iText 7 Building Blocks

Bruno Lowagie

iText Software

iText 7: Building Blocks

iText Software

This book is for sale at http://leanpub.com/itext7buildingblocks

This version was published on 2016-08-27

Contents

This is the second tutorial in the iText 7 series. In the first tutorial, iText 7: Jump-Start Tutorial[1], we discussed a series of examples that explained the core functionality of iText 7. In this book, we'll focus on the high-level building blocks that were introduced in the first chapter of that tutorial: Introducing basic building blocks[2]. In that chapter, we created PDFs using objects such as `Paragraph`, `List`, `Image` and `Table`, but we didn't go into detail. This tutorial is the extended version of that chapter. In this tutorial, you'll discover which building blocks are available and how they all relate to each other.

Throughout the book, we'll use the following symbols:

The information sign indicates interesting extra information, for instance about different options for a parameter, different flavors of a method, and so on.

The question mark will be used when this information is presented in the form of a question and an answer.

The bug highlights an `Exception` that gets thrown if you make a common mistake.

The text balloons are used for a chatty remark or a clarification.

The triangle with the exclamation point warns for functionality that was introduced at a later stage in the development of iText 7.

The key indicates functionality that is new in iText 7, or at least very different from what developers were used to in iText 5 or earlier versions.

All the examples of this book along with the resources needed to build them, are available online at the following address: http://developers.itextpdf.com/content/itext-7-building-blocks/examples[3]

[1]http://developers.itextpdf.com/content/itext-7-jump-start-tutorial

[2]http://developers.itextpdf.com/content/itext-7-jump-start-tutorial/chapter-1-introducing-basic-building-blocks

[3]http://developers.itextpdf.com/content/itext-7-building-blocks/examples

Before we start: Overview of the classes and interfaces

When we talk about iText 7's basic building blocks, we refer to all classes that implement the `IElement` interface. iText 7 is originally written in Java, then ported to C#. Because of our experience with both programming languages, we've adopted the convenient habit â€“typical for C# developersâ€“ to start every name of an interface with the letter `I`.

Figure 0.1 shows an overview of the relationship between `IElement` and some other interfaces.

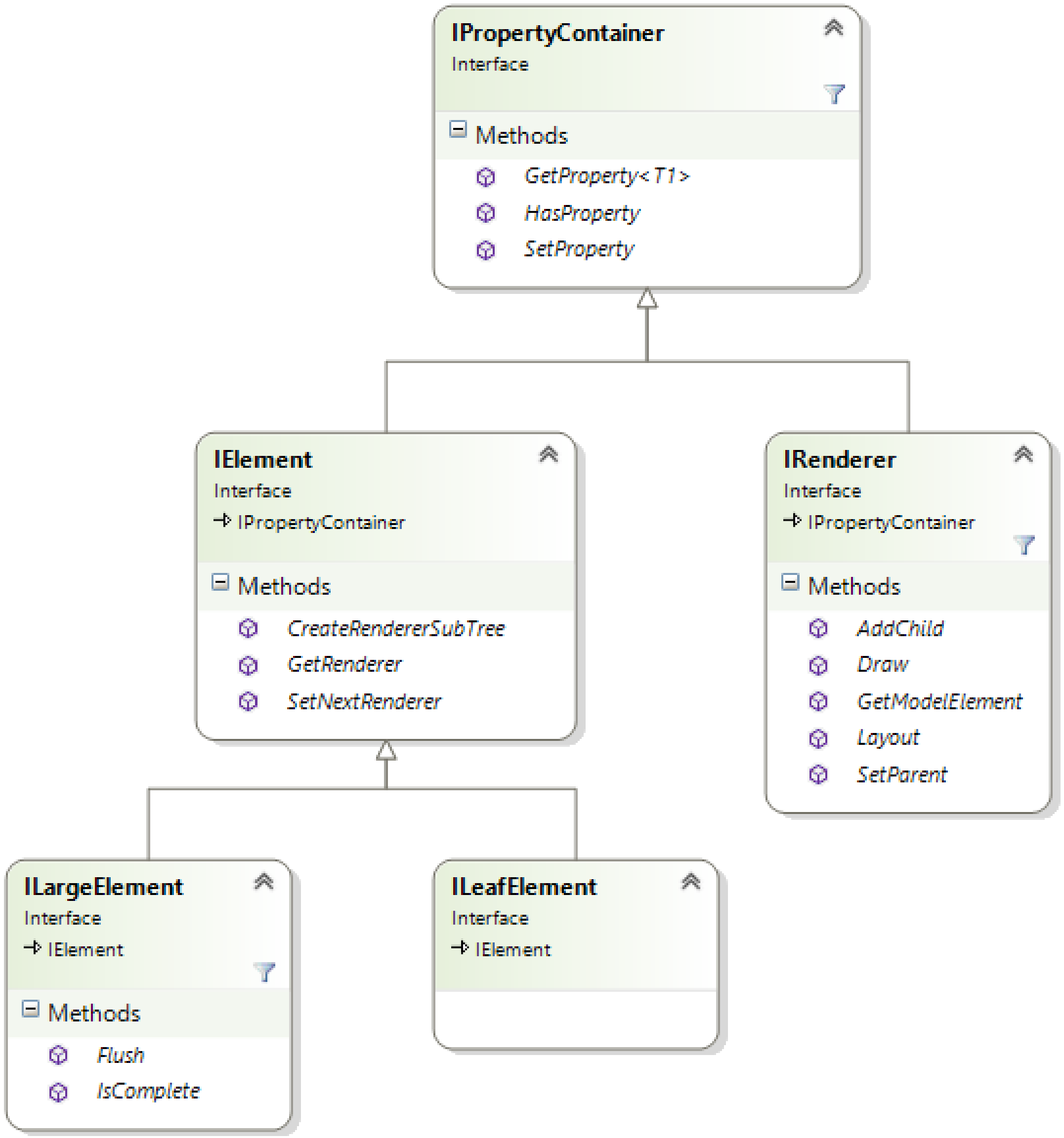


Figure 0.1: Overview of the interfaces

At the top of the hierarchy, we find the `IPropertyContainer` interface. This interface defines methods to set, get, and delete properties. This interfaces has two direct subinterfaces: `IElement` and `IRenderer`. The `IElement` interface will be implemented by objects such as `Text`, `Paragraph` and `Table`. These are the objects that we'll add to a document, either directly or indirectly. The `IRenderer` interface will be implemented by objects such as `TextRenderer`, `ParagraphRenderer` and `TableRenderer`. These renderers are used internally by iText, but we can subclass them if we want to tweak the way an object is rendered.

The `IElement` interface has two subinterfaces of its own. The `ILeafElement` interface will be implemented by building blocks that can't contain any other elements. For instance: you can add a `Text` or an `Image` element to a `Paragraph` object, but you can't add any object to a `Text` or an `Image` element. `Text` and `Image` implement the `ILeafElement` interface to reflect this. Finally, there's the `LargeElement` interface that allows you to render an object before you've finished adding all the content. It's implemented by the `Table` class, which means that you add a table to a document before you've completed adding all the `Cell` objects. By doing so, you can reduce the memory use: all the table content that can be rendered before the content of the table is completed, can be flushed from memory.

The `IPropertyContainer` interface is implemented by the abstract `ElementPropertyContainer` class. This class has three subclasses; see figure 0.2.

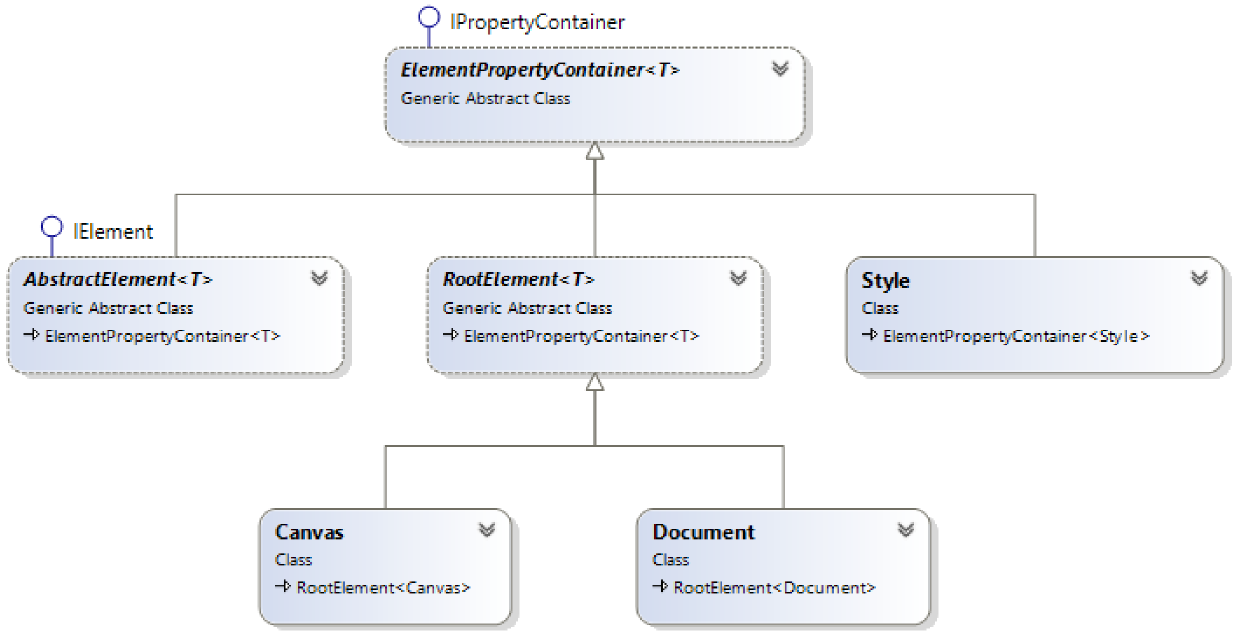


Figure 0.2: Implementations of the IPropertyContainer interface

The `Style` class is a container for all kinds of style attributes such as margins, paddings and rotation. It inherits style values such as widths, heights, colors, borders and alignments from the abstract `ElementPropertyContainer` class.

