WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

Bachelor Thesis

English title of the Bachelor Thesis	ooRexx and the Apache PDFBox Library – Nutshell Examples for Managing a PDF File
German title of the Bachelor Thesis	ooRexx und die Apache PDFBox-Library – Nutshell-Beispiele für die Verwaltung von PDF-Dateien
Author last name, first name(s)	Dobrea, Cristina Nicoleta
Student ID number	0652377
Degree program	J 033 561
Examiner degree, first name(s), last name	ao. Univ.Prof. Mag. Dr. Rony G. Flatscher

I hereby declare that

1. I have written this Bachelor thesis independently and without the aid of unfair or unauthorized resources. Whenever content was taken directly or indirectly from other sources, this has been indicated and the source referenced.

2. this Bachelor thesis has neither previously been presented for assessment, nor has it been published.

3. this Bachelor thesis is identical with the assessed thesis and the thesis which has been submitted in electronic form.

4. (only applicable if the thesis was written by more than one author): this Bachelor thesis was written together with first name(s), last name(s). The individual contributions of each writer as well as the co-written passages have been indicated.

Date _____ _____

Signature

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Bachelor Thesis

ooRexx and the Apache PDFBox Library – Nutshell Examples for Managing a PDF File

Summer Term 2012

Author:	**Cristina Nicoleta Dobrea**
Enrollment Number:	**0652377**
Examiner:	ao. Univ.Prof. Mag. Dr. **Rony G. Flatscher**

Abstract

This paper provides short examples for working with the Apache PDFBox library. Basic information about the structure of a PDF file is provided to ease understanding. The nutshell examples are written in ooRexx. The functionality of the Java library is imported using BSF4ooRexx.

Keywords

ooRexx, BSF4ooRexx, PDFBox, PDF, object oriented programming

Acknowledgements

Two people have helped and comfort me while writing the bachelor thesis and deserve my appreciation. I now take the opportunity to thank them for their involvement.

First of all, thank you Prof. Dr. Flatscher for encouraging me to take over this ambitions project. Also, thank you for keeping faith that I will successfully handle this venture.

Second, I would like to express my gratitude to a dear friend who reviewed my progress and helped me to overcome difficulties.

Table of Contents

Table of Source Code Listings

Table of Figures

Table Directory

1. Introduction

Developers will know that even though writing code is mainly about processing information, sharing the outcome is an important aspect of their activity. The electronic infrastructure for data exchange has become increasingly stable over the last two decades. On the other hand, growing differences between operating systems and other software hold the hazard of slowing down data circulation. As a consequence of this development, the Portable Document Format has gained great popularity. This file format is an effective solution to compatibility issues due to different hardware and software settings across computers.

While solutions for creating and reading PDF files are widely available, few people possess the knowledge and the ability to manage content and manipulate components of a PDF file. It is the challenge of this thesis to divulge basic knowledge about a PDF file's structure and provide a practical method for file administration.

To fulfill this challenging assignment, a whole set of nutshell examples illustrates how certain tasks can be achieved by using the PDFBox package. The language the programs are written in is ooRexx, extended by the BSF4ooRexx package for bridging the full Java functionality to ooRexx. In order to confer a strong awareness for the presented practice, some theoretical notions and backgrounds are explained in the initial chapters of this paper. First, the file format centric of this research, the Portable Document Format, is introduced. Technical details relevant to further proceedings are highlighted.

Next, all components involved in the creation of the examples are described in detail, starting with the basic principle of object oriented programming the languages obey. To permit the developer to re-enact the examples, instructions for installing all required components are provided. In addition, the environment used in creating the nutshell examples is displayed.

After consolidation of knowledge and requirements for working with ooRexx and the Apache PDFBox, the nutshells are presented as a solution to one scientific problem at a time. The nutshells are listed in increasing order of functionality and complexity. They are meant to be executed in the provided order, but can be carried out in arbitrary order.

Additional information is supplied whenever new classes and methods are implemented. The full source code has been inserted in the paper, since the programs represent the fulfillment of the research assignment.

The last section of this paper summarizes the added value provided by the implementation of the PDFBox and drafts conceivable use-case.

