipedia tells us that MM fonts are

"an extension to Adobe Systems' Type 1 Postscript fonts. [...] From one MM font, it is conceivable to create a wide gamut of typeface styles of different widths, weights, and proportions without losing the integrity or readability of the character glyphs."

Adobe Reader for Linux uses a generic font when it encounters a non-embedded font for which it can't find an exact match. Looking at the output of *File* > *Properties* > *Fonts* in Evince (Ubuntu's default document viewer), you might have the impression that the actual Times-Bold, Times-Roman, and Helvetica fonts are used, but that's just Evince fooling us. Helvetica and Times-Roman aren't present on my Linux distribution. Evince is using other fonts instead. On Ubuntu Linux, you can consult the configuration files in the `/etc/fonts` directory. I did, and I discovered that on my Linux installation, Times and Helvetica are mapped to Nimbus Roman No9 L and Nimbus Sans, free fonts that can be found in the `/usr/share/fonts/type1/gsfonts` directory.

Observe that we are looking at the same document, on the same operating system (Ubuntu Linux), yet the names of the directors in our document look slightly different because different fonts were used. We were very lucky that the names were legible. Not embedding fonts is always a risk, especially if you need special glyphs in your document. Not every font has the descriptions for every possible glyph.

NOTE:

Characters in a file are rendered on screen or on paper as glyphs. ISO-32000 9.2.1 states: "A character is an abstract symbol, whereas a glyph is a specific graphical rendering of a character. For example: The glyphs A, **A**, and *A* are renderings of the abstract 'A' character. Glyphs are organized into fonts. A font defines glyphs for a particular character set."

FAQ:

Why are the special characters missing in my PDF document? This isn't an iText problem. You could be using a character that has the description for a corresponding glyph on your system, but if you don't embed the font, that glyph can be missing on an end user's system. If the PDF viewer on that system can't find a substitution font, it won't be able to display the glyph. The solution is to embed the font. But even if you embed the font, some glyphs can be missing because they weren't even present in the font you tried to embed. The solution here is to use a different font that does have the appropriate glyph descriptions.

In the next example, we'll avoid possible problems caused by font substitution by embedding the font.

Embedding fonts

Up until now, we've created fonts using nothing but the `Font` object. The fonts that are available in this class are often referred to as the *standard type 1 fonts*. These fonts aren't embedded by iText. The next example is a variation on the previous one. We don't have to change listing 2; we only have to replace listing 1 with listing 3.

Listing 3 DirectorPhrases2.java

```
public static final Font BOLD;                                             A
public static final Font NORMAL;                                           A

static {
  BaseFont timesbd = null;
  BaseFont times = null;
  try {
    timesbd = BaseFont.createFont("c:/windows/fonts/timesbd.ttf",          B
      BaseFont.WINANSI, BaseFont.EMBEDDED);                                B
    times = BaseFont.createFont("c:/windows/fonts/times.ttf",              B
      BaseFont.WINANSI, BaseFont.EMBEDDED);                                B
    } catch (DocumentException e) {
      e.printStackTrace();
      System.exit(1);
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
    BOLD = new Font(timesbd, 12);                                          C
```

```
    NORMAL = new Font(times, 12);                                          C
  }

  public Phrase createDirectorPhrase(ResultSet rs)
    throws UnsupportedEncodingException, SQLException {
    Phrase director = new Phrase();
    Chunk name =
      new Chunk(new String(rs.getBytes("name"), "UTF-8"), BOLD);
    name.setUnderline(0.2f, -2f);                                          D
    director.add(name);
    director.add(new Chunk(",", BOLD));
    director.add(new Chunk(" ", NORMAL));
    director.add(new Chunk(new String(
      rs.getBytes("given_name"), "UTF-8"), NORMAL));
    director.setLeading(24);                                               E
    return director;
  }
```

A Font objects
B BaseFont objects
C Creates a Font using a BaseFont and a size
D A different way to underline a Chunk
E Defines a custom leading

In this example, we're telling iText where to find the font programs for Times New Roman (`times.ttf`) and Times New Roman Bold (`timesbd.ttf`) by creating a `BaseFont` object. We're asking iText to embed the characters (`BaseFont.EMBEDDED` versus `BaseFont.NOT_EMBEDDED`), using the ANSI character set (`BaseFont.WINANSI`). We can create a `Font` instance using a `BaseFont` object and a `float` value for the font size.

Figure 2 looks very similar to the screen shot in figure 1; only now, the PDF file is rendered in the exact same way in both viewers.

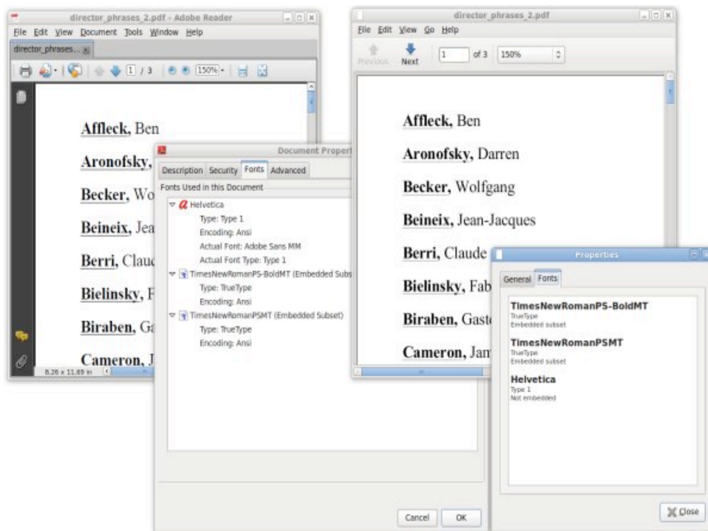


Figure 2 PDF file opened in Adobe Reader and Evince on Ubuntu

Observe that there's more space between the different names because we've used `setLeading()` to change the leading. The name of the directors are also underlined differently compared to the previous example, because we've chosen not to define "underline" as a property of the `Font`, but as an attribute of the `Chunk`.

With the method `Chunk.setUnderline()`, we can set the line thickness (in the example. 0.2 pt) and a Y position (in the example: 2 pt below the baseline). The parameter to set the Y position allows us to use the same method to strike a line through a `Chunk`. There's also a variant of the method that accepts six parameters:

1. A `Color`, if you want the line to have a different color than the text

2. The absolute thickness
3. A thickness multiplication factor that will adapt the line width based on the font size
4. An absolute Y position
5. A position multiplication factor that will adapt the Y position based on the font size
6. The end line cap, defining what the extremities of the line should look like. Allowed values are `PdfContentByte.LINE_CAP_BUTT` (the default value), `PdfContentByte.LINE_CAP_ROUND`, and `PdfContentByte.LINE_CAP_PROJECTING_SQUARE`.

One thing looks peculiar when you look at figure 2. Why do both viewers still list Helvetica? Looking at listing 2 and 3, you can't find any explicit reference to it, but it's added implicitly in this line:

```
document.add(Chunk.NEWLINE);
```

`Chunk.NEWLINE` contains a newline character in the default font; and the default font is Helvetica. We could have solved this by replacing that line with:

```
document.add(new Chunk("\n", NORMAL));
```

Summary

In this article, you've learned about `Chunk` objects and several `Chunk`'s attributes. You've worked with `Phrase`s and you've been introduced to the `Font` and `BaseFont` class. These are the building blocks of iText, which are often referred to as high-level objects.