The `RootElement` class defines methods to add content, using either an `add()` method or a `showTextAligned()` method. The `Document` object will add this content to a page. The `Canvas` object doesn't know the concept of a page. It acts as a bridge between the high-level *layout* API and the low-level *kernel* API.

Figure 0.3 gives us an overview of the `AbstractElement` implementations.

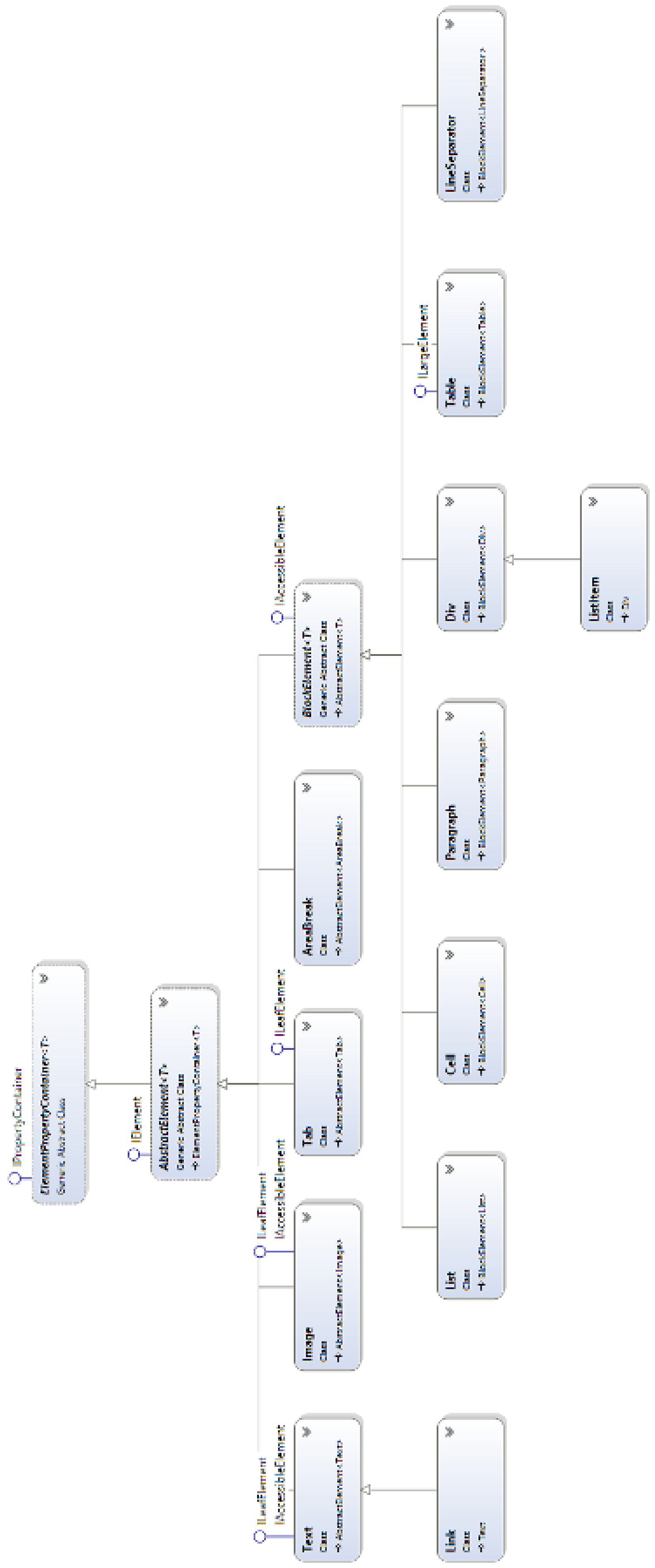


Figure 0.3: Implementations of the IElement interface

All classes derived from the `AbstractElement` class implement the `IElement` interface. `Text`, `Image`, `Tab` and `Link` also implement the `ILeafElement` interface. The `ILargeElement` interface is only implemented by the `Table` class. The basic building blocks make it very easy for you to create *tagged PDF*. Tagged PDF is a requirement for PDF/A, a standard for long-term preservation of document, and, PDF/UA, an accessibility standard. A properly tagged PDF includes semantic information about all the relevant content.

An ordinary PDF can show a human reader content that is organized as a table. This table is rendered using a bunch of text snippets and lines. To a machine, the table isn't more than that: text positioned at arbitrary places, lines drawn at arbitrary places. A seeing person can detect rows and columns and understand which rows are actually header or footer rows and which rows are body rows. There is no simple way for a machine to do this. When a machine detects a text snippet, it doesn't know if that text snippet is part of a paragraph, part of a title, part of a cell, or part of something else. When a PDF is tagged, it contains a structure tree that allows a machine to understand the structure of the content. Some text will be marked as part of a cell in a header row, other text will be marked as the caption of the table. All *real content* will be tagged. Other content, such as lines between rows and columns, running headers, page numbers, will be marked as an *artifact*.

In iText, we have introduced the `IAccessibleElement` interface. It is implemented by all the basic building blocks that contain real content: `Text`, `Link`, `Image`, `Paragraph`, `Div`, `List`, `ListItem`, `Table`, `Cell`, `LineSeparator`. If we define a `PdfDocument` as a tagged PDF using the `setTagged()` method, iText will create a structure tree so that a `Table` is properly tagged as a table, a `List` properly tagged as a list, and so on. There is no real content in a `Tab` or an `AreaBreak`, which is why these classes don't implement that interface. It's just white space; a tab and an area break don't even need to be marked as an artifact.

In this tutorial, we won't create tagged PDF; iText will just render the content to the document using the appropriate `IRenderer` implementation. Figure 0.4 shows an overview of the `IRenderer` implementations.

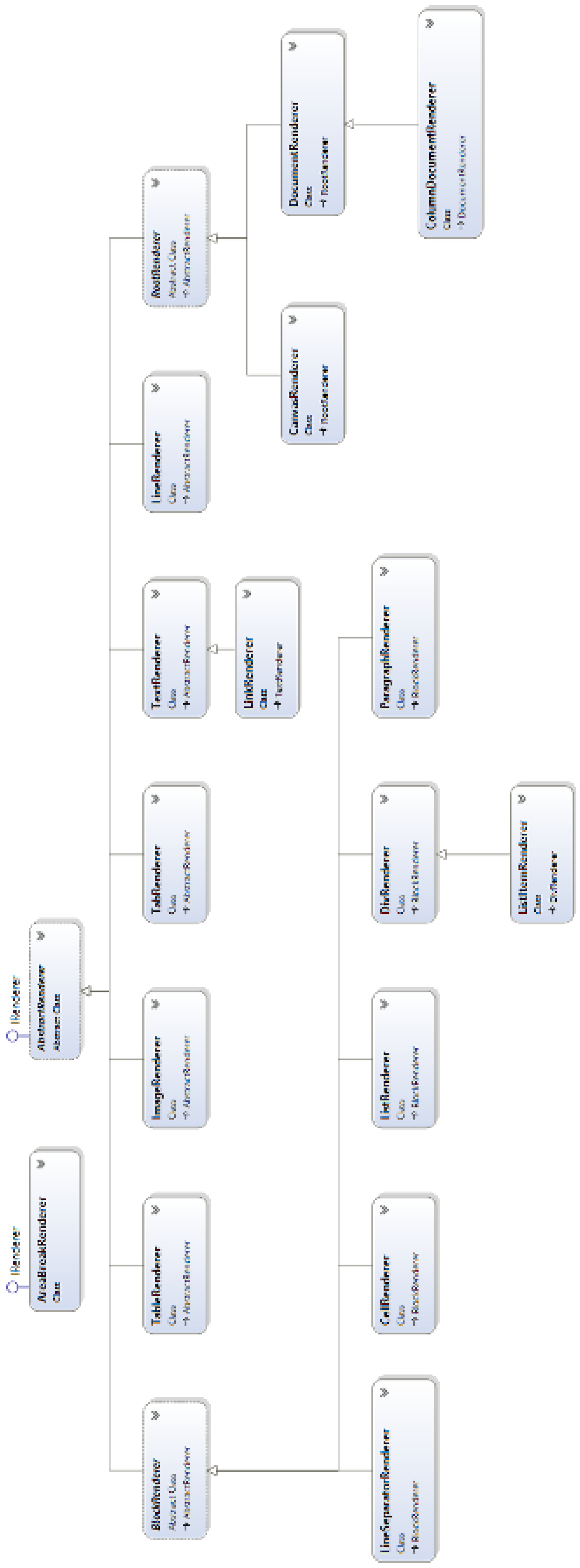


Figure 0.4: Implementations of the IRenderer interface

When you compare figure 0.4 with 0.3, you'll discover that each `AbstractElement` and each `RootElement` has its corresponding renderer. We won't discuss figure 0.4 in much detail. The concept of renderers will become clear the moment we start making some examples.