2. PDF – The Portable Document Format

One of the main problems that had to be solved in the digital area was the compatibility of documents across users, computers and operating systems. While sharing content became increasingly easy, displaying it in the original manner was a problem yet to be solved. In the early 1990s a multitude of competing file formats were developed to answer to this demand. Amongst them was the Portable Document Format (PDF) developed by 'Adobe Systems'.

First released in 1993, the Adobe solution for creating and reading PDF files was not free of charge. The second version, released one year later, allowed users to freely open and print out PDF files true to the original. The suitable application for creating PDF files remained chargeable until 2007. By 2008, the PDF became an open standard (ISO 32000:1:2008). Ever since, the Portable Document Format became the most popular file sharing format, independent of the hardware and the software used. [Wiki1]

Adobe published six editions of the program, continuously improving its functionality. Since the release of the open standard, three further extension levels have been made available by Adobe in collaboration with contributing experts.

2.1. The Acrobat Layers

On a technical level, every PDF file consists of multiple layers. Each layer of abstraction has its own independent set of rules. The lowest level of them all contains the raw data to be included in the document. A PDF file consists of objects of the following types: Boolean, numbers, stings, names, arrays, dictionaries - collections of objects indexed by names, streams and the null object. The second layer is the COS Layer. On this level, data is organized into a data tree. The Portable Document (PD) layer, also called PDModel aggregates the simple objects and organizes them into logical structure like paragraphs. In addition, useful intermediate level structures like Fonts and Images can now be implemented. [Parker]

There are, however more layers to a PDF file. The following table provides an accurate overview:

Package	Description
Acrobat_Color_Layer	AcroColor is an HFT that allows you to access the AcroColor engine (ACE), which controls color profile management for Adobe Acrobat. Plug-ins can import the AcroColor HFT to use the color management methods.
Acrobat_Forms_Layer	The Acrobat Forms plug-in exports its own Host Function Table (HFT), whose methods can be used by other plug-ins.

AS_Layer	The Acrobat Support (AS) layer of the core API provides a variety of utility methods, including platform-independent memory allocation and fixed-point math utilities. In addition, it allows plug-ins to replace low-level file system routines used by Acrobat (including read, write, reopen, remove file, rename file, and other directory operations). This enables Acrobat to be used with other file systems, such as on-line systems.
AV_Layer	The Acrobat Viewer (AV) layer of the core API (also known as AcroView) allows plug-ins to control Acrobat and modify its user interface. Using the AV methods, a plug-in can add menus and menu items, add buttons to the toolbar, open and close files, display simple dialog boxes, and perform many other application-level tasks. Plug-ins must use AV layer methods to be accessible through the Acrobat viewer's user interface.
COS_Layer	The Cos layer provides access to the low-level object types and file structure used in PDF files. PDF documents are trees of Cos objects. Cos objects represent document components such as bookmarks, pages, fonts, and annotations, as described in Section 3.6, "Document Structure", in the PDF Reference.
Digital_Signatures	Digital signatures allow a person to attest to something about a document by signing their name to it. An Acrobat signature in a document is bound to that document in such a way that altering the signed document or moving the signature to a different document invalidates the signature.
Error_Messages	
PDFEdit_Layer	The PDFEdit API provides easy access to PDF page contents. With PDFEdit, your plug-in can treat a page's contents as a list of objects rather than manipulating the content streams marking operators.
PDF_Library	
PD_Layer	The Portable Document (PD) layer of the core API (also called PDModel) is a collection of object methods enabling plug-ins to access and manipulate most data in a PDF file.
PDSEdit_Layer	PDF files are well known for representing the physical layout of a document; that is, the page markings that comprise the page contents. In addition, PDF versions 1.3 and later provide a mechanism for describing logical structure in PDF files. This includes information such as the organization of the document into chapters and sections, as well as figures, tables, and footnotes.

Table 1: All Acrobat Layers [Adobe1]

The PDModel is the most important layer to the purpose of this paper. The upcoming nutshell examples aim to access and manipulate the data of a PDF file. The way of operating objects will be explained in detail later in this paper.