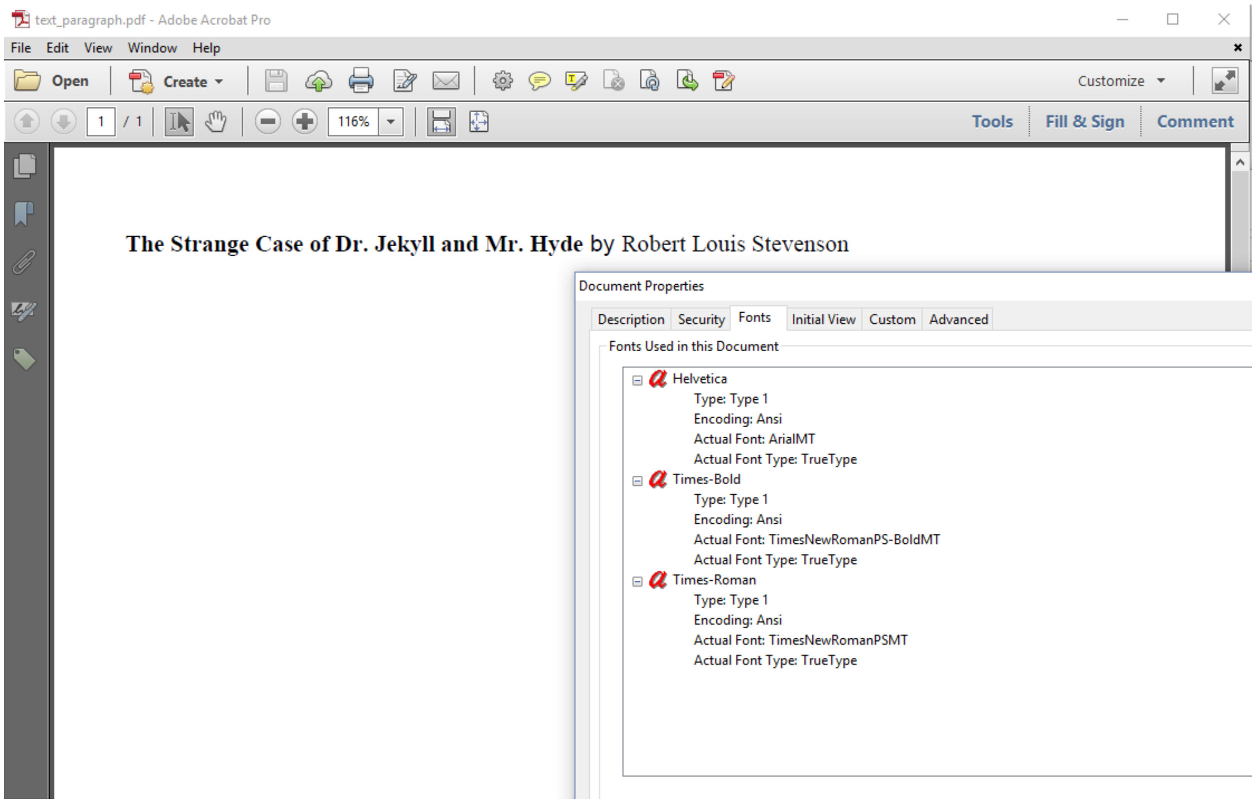
Chapter 1: Introducing the PdfFont class

When writing a tutorial, I always prefer working with real-world use cases. That's not always easy because real-world use cases can get quite complex, whereas a tutorial needs to explain different concepts as simple as possible. While I was looking for a theme for this tutorial, I stumbled upon the short story "The Strange Case of Dr. Jekyll and Mr. Hyde" by Robert Louis Stevenson. I made a first example turning a plain text file into a PDF eBook and I liked the result. When I discovered how many movies, cartoons and series were made based on this work, I saw an opportunity to create a database that could be converted into a table. The movie posters could serve as sample material when discussing images in PDF.

But first things first: let's start with an example that displays the title and the author in different fonts. The `PdfFont` class doesn't appear in any of the hierarchical charts showing the relationship between element interfaces and classes, but it's needed for all the building blocks that involve text. We could spend a complete tutorial about fonts (and we probably will), but this chapter will explain the basic font functionality that you need to be aware of.

Creating a PdfFont object

If we look at figure 1.1, we see that three different fonts were used to create a PDF document with the title and the author of the Jekyll and Hyde story: Helvetica, Times-Bold and Times-Roman. In reality, three other fonts are used by the viewer: ArialMT, TimesNewRomanPS-BoldMT and TimesNewRomanPSMT.


text_paragraph.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
116%
Tools
Fill & Sign
Comment

The Strange Case of Dr. Jekyll and Mr. Hyde by Robert Louis Stevenson

Document Properties

Description | Security | Fonts | Initial View | Custom | Advanced

Fonts Used in this Document

- Helvetica
  - Type: Type 1
  - Encoding: Ansi
  - Actual Font: ArialMT
  - Actual Font Type: TrueType
- Times-Bold
  - Type: Type 1
  - Encoding: Ansi
  - Actual Font: TimesNewRomanPS-BoldMT
  - Actual Font Type: TrueType
- Times-Roman
  - Type: Type 1
  - Encoding: Ansi
  - Actual Font: TimesNewRomanPSMT
  - Actual Font Type: TrueType

Figure 1.1: Standard Type 1 fonts

The MT in the names of the *Actual Font* refers to the vendor of the fonts: the Monotype Imaging Holdings, Inc. These are fonts shipped with Microsoft Windows. If you'd open the same file on a Linux machine, other fonts would be used as actual fonts. This is typically what happens when you don't embed fonts. The viewer searches the operating system for the fonts that are needed to present the document. If a specific font can be found, another font will be used instead.

Traditionally, there are 14 fonts that every PDF viewer should be able to recognize and render in a reliable way: four Helvetica fonts (normal, bold, oblique, and bold-oblique), four Times-Roman fonts (normal, bold, italic, and bold-italic), four Courier fonts (normal, bold, oblique, and bold-oblique), Symbol and Zapfdingbats. These fonts are often referred to as the Standard Type 1 fonts. Not every viewer will use that exact font, but it will use a font that looks almost identical.

To create the PDF shown in figure 1.1, we used three of these fonts: we defined two fonts explicitly; one font was defined implicitly. See the Text_Paragraph[4] example.

[4]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1822-c01e01_text_paragraph.java

```
 1 PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
 2 Document document = new Document(pdf);
 3 PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
 4 PdfFont bold = PdfFontFactory.createFont(FontConstants.TIMES_BOLD);
 5 Text title =
 6     new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
 7 Text author = new Text("Robert Louis Stevenson").setFont(font);
 8 Paragraph p = new Paragraph().add(title).add(" by ").add(author);
 9 document.add(p);
10 document.close();
```

In line 1, we create a `PdfDocument` using a `PdfWriter` as parameter. These are low-level objects that will create PDF output based on your content. We're creating a `Document` instance in line 2. This is a high-level object that will allow you to create a document without having to worry about the complexity of PDF syntax.

In lines 5 and 6, we create a `PdfFont` using the `PdfFontFactory`. In the `FontConstants` object, you'll find a constant for each of the 14 Standard Type 1 fonts. In line 7, we create a `Text` object with the title of Stevenson's short story and we set the font to `TIMES_BOLD`. In line 8, we create a `Text` object with the name of the author and we set the font to `TIMES_ROMAN`. We can't add these `Text` objects straight to the `document`, but we add them to a `BlockElement`, more specifically a `Paragraph`, in line 9.

> Between the `title` and the `author`, we add `" by "` as a `String` object. Since we didn't define a font for this `String`, the default font of the `Paragraph` is used. In iText, the default font is Helvetica. This explains why we see the font Helvetica listed in the font overview in figure 1.1.

In line 10, we add the paragraph to the `document` object; we close the `document` object in line 11.