2.2. Document Structure

A PDF file can be regarded as a data tree. Objects within the file constitute the tree nodes. Relationships between the node items are captured in dictionaries.

The root element of the document structure tree is called the document catalog. “The catalog contains references to other objects defining the document’s contents, outline, article threads, named destinations, and other attributes. In addition, it contains information about how the document should be displayed on the screen, such as whether its outline and thumbnail page images should be displayed automatically and whether some location other than the first page should be shown when the document is opened” [PDF Reference, page 112]. The figure below illustrates the entries of the catalog dictionary and the hierarchical relationships between them.

One of the dictionaries of the document catalog is the ‘Page Tree’ dictionary, which contains all of the document page objects. “The leaves of the page tree are page objects, each of which is a dictionary specifying the attributes of a single page of the document” [PDF Reference, page 119].

How to access certain node items of the document catalog will be shown in the nutshell examples of this paper. Further details are provided when dealing with the dictionaries and objects in question.

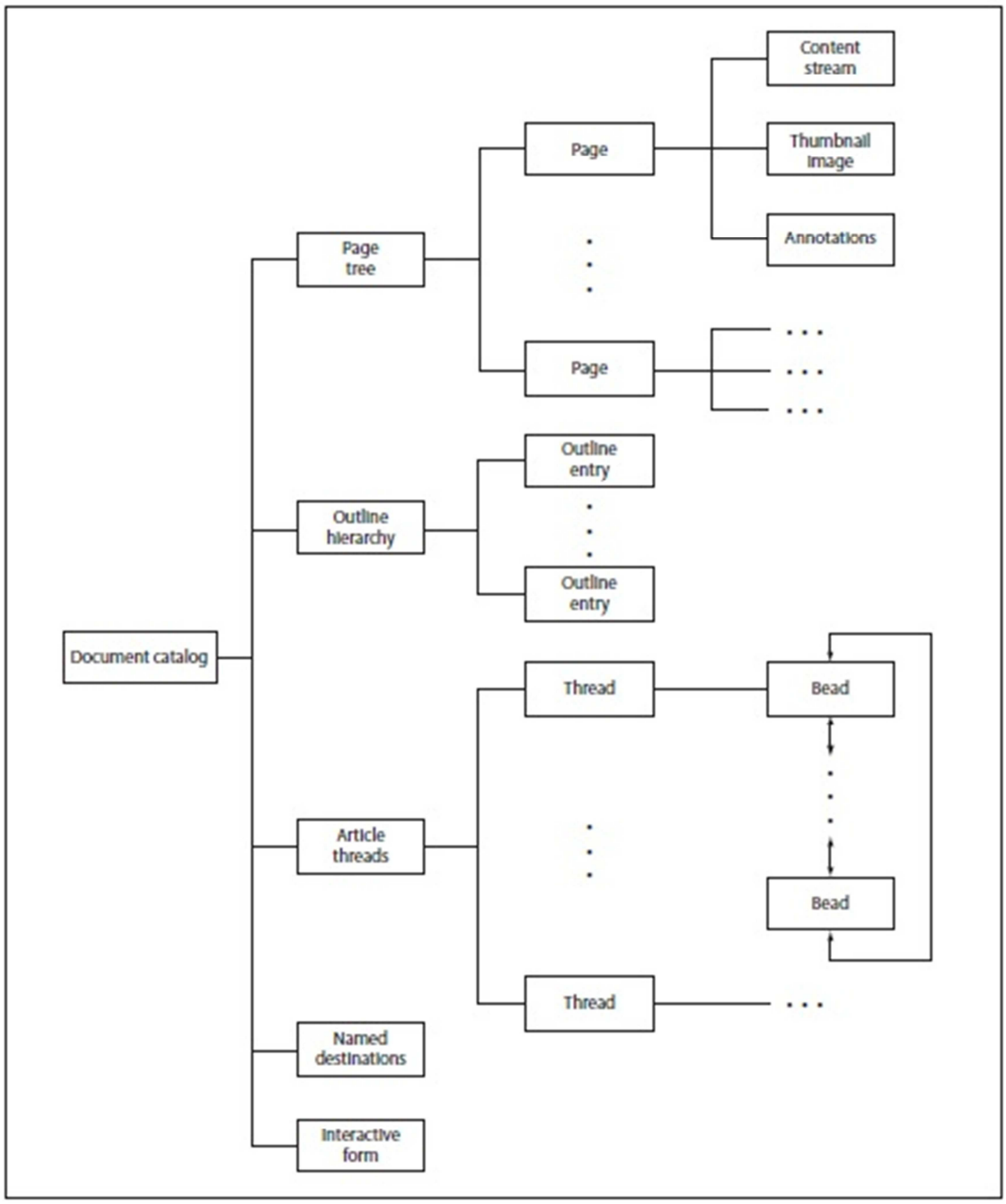

Figure 1: Structure of a PDF Document [PDF Reference, Page 113]

3. Involved Components

Several components have been used in generating the nutshell examples presented later on in this paper. At the very top, the Apache PDFBox Library is used for working with PDF files. Since the Apache PDFBox Library is a Java tool, the Bean Scripting Framework for Open Object Rexx is needed to enable developer's access to the Java library from ooRexx.

The following section will provide a description of the relevant components mentioned above.

3.1. The Apache PDFBox Library

Adobe still holds the patent for the Portable Document Format, but permits the development of complying software under a royalty-free license [Adobe2]. One of the applications leant on the PDF standard is the PDFBox, licensed under the Apache License v2.0. The PDFBox is an open source Java library allowing access to the inner layers of a PDF file and facilitating creation, manipulation and text extraction of the document and its components, to name just a few. [PDFBox1]

The convenience of using the PDFBox rises out of the embedding of PDF files in self-developed software. Since the main purpose of developing software is the processing of information, the outcome can be captured in a PDF document and made available for other users independent of used applications, software and hardware. The PDFBox also allows importing content from a PDF file for further processing.