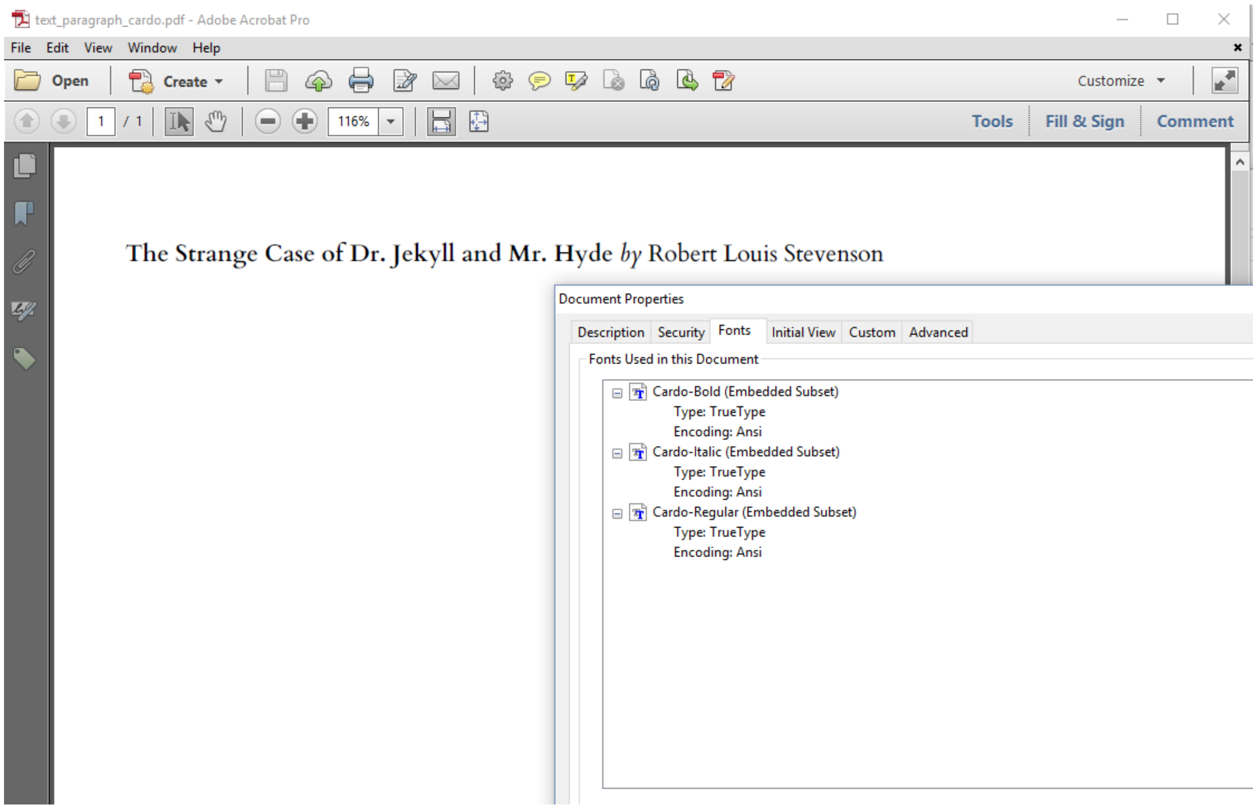
We have created our first Jekyll and Hyde PDF using fonts that aren't embedded. As a result, slightly different fonts can be used when rendering the document. We can avoid this by embedding the fonts.

Embedding a font

iText supports the Standard Type 1 fonts, because the io-jar contains the Adobe Font Metrics (AFM) files of those 14 fonts. iText can't embed these 14 fonts because the PostScript Font Binary (PFB) files are proprietary. They can't be shipped with iText because iText Group doesn't have a license to do so. We are only allowed to ship the metrics files.

In the Text_Paragraph_Cardo[5] example, we use three fonts of the Cardo font family. These are fonts that were released under the Summer Institute of Logistics (SIL) Open Font License (OFL). The result is shown in figure 1.2.

[5]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1823-c01e02_text_paragraph_cardo.java


text_paragraph_cardo.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
Tools
Fill & Sign
Comment
116%

The Strange Case of Dr. Jekyll and Mr. Hyde *by* Robert Louis Stevenson

Document Properties

Description | Security | Fonts | Initial View | Custom | Advanced

Fonts Used in this Document

```
Cardo-Bold (Embedded Subset)
    Type: TrueType
    Encoding: Ansi
Cardo-Italic (Embedded Subset)
    Type: TrueType
    Encoding: Ansi
Cardo-Regular (Embedded Subset)
    Type: TrueType
    Encoding: Ansi
```

Figure 1.2: Embedded fonts

First we need the path to the font programs for the three Cardo fonts: `Cardo-Regular.ttf`, `Cardo-Bold.ttf` and `Cardo-Italic.ttf`:

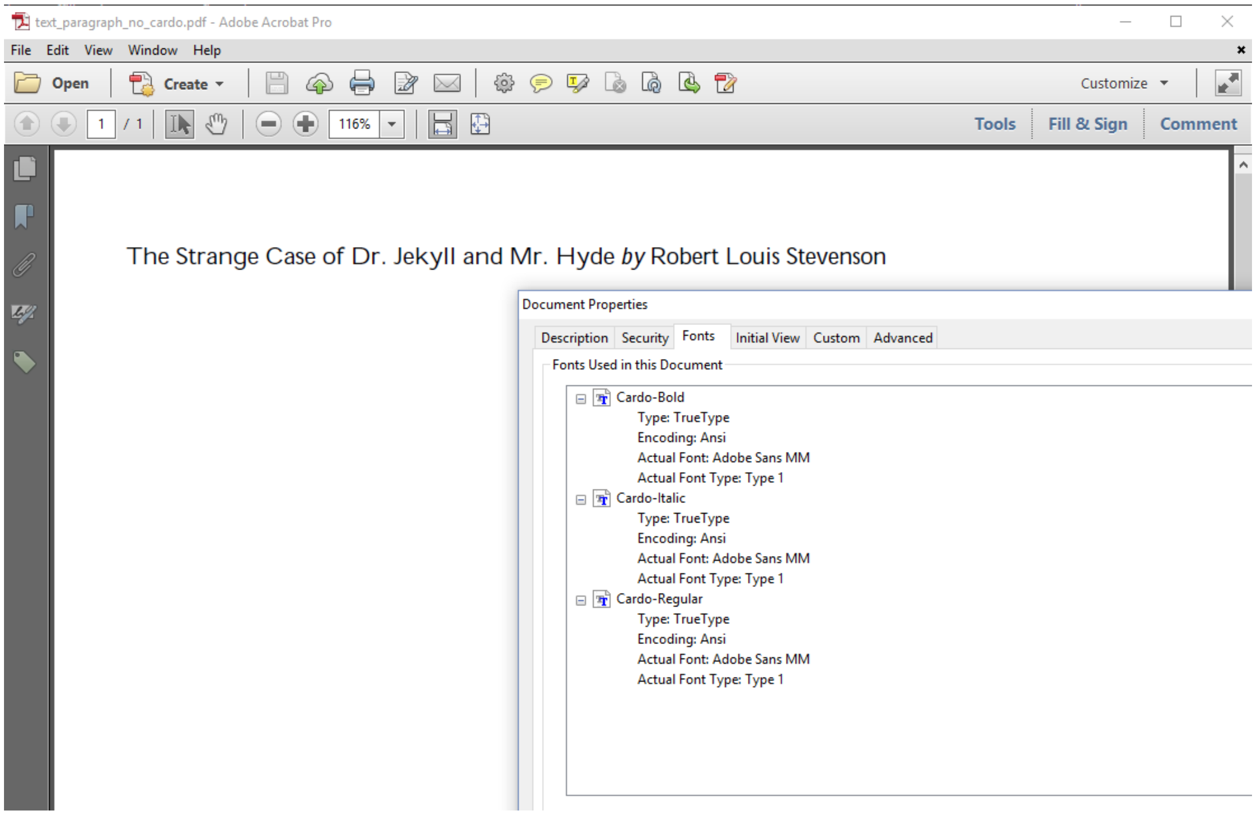
```
1 public static final String REGULAR =
2     "src/main/resources/fonts/Cardo-Regular.ttf";
3 public static final String BOLD =
4     "src/main/resources/fonts/Cardo-Bold.ttf";
5 public static final String ITALIC =
6     "src/main/resources/fonts/Cardo-Italic.ttf";
```

In line 1 to 3 of the following snippet, we use these paths as the first parameter of the `createFont()` method. The second parameter is a Boolean indicating whether or not we want to embed the font.

```java
1  PdfFont font = PdfFontFactory.createFont(REGULAR, true);
2  PdfFont bold = PdfFontFactory.createFont(BOLD, true);
3  PdfFont italic = PdfFontFactory.createFont(ITALIC, true);
4  Text title =
5      new Text("The Strange Case of Dr. Jekyll and Mr. Hyde").setFont(bold);
6  Text author = new Text("Robert Louis Stevenson").setFont(font);
7  Paragraph p = new Paragraph().setFont(italic)
8      .add(title).add(" by ").add(author);
9  document.add(p);
```

Line 4 to 6 are identical to what we had before, but in line 7, we change the default font of the `Paragraph` to `italic`. This explains why `" by "` was written in italic in figure 1.2 and why the font Helvetica no longer appears in the font list. In line 7, we add the `Paragraph` to the `Document` instance.

Figure 1.3 shows what would happen if we don't embed the fonts.


text_paragraph_no_cardo.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
116%
Tools
Fill & Sign
Comment
Document Properties
Description Security Fonts Initial View Custom Advanced

The Strange Case of Dr. Jekyll and Mr. Hyde *by* Robert Louis Stevenson

Fonts Used in this Document

- Cardo-Bold
  - Type: TrueType
  - Encoding: Ansi
  - Actual Font: Adobe Sans MM
  - Actual Font Type: Type 1
- Cardo-Italic
  - Type: TrueType
  - Encoding: Ansi
  - Actual Font: Adobe Sans MM
  - Actual Font Type: Type 1
- Cardo-Regular
  - Type: TrueType
  - Encoding: Ansi
  - Actual Font: Adobe Sans MM
  - Actual Font Type: Type 1

Figure 1.3: Ugly font substitution

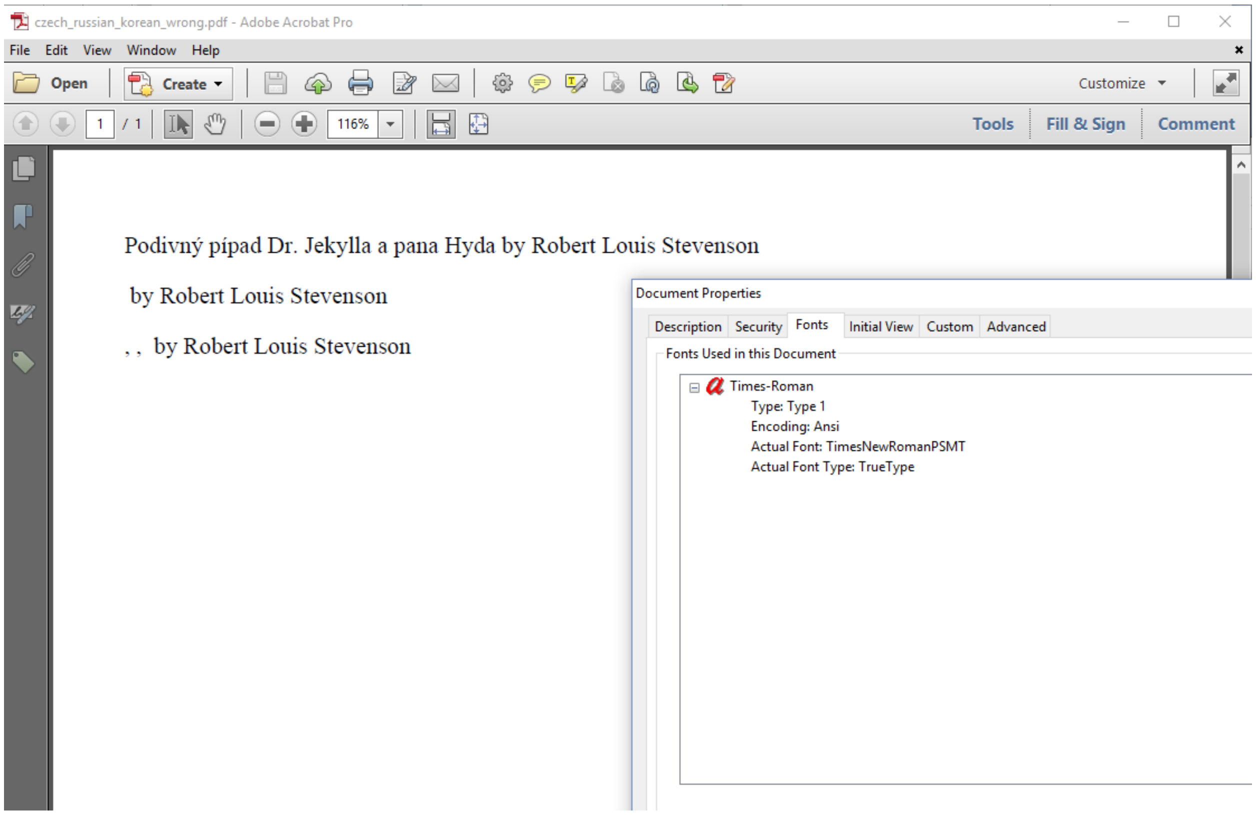
In the Text_Paragraph_NoCardo[6] example, we have defined the fonts like this:

[6] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1824-c01e03_text_paragraph_nocardo.java

```
PdfFont font = PdfFontFactory.createFont(REGULAR);
PdfFont bold = PdfFontFactory.createFont(BOLD);
PdfFont italic = PdfFontFactory.createFont(ITALIC);
```

The constants `REGULAR`, `BOLD` and `ITALIC` refer to the correct Cardo `.ttf` files, but we omitted the parameter that tells iText to embed the font. Incidentally, the Cardo fonts aren't present on my PC. Adobe Reader replaced them with Adobe Sans MM. As you can see, the result doesn't look nice. If you don't use any of the standard Type 1 fonts, you should always embed the font.

The problem is even worse when you try to create PDFs in different languages. In figure 1.4, we try to add some text in Czech, Russian and Korean. The Czech text looks more or less OK, but we'll soon discover that there one character missing. The Russian and Korean text is invisible.


czech_russian_korean_wrong.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
Tools
Fill & Sign
Comment
Document Properties
Description Security Fonts Initial View Custom Advanced
Fonts Used in this Document

Podivný pípad Dr. Jekylla a pana Hyda by Robert Louis Stevenson

by Robert Louis Stevenson

, , by Robert Louis Stevenson

Times-Roman
Type: Type 1
Encoding: Ansi
Actual Font: TimesNewRomanPSMT
Actual Font Type: TrueType

Figure 1.4: Wrong rendering of Czech, Russian and Korean

Not embedding the font isn't the only problem here. We also need to define the appropriate encoding.

Choosing the appropriate encoding

In figure 1.4, we tried to render the following text:

> Podivný případ Dr. Jekylla a pana Hyda by Robert Louis Stevenson

Странная история доктора Джекила и мистера Хайда by Robert Louis Stevenson

⊠⊠⊠, ⊠⊠, ⊠ by Robert Louis Stevenson

The first line is the Czech translation of “The Strange Case of Dr. Jekyll and Mr. Hyde.” If you look closely at figure 1.4, you’ll see that the character ř is missing. That’s because the ř character is missing in the Winansi encoding. Winansi, also known as code page 1252 (CP-1252), Windows 1252, or Windows Latin 1, is a superset of ISO 8859-1 also known as Latin-1. It’s a character encoding of the Latin alphabet, used by default in many applications on Western operating systems.

For the Czech text, we need to use another encoding. One option is to use code page 1250, an encoding to represent text in Central European and Eastern European languages that use Latin script. The second line reads as *Strannaya istoriya doktora Dzhekila i mistera Khayda*. For this text, we could use code page 1251, an encoding designed to cover languages that use the Cyrillic script. Cp1250 and Cp1251 are 8-bit character encodings. The third line is Korean for *Hyde, Jekyll, Me*, a South-Korean television series loosely based on the Jekyll and Hyde story. We can’t use an 8-bit encoding for Korean. To render this text, we need to use Unicode. Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world’s writing systems.

When you create a font using an 8-bit encoding, iText will create a *simple font* for the PDF. A simple font consists of at most 256 characters that are mapped to at most 256 glyphs. When you create a font using Unicode (in PDF terms: *Identity-H* for horizontal writing systems or *Identity-V* for vertical writing systems), iText will create a *composite font*. A composite font can contain 65,536 characters. This is less than the total number of available code points in Unicode (1,114,112). This means that no single font can contain all possible characters in every possible language.

Instead of Cp1250 and Cp1251, we could also use Unicode for the Czech and Russian text. Actually, when we store hard-coded text in source code, it is preferred to store Unicode values.

```java
public static final String CZECH =
        "Podivn\u00fd p\u0159\u00edpad Dr. Jekylla a pana Hyda";
public static final String RUSSIAN =
        "\u0421\u0442\u0440\u0430\u043d\u043d\u0430\u044f "
        + "\u0438\u0441\u0442\u043e\u0440\u0438\u044f "
        + "\u0434\u043e\u043a\u0442\u043e\u0440\u0430 "
        + "\u0414\u0436\u0435\u043a\u0438\u043b\u0430 \u0438 "
        + "\u043c\u0438\u0441\u0442\u0435\u0440\u0430 "
        + "\u0425\u0430\u0439\u0434\u0430";
public static final String KOREAN =
        "\ud558\uc774\ub4dc, \uc9c0\ud0ac, \ub098";
```

We'll use the values `CZECH`. `RUSSIAN` and `KOREAN` in our next couple of examples.