Following the principles of the object oriented paradigm, the PDFBox library is a collection of classes with inbuilt methods for creating or accessing every item matching the PDF file format. Documentation for all available classes is published at the PDFBox site at the Apache webpage. Packages of the application programming interface (API) are split up according to features they possess. Inside these packages, classes are hierarchically ordered and subclasses inherit the methods of parent classes. The packages are in compliance with other Java libraries and do not require a complex installation procedure.

3.2. Involved Languages

The first part of this chapter describes the two programming languages used in the creation of the examples. Java, as well as ooRexx, is an object oriented programming language. Since this paper presents an introduction to using ooRexx and BSF4ooRexx, the author does not expect the reader to possess in depth knowledge of the components involved. Before shortly presenting the two mentioned languages, a short introduction in the principles of object oriented programming (OOP) is meant to create a fundamental understanding for the sections to come.

3.2.1. The Principles of Object Oriented Programming

There are four fundamental styles of programming languages, called paradigms, one of which is the object oriented paradigm. The main idea behind OOP is the usage of objects, characterized by its behavior and capable of interacting and carrying out tasks. [Husband et al.]

This principle is similar to the “human interaction with real world phenomena”. Objects are the instances of classes. A class represents a concept and defines the proprieties and functions of its appending objects. It can be used as a template for creating objects. Since objects of a class have many similarities, the functionality can be easily manipulated by changing the class rather than changing each individual object. [Nørmark]

Thinking of the real world, every item that surrounds us can be thought of as an object. Objects can be categorized into groups according to their characteristics. For example, think of a hand as a general concept. This general concept represents a ‘class’. The hand has its own individualities like the form and the set of functions it can fulfill. The human body has two objects of the type hand. In addition to the general functionality of any hand, a particular hand object, be it the right hand or the left one, has additional or slightly changed properties. [Nirosh]

Old procedural, non-object oriented programming mainly consists of a list of commands to be carried out. Items are created manually and properties are assigned individually. Increased complexity of the application typically leads to vast program size and increased vulnerability of the code. Since every piece of the code can be modified, it represents a potential source of errors. In contrast, in OOP, “data as well as operations are encapsulated in objects. Information hiding is used to protect internal properties of an object. Objects interact by means of message passing”. [Nørmark]