Why should we always use Unicode notations for special characters?

When the source code file is stored on disk, committed to a version control system, or transferred in any way, there's always a risk that the encoding gets lost. If a Unicode file is stored as plain text, two-byte characters change into two single-byte characters. For example, the character 킬 with Unicode value `\ud0ac` will change into two characters with ASCII code `d0` and `ac`. When this happens the syllable 킬 (pronounced as "kil") changes into `Ð¬` and the text becomes illegible. It is good practice to use the Unicode notation as done in the above snippet; this will help you avoid encoding problems with your source code.

Using the correct encoding isn't sufficient to solve every font problem you might encounter. In the Czech_Russian_Korean_Wrong[7] example, we create the `Paragraph` objects like this:

```java
PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
document.add(new Paragraph().setFont(font)
        .add(CZECH).add(" by Robert Louis Stevenson"));
document.add(new Paragraph().setFont(font)
        .add(RUSSIAN).add(" by Robert Louis Stevenson"));
document.add(new Paragraph().setFont(font)
        .add(KOREAN).add(" by Robert Louis Stevenson"));
```

This won't work because we didn't use the correct encoding, but also because we didn't define a font that supports Russian and Korean. We fix this problem in the Czech_Russian_Korean[8] example by embedding the free font "FreeSans" for the Czech and Russian translation of the title. We'll use a Hancom font "HCR Batang" for the Korean text.

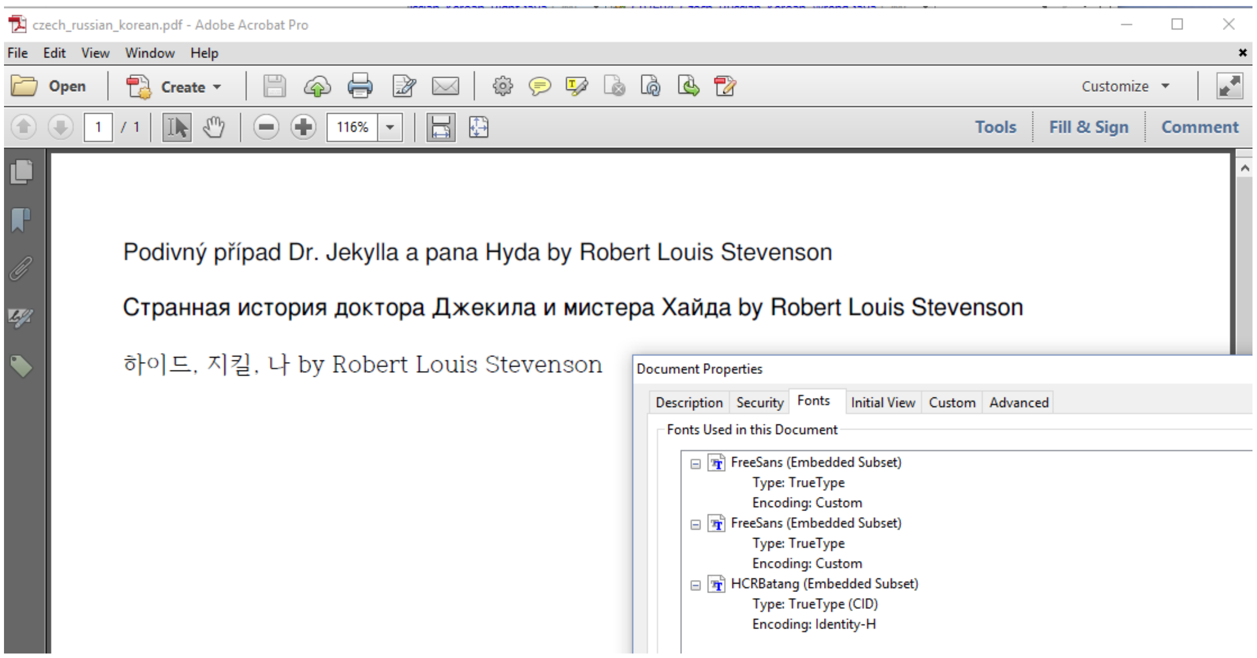
```java
public static final String FONT = "src/main/resources/fonts/FreeSans.ttf";
public static final String HCRBATANG = "src/main/resources/fonts/HANBatang.ttf";
```

We'll use these paths as the first parameter for the `PdfFont` constructor. We pass the desired encoding as the second parameter. The third parameter indicates that we want to embed the font.

[7]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1836-c01e04_czech_russian_korean_wrong.java
[8]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1837-c01e05_czech_russian_korean_right.java

```
PdfFont font1250 = PdfFontFactory.createFont(FONT, PdfEncodings.CP1250, true);
document.add(new Paragraph().setFont(font1250)
        .add(CZECH).add(" by Robert Louis Stevenson"));
PdfFont font1251 = PdfFontFactory.createFont(FONT, "Cp1251", true);
document.add(new Paragraph().setFont(font1251)
        .add(RUSSIAN).add(" by Robert Louis Stevenson"));
PdfFont fontUnicode =
    PdfFontFactory.createFont(HCRBATANG, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(fontUnicode)
        .add(KOREAN).add(" by Robert Louis Stevenson"));
```

Figure 1.5 shows the resulting PDF.


czech_russian_korean.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
Tools
Fill & Sign
Comment

Podivný případ Dr. Jekylla a pana Hyda by Robert Louis Stevenson

Странная история доктора Джекила и мистера Хайда by Robert Louis Stevenson

하이드, 지킬, 나 by Robert Louis Stevenson

Document Properties

Description Security Fonts Initial View Custom Advanced

Fonts Used in this Document

FreeSans (Embedded Subset)
Type: TrueType
Encoding: Custom
FreeSans (Embedded Subset)
Type: TrueType
Encoding: Custom
HCRBatang (Embedded Subset)
Type: TrueType (CID)
Encoding: Identity-H

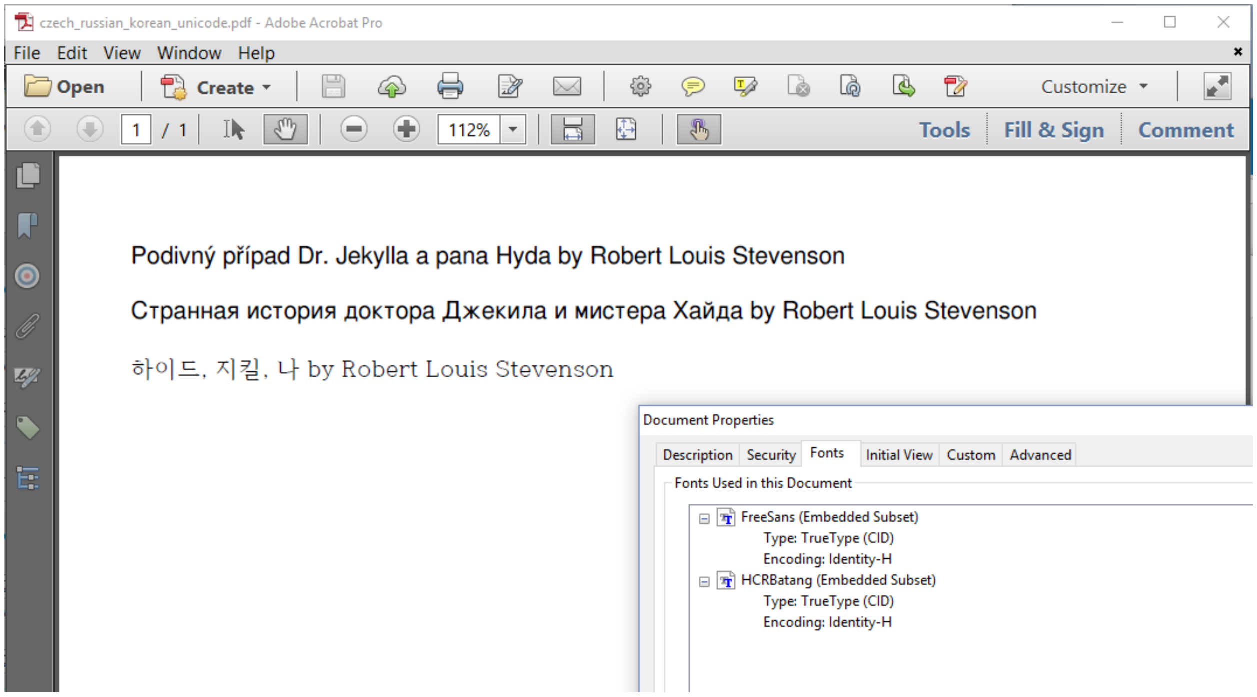
Figure 1.5: Correct rendering of Czech, Russian and Korean

When we look at the Fonts panel in the document properties, we notice that FreeSans is mentioned twice. That is correct: we've added the font once with the encoding Cp1250 and once with the encoding Cp1251, In the Czech_Russian_Korean_Unicode[9] example, we'll create one composite font, `freeUnicode`, for both languages, Czech and Russian.