Other characteristics of OOP are:

“Classes are organized in inheritance hierarchies”. Each class is an extension or specialization of the preceding class, while the object inherits properties from all parent classes. To stick with the example of human body parts, the class ‘finger’ is a subclass of ‘hand’, which is also a subclass of ‘human body part’. As a result, the class ‘finger’ inherits the proprieties of both, ‘hand’ and ‘human body part’. Inheriting functionality from other classes substantially eases the effort of programming and allows defining a fine-grained set of specialized functions for every subclass. [Nørmark]

The object-oriented paradigm has gained great popularity in recent times and many programming languages now support OOP.

3.2.2. Java

The Java programming language has been developed by James Gosling and released by Sun Microsystems – which eventually became the Oracle Corporation – by 1995. Java follows the object oriented paradigm of programming. Since 2007, most of the Java technologies were published under the GNU General Public License.

Developers tried to implement as few platform dependencies as possible. This would guarantee that a code, once written, would run on other machines and not fail due to different operating systems. James Gosling summarized this principle as “WORA: write once, run anywhere”.

Other principles lying at the very bottom of Java are:

- ‘It should be "simple, object-oriented and familiar"
- It should be "robust and secure"
- It should be "architecture-neutral and portable"
- It should execute with "high performance"
- It should be "interpreted, threaded, and dynamic"’ [Wiki2]

The current version is the Java Standard Edition 7, released in 2011, extended by a number of updates. Recent statistics show that, even though Java has lost leadership among most used programming languages, it is still far away from fading. [Tiobe]

3.2.3. ooRexx

“Rexx is a procedural programming language that allows programs and algorithms to be written in a clear and structured way. It is easy to use by experts and casual users alike. Rexx has been designed to make easy the manipulation of the kinds of symbolic objects that people normally deal with such as words and numbers. Although Rexx has the capability to issue commands to its host environment and to call programs and functions written in other languages, Rexx is also designed to be independent of its supporting system software when such commands are kept to a minimum.” [RexxLA]

The birth year of the Restructured Extended Executor (Rexx) language interpreter is known to be 1979, when the IBM employee Mike F. Cowlishaw laid the fundament of a "human centric language". Several years later, Rexx became the official strategic procedural (scripting) language of IBM's operating systems. [Flatscher]

In the late 1980’s, work began for developing an object oriented version of Rexx, fulfilling the programming needs of the time. It was only 1997 that the commercial available successor of Rexx, the object Rexx was released. By the mid 2000’s, IBM handed the source code for Rexx and object Rexx to the ‘Rexx Language Association’. Rexx and its successor became available as open source software. In 2005, the Object Oriented Rexx was released. [Flatscher]

“Open Object Rexx (ooRexx) is an Open Source project providing a free implementation of Object Rexx. ooRexx is distributed under Common Public License (CPL) v1.0. Object Rexx is an enhancement of *classic* Rexx; a powerful, full-featured programming language which has a *human-oriented* syntax. The Open Object Rexx interpreter allows you to write programs procedurally as well as in an object-oriented fashion. Its main benefits include:

- Easy to use and easy to learn
- Upwardly compatible with *classic* Rexx
- The ability to issue commands to multiple environments
- Offers powerful functions
- Based on English-like commands
- Enhanced with full object orientation
- Designed for object-oriented programming, and also allows Rexx conventional programming
- Provides a standard Rexx API to develop external function libraries written in C” [ooRexx1]

"Open Object Rexx includes features typical of an object-oriented language, such as subclassing, polymorphism, and data encapsulation. It is an extension of the *classic* Rexx language, which has been expanded to include classes (a base set of classes is supplied), objects, and methods. These extensions do not replace *classic* Rexx functions or preclude the development or running of *classic* Rexx programs.