[9]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1844-c01e06_czech_russian_korean_unicode.java

```
PdfFont freeUnicode =
    PdfFontFactory.createFont(FONT, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(freeUnicode)
    .add(CZECH).add(" by Robert Louis Stevenson"));
document.add(new Paragraph().setFont(freeUnicode)
    .add(RUSSIAN).add(" by Robert Louis Stevenson"));
PdfFont fontUnicode =
    PdfFontFactory.createFont(HCRBATANG, PdfEncodings.IDENTITY_H, true);
document.add(new Paragraph().setFont(fontUnicode)
    .add(KOREAN).add(" by Robert Louis Stevenson"));
```

Figure 1.6 shows the result. The page looks identical to what we saw in figure 1.5, but now the PDF only contains one FreeSans font with Identity-H as encoding.


czech_russian_korean_unicode.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
112%
Tools
Fill & Sign
Comment
Document Properties
Description Security Fonts Initial View Custom Advanced

Podivný případ Dr. Jekylla a pana Hyda by Robert Louis Stevenson

Странная история доктора Джекила и мистера Хайда by Robert Louis Stevenson

하이드, 지킬, 나 by Robert Louis Stevenson

Fonts Used in this Document

FreeSans (Embedded Subset)
Type: TrueType (CID)
Encoding: Identity-H
HCRBatang (Embedded Subset)
Type: TrueType (CID)
Encoding: Identity-H

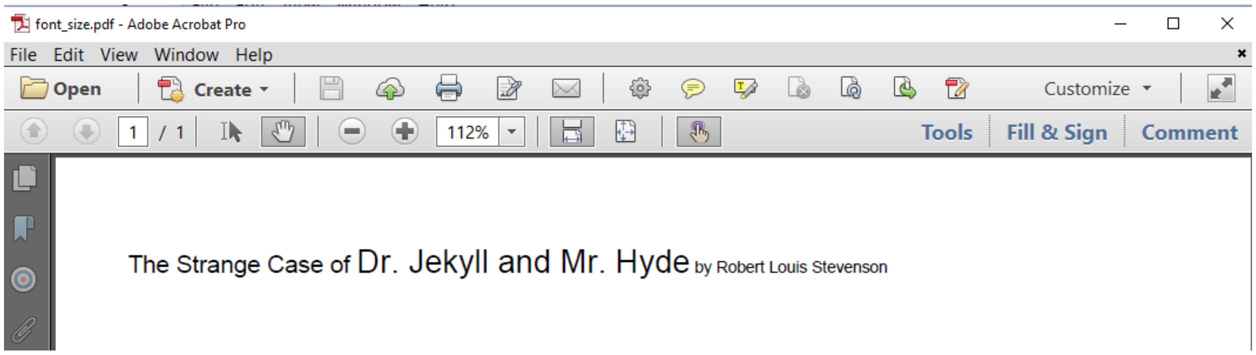
Figure 1.6: Correct rendering of Czech, Russian and Korean (Unicode)

Using Unicode is one of the requirements of PDF/UA and of certain flavors of PDF/A for reasons of accessibility. With custom encodings, it isn't always possible to know which glyphs are represented by each character.

In the next series of font examples, we'll experiment with some font properties such as font size, font color, and rendering mode.

Font properties

Figure 1.7 shows a screen shot of yet another PDF with the Jekyll and Hyde title. This time, the default font Helvetica is used, but we've defined different font sizes.



The Strange Case of Dr. Jekyll and Mr. Hyde by Robert Louis Stevenson

Figure 1.7: Different font sizes

The font size is set with the `setFontSize()` method. This method is defined in the abstract class `ElementPropertyContainer`, which means that we can use it on many different objects. In the FontSize[10] example, we use the method on `Text` and `Paragraph` objects:

```
Text title1 = new Text("The Strange Case of ").setFontSize(12);
Text title2 = new Text("Dr. Jekyll and Mr. Hyde").setFontSize(16);
Text author = new Text("Robert Louis Stevenson");
Paragraph p = new Paragraph().setFontSize(8)
        .add(title1).add(title2).add(" by ").add(author);
document.add(p);
```

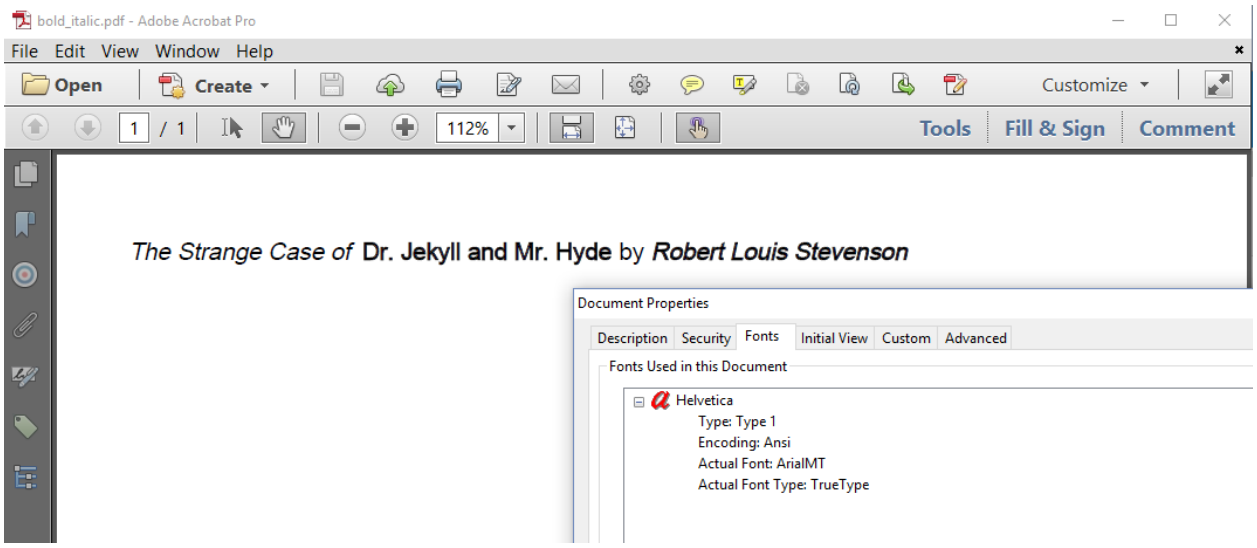
We set the font size of the newly created `Paragraph` to 8 pt. This font size will be inherited by all the objects that are added to the `Paragraph`, unless the objects override that default size. This is the case for `title1` for which we defined a font size of 12 pt and for `title2` for which we defined a font size of 16 pt. The content added as a `String` (`" by "`) and the content added as a `Text` object for which no font size was defined inherit the font size 8 pt from the `Paragraph` to which they are added.

In iText 5, it was necessary to create a different `Font` object if you wanted a font with a different size or color. We changed this in iText 7: you only need a single `PdfFont` object. The font size and color is defined at the level of the building blocks. We also made it possible for elements to inherit the font, font size, font color and other properties from the parent object.

In previous examples, we've worked with different fonts from the same family. For instance, we've created a document with three different fonts from the Cardo family: Cardo-Regular, Cardo-Bold,

[10]http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1846-c01e07_fontsize.java

and Cardo-Italic. For most of the Western fonts, you'll find at least a regular font, a bold font, an italic font, and a bold-italic font. It will be more difficult to find bold, italic and bold-italic fonts for Eastern and Semitic languages. In that case, you'll have to mimic those styles as is done in figure 1.8. If you look closely, you see that different styles are used, yet we've only defined a single font in the PDF.


bold_italic.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
112%
Tools
Fill & Sign
Comment

The Strange Case of **Dr. Jekyll and Mr. Hyde** by ***Robert Louis Stevenson***

Document Properties

Description Security Fonts Initial View Custom Advanced

Fonts Used in this Document

Helvetica
Type: Type 1
Encoding: Ansi
Actual Font: ArialMT
Actual Font Type: TrueType