Open Object Rexx is fully compatible with earlier versions of IBM REXX Interpreters that were not object-oriented and compatible with other Open Source Rexx interpreters currently available." [ooRexx1]

3.2.4. BSF4ooRexx

As mentioned before, the Apache PDFBox is a Java library. The first challenge when managing a PDF file by using the ooRexx programming language is establishing a connection to Java objects and allowing implementation of Java methods. For means of addressing Java classes and objects, the Bean Scripting Framework for object oriented Rexx (BSF4ooRexx) has been developed as an extension for ooRexx. This library allows bridging full functionality of the Java Runtime Environment to the human-centric programming language ooRexx without requiring knowledge of the Java programming language.

Whenever the BSF class is used to camouflage Java as ooRexx, the fully qualified Java class name has to be provided. Moreover, when importing a Java class, use of the exact case is mandatory. Afterwards, the imported Java class can be used in the code as any regular ooRexx class.

Objects created from Java classes can be addressed by supplying the correct method name and arguments, as given in the Java documentation. However, the syntax will be written in ooRexx, ensuring less programming effort compared to programming in Java.

Using `BSF.cls`, the developer can create Java objects and operate with the build-in arguments and functions. BSF needs fully qualified Java class names. Note that the use of the exact case is mandatory. To bridge any Java class to ooRexx, the user can chose between two different options regarding the syntax:

For one, the statement `'name=BFS.import("className")'` at the beginning of the source code imports the class without instantiating an object. This procedure is recommended whenever multiple objects of the same class needs to be created, since it avoids redundancy of repeatedly importing the same class for multiple objects. To create an

```
--import Java class
importdoc = BSF.import("org.apache.pdfbox.pdmodel.PDDocument")

--create class object
doc = importdoc~new()

--supply class BSF
::requires BSF.cls
```

affiliated object, simply use the syntax ‘object1=name~new’.

Listing 1: Using the BSF class – option1

Another option for importing Java classes is importing a class and creating an object at once. This action is performed by the statement ‘object2=.bsf~new(“className”)’. This procedure is practical since the importing of the classes and the creating of object are tied together, increasing transparency and preventing extensive length of the application code.

```
--import and instanciate class
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")

--supply BSF class
::requires BSF.cls
```

Listing 2: Using the BSF class – option 2

By now, regardless of the procedure chosen, the class has been imported and am object created. Class names for the PDFBox are available in the PDFBox documentation and can be looked up on the Apache PDFBox Website. To handle the object, implement the methods provided in the documentation. The methods listed in the documentation are typically displayed as ‘method(argument)’. In contrast to Java, ooRexx does not call for the brakes after the method name if no argument is provided.

Also, for loading a class in order to access the class methods, the following syntax is adequate: ‘object3=BSF.loadClass(“className”)~method’

```
--import class to retrieve methods
doc=BSF.loadClass("org.apache.pdfbox.pdmodel.PDDocument")~save

--supply BSF class
::requires BSF.cls
```

Listing 3: Using the BSF class – option 3

Last, at the end of the code, a ‘::requires’ directive is used to address the BSF.cls class.

The BSF4ooRexx project was registered at SourceForge.net and is available free of charge.

4. Installation Guide

To help with the application of the required programs, the following section supplies instructions for correct installation of all the involved components as they have been described above.

4.1. ooRexx

For download, the ooRexx.net website will redirect the user to the SourceForge.net site of ooRexx. Choose the package which description is closest to your operating system and environment. Download the file and follow the installation procedure, as it is self-explanatory.

4.2. BSF4ooRexx

For download of the ooRexx extension for addressing Java classes and objects, go to the BSF4ooRexx homepage at SourceForge.net and download the latest version of the package, available for ooRexx version 4.1.0 or higher. Download and unzip the archive and choose the installation according to your operating system. A new CLASSPATH variable should now have been added to the environment variables.

4.3. Apache PDFBox

Installation of the Apache PDFBox is quite simple and not at all time consuming, provided the right package is being retrieved. The Apache PDFBox is an open source project. This means that the project is and will remain available free of charge.

To install the PDFBox library, visit the PDFBox homepage and navigate to the download section. The newest available version is automatically suggested for download. If the user desires to load a different version, a link will lead to all previous releases.

For means of quick installation, download the pre-build standalone binary version provided as a .jar file. This standalone version contains all classes of the PDFBox. Check the integrity of the download by verifying the signature and the checksums, as recommended in the download section of the website.

After download and verification, add the file path to the CLASSPATH system variable of the environment variable. The PDFBox library is now ready for use.

5. Used Environment

The following operating system was running on the computer involved in creating the nutshell examples:

Windows 7 Enterprise Edition with Service Pack 1, 64-bit system.

The ooRexx version 4.1.0.6441 published by the Rexx Language Association has been used for creating the examples of this paper.

It has been extended by the BSF4ooRexx package 4.10 as released in June 2012.

Also, the Java version running on the machine the examples have been created on is 7.0.40 – 32 bit.

For creating the nutshell examples, the Apache PDFBox version 1.7.0. has been installed. By the end of July 2012, a newer version - PDFBox 1.7.1.- has been released. Although creating the code was accomplished by using the older library, full compatibility with the new release has been tested.

In addition, the VIM editor version 7.3 has been used for writing the code. Version 7.1 or higher supports the ooRexx syntax.

6. Managing a PDF File – Nutshell Examples

In this chapter, the actual source code for handling a PDF file by using the Apache PDFBox with ooRexx and BSF4ooRexx is presented. Eleven nutshell examples have been created to illustrate accomplishment of diverse tasks. The complete source code has been added to each section, since it represents the completion of the assignment. Whenever needed, additional information on classes and objects are provided, same as definitions of PDF file components.

6.1. Create a New Document and Add Text

The first example illustrates how a new writeable document is created. In order to accomplish this task, some simple steps need to be taken as demonstrated below.

```
--create a new file and add a blank page
doc = .bsf~new("org.apache.pdfbox.pdmodel.PDDocument")
page = .bsf~new("org.apache.pdfbox.pdmodel.PDPage")
doc~addPage(page)

--create a content stream to hold data
contentStream = .bsf~new("org.apache.pdfbox.pdmodel.edit.PDPageContentStream", doc, page)

--create font object
font = BSF.loadClass("org.apache.pdfbox.pdmodel.font.PDType1Font")~HELVETICA_BOLD

--define content stream
contentStream~beginText
contentStream~setFont(font, 12)
contentStream~moveTextPositionByAmount( 100, 700 )
contentStream~drawString("This is page ONE of ONE")
contentStream~endText
contentStream~close

--close and save the file
doc~save(doc.pdf)
doc~close

::requires BSF.cls
```

Listing 4: PDF file creation and text insertion

In the current example, four Java classes have been imported.

The PDDocument class (fully qualified Java class name:

`org.apache.pdfbox.pdmodel.PDDocument)` is an extension of the `java.lang.Object` class. “This class is the in-memory representation of the PDF document.” PDDocument is used to create a new PDF file. For the document to be valid, at least one page needs to be added. This can be accomplished by using the PDPage class (fully qualified Java class name: `org.apache.pdfbox.pdmodel.PDPage`), another extension of the `java.lang.Object` class. “This represents a single page in a PDF document“. After creating a new file and adding a blank page, the document needs to be saved and closed.

If adding text to the document is desired, two other classes need to be implemented. The first one, PDPageContentStream, creates the data stream for writing to the page object.

“A content stream is a PDF stream object whose data consists of a sequence of instructions […]. The instructions are represented in the form of PDF objects, using the same object syntax as in the rest of the PDF document. However, whereas the document as a whole is a static, random-access data structure, the objects in the content stream are intended to be interpreted and acted upon sequentially. Each page of a document is represented by one or more content streams” [PDF Reference, Page 126]. It is mandatory to close the object after finishing text operations, using the `~close` method.

The PDPageContentStream object (`org.apache.pdfbox.pdmodel.edit.PDPageContentStream`) needs specification of two parameters: the document and the page of the document it will write to.

Additionally, the following methods need to be defined:
`~beginText` - the beginning of the text operations
`~setFont(PDFont font, float fontSize)`- the PDFont class is discussed in the upcoming paragraph
`~moveTextPositionByAmount(float x, float y)`- specify the desired location for text insertion
`~drawString(String text)`- this will draw a string at the specified location
`~endText`-marks the ending of the text operation
`~close` -close the content stream

To create a new font object, the class PDFont or one of its subclasses is needed. In the current example, one of the PDF base fonts was selected (Helvetica Bold). The following listing shows the class hierarchy of PDType1Font:

```
java.lang.Object
      org.apache.pdfbox.pdmodel.font.PDFont
          org.apache.pdfbox.pdmodel.font.PDSimpleFont
              org.apache.pdfbox.pdmodel.font.PDType1Font
```

Table 2: Class hierarchy for PDType1Font [PDFBox PDType1Font]

"The PDF specification states that a standard set of 14 fonts will always be available when consuming PDF documents. In PDFBox these are defined as constants in the PDType1Font class." [PDFBox Standard 14 Fonts]

Standard Font
PDType1Font.TIMES_ROMAN
PDType1Font.TIMES_BOLD
PDType1Font.TIMES_ITALIC
PDType1Font.TIMES_BOLD_ITALIC
PDType1Font.HELVETICA
PDType1Font.HELVETICA_BOLD
PDType1Font.HELVETICA_OBLIQUE
PDType1Font.HELVETICA_BOLD_OBLIQUE
PDType1Font.COURIER
PDType1Font.COURIER_BOLD
PDType1Font.COURIER_OBLIQUE
PDType1Font.COURIER_BOLD_OBLIQUE
PDType1Font.SYMBOL
PDType1Font.ZAPF_DINGBATS

Table 3: Standard fonts available via PDType1Font class [PDFBox Standard 14 Fonts]

After completion of all of these steps, save the document by setting the desired file name as a string parameter and close the document.

References for this example:

[PDFBox PDDocument]
[PDFBox PDPage]
[PDF Box PDPageContentStream]
[PDFBox PDType1Font]
[PDFBox Standard 14 Fonts]

6.2. Add Image to PDF

The second example demonstrates how an image can be added to a PDF file.

```
--prepare new document
doc = .bsf~new("org.apache.pdfbox.pdmodel.PDDocument")
page = .bsf~new("org.apache.pdfbox.pdmodel.PDPage")
doc~addPage(page)

--prepare image
input=.bsf~new("java.io.FileInputStream", "image1.jpg")
image = .bsf~new("org.apache.pdfbox.pdmodel.graphics.xobject.PDJpeg", doc, input)

--define content stream
stream = .bsf~new("org.apache.pdfbox.pdmodel.edit.PDPageContentStream", doc, page)
stream~drawImage(image,55,80)
stream~close

--save and close file
doc~save("ImageNowPdf.pdf")
doc~close

::requires bsf.cls
```

Listing 5: Image insertion

First, a new document is created and a blank page is inserted. This action is the same as described in detail for the previous nutshell example. Second, the content stream presented before is created.

Since the data to be written in the file is a stream of bytes in contrast to characters as was the case in the previous example, the PDPageContentStream is not sufficient to handle data extraction from a file and embed it into another. To solve this issue, the Java FileInputStream has been used in the current exemplification.

```
java.lang.Object
       java.io.InputStream
              java.io.FileInputStream
```

Table 4: Class hierarchy for FileInputStream [Java FileInputStream]

"A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment. FileInputStream is meant for reading streams of raw bytes such as image data" [Java FileInputStream]. By setting the image's file name as a parameter of the FileInputStream object, a connection to the file is established.

Next, an object of the PDJpeg class (org.apache.pdfbox.pdmodel.graphics.xobject.PDJpeg), an image class for Jpeg files, is used to import the image.

"An external object (XObject) is an object defined outside the content stream and referenced as a named resource. The interpretation of an XObject depends on its type. An image XObject defines a rectangular array of color samples to be painted; a form XObject is an entire content stream to be treated as a single graphics object" [PDF Reference, Page 165].

The parameters used with the PDJpeg object are the destination document of the image and the stream containing the Jpeg data – in our case the FileInputStream.

After preparation of the image by following the described steps, the image can now be added to the ContentStream and drawn at the specified coordinates. To draw the image according to the default size, the method `~drawImage(PDXObjectImage image, float x, float y)` has been implemented