Figure 1.8: Mimicking different font styles

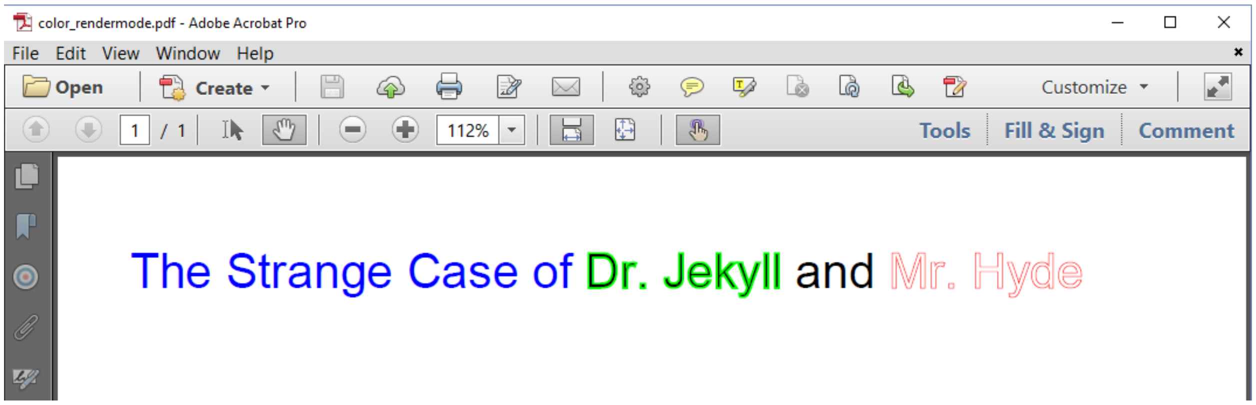
Let's take a look at the BoldItalic[11] example to find out how this was done.

```
Text title1 = new Text("The Strange Case of ").setItalic();
Text title2 = new Text("Dr. Jekyll and Mr. Hyde").setBold();
Text author = new Text("Robert Louis Stevenson").setItalic().setBold();
Paragraph p = new Paragraph()
        .add(title1).add(title2).add(" by ").add(author);
document.add(p);
```

In lines 1 to 3, we use the methods `setItalic()` and `setBold()`. The `setItalic()` method won't switch from a regular to an italic font. Instead, it will skew the glyphs of the italic font in such a way that it looks as if they are italic. The `setBold()` font will change the render mode of the text and increase the stroke width. Let's introduce some color to show what this means.

Figure 1.9 shows the text using different colors and different rendering modes.

[11] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1847-c01e08_bolditalic.java


color_rendermode.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
112%
Tools
Fill & Sign
Comment

The Strange Case of Dr. Jekyll and Mr. Hyde

Figure 1.9: different font colors and rendering modes

The ColorRendering[12] example explains what happens.

```
1  Text title1 = new Text("The Strange Case of ").setFontColor(Color.BLUE);
2  Text title2 = new Text("Dr. Jekyll")
3          .setStrokeColor(Color.GREEN)
4          .setTextRenderingMode(PdfCanvasConstants.TextRenderingMode.FILL_STROKE);
5  Text title3 = new Text(" and ");
6  Text title4 = new Text("Mr. Hyde")
7          .setStrokeColor(Color.RED).setStrokeWidth(0.5f)
8          .setTextRenderingMode(PdfCanvasConstants.TextRenderingMode.STROKE);
9  Paragraph p = new Paragraph().setFontSize(24)
10         .add(title1).add(title2).add(title3).add(title4);
11 document.add(p);
```

A font program contains the syntax to construct the path of each glyph. By default the path is painted using a *fill* operator, not drawn with a *stroke* operation, but we can change this default.

- In line 1, we change the font color to blue using the `setFontColor()` method. This changes the *fill color* for the paint operation that fills the paths of all the text.
- In line 2-4, we don't define a font color, which means the text will be painted in black. Instead we define a stroke color using the `setStrokeColor()` method, and we change the text rendering mode to `FILL_STROKE` with the `setTextRenderingMode()` method. As a result the contours of each glyph will be drawn in green. Inside those contours, we'll see the default fill color black.
- We don't change any of the defaults in line 5. This `Text` object will simply inherit the font size of the `Paragraph`, just like all of the other `Text` objects.
- In line 6-8, we change the stroke color to red and we use the `setStrokeWidth()` to 0.5 user units. By default, the stroke width is 1 user unit, which by default corresponds with 1 point.

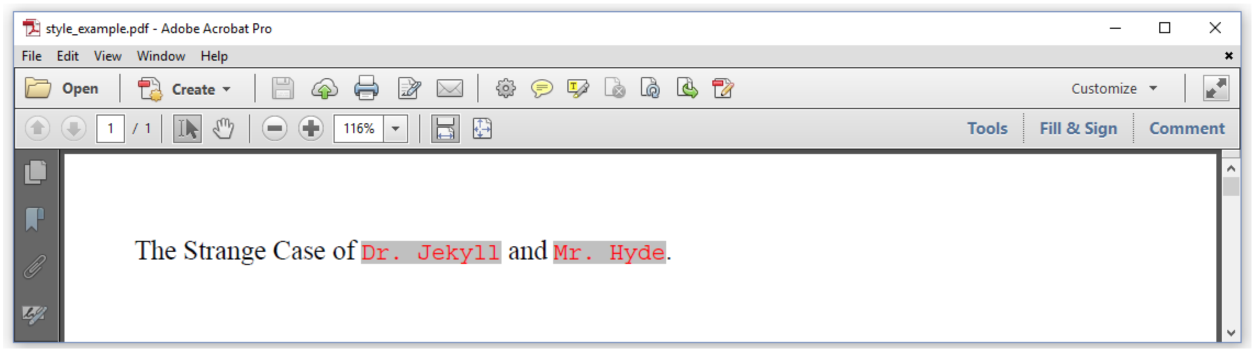
[12] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1848-c01e09_colorrendering.java

There are 72 user units in one inch by default. We also change the text rendering mode to `STROKE` which means the text won't be filled using the default fill color. Instead, we'll only see the contours of the text.

Mimicking bold is done by setting the text rendering mode to `FILL_STROKE` and by increasing the stroke width; mimicking italic is done by using the `setSkew()` method that will be discussed in chapter 3. Although this approach works relatively well, the `setBold()` and `setItalic()` method should only be used as a last resort when it's really impossible to find the appropriate fonts for the desired styles. Mimicking styles makes it very hard –if not impossible– for parsers extracting text from PDF to detect which part of the text is rendered in a different style.

Reusing styles

If you have many different building blocks, it can become quite cumbersome to define the same style over and over again for each separate object. See for instance figure 1.10 where parts of the text –the title of a story– are written in 14 pt Times-Roman, but other parts –the names of the main characters– are written in 12 pt Courier with red text on a light gray background.


style_example.pdf - Adobe Acrobat Pro
File Edit View Window Help
Open
Create
Customize
1 / 1
116%
Tools
Fill & Sign
Comment

The Strange Case of Dr. Jekyll and Mr. Hyde.

Figure 1.10: reusing styles

We could define the font family, font size, font color and background for each separate `Text` object that is added to the title `Paragraph`, but in the ReusingStyles[13] example, we use the `Style` object to define all the different styles at once.

[13] http://developers.itextpdf.com/content/itext-7-building-blocks/examples/chapter-1#1889-c01e10_reusingstyles.java

```
 1  Style normal = new Style();
 2  PdfFont font = PdfFontFactory.createFont(FontConstants.TIMES_ROMAN);
 3  normal.setFont(font).setFontSize(14);
 4  Style code = new Style();
 5  PdfFont monospace = PdfFontFactory.createFont(FontConstants.COURIER);
 6  code.setFont(monospace).setFontColor(Color.RED)
 7          .setBackgroundColor(Color.LIGHT_GRAY);
 8  Paragraph p = new Paragraph();
 9  p.add(new Text("The Strange Case of ").addStyle(normal));
10  p.add(new Text("Dr. Jekyll").addStyle(code));
11  p.add(new Text(" and ").addStyle(normal));
12  p.add(new Text("Mr. Hyde").addStyle(code));
13  p.add(new Text(".").addStyle(normal));
14  document.add(p);
```

In line 1-3, we define a `normal` style; in line 4-7, we define a `code` style –Courier is often used when introducing code snippets in text. In line 8-13, we compose a `Paragraph` using different `Text` objects. We set the style of each of these `Text` objects to either `normal`, or `code`.

The `Style` object is a subclass of the abstract `ElementPropertyContainer` class, which is the superclass of all the building blocks we are going to discuss in the next handful of chapters. It contains a number of setters and getters for numerous properties such as fonts, colors, borders, dimensions, and positions. You can use the `addStyle()` method on every `AbstractElement` subclass to set these properties in one go.

Being able to combine different properties in one class, is one of the many new features in iText 7 that can save you many lines of code when compared to iText 5.

The `Style` class is about much more than fonts. You can even use it to define padding and margin values for `BlockElement` building blocks. But let's not get ahead of ourselves, the `BlockElement` class will be discussed in chapters 4 and 5.

Summary

In this chapter, we've introduced the `PdfFont` class and we talked about font programs, embedding fonts and using different encodings. This allowed us to show the title of a short story by Robert Louis Stevenson in different languages: English, Czech, Russian, and Korean. We also looked at font properties such as font size, font color, and rendering mode. We even discovered how to mimic styles in case we can't find the font program to render text in italic or bold.

There's much more that could be said about fonts, but we'll leave that for a separate tutorial. In the next chapter, we'll create a PDF with the full story while we discuss the `RootElement` implementations `Document` and `Canvas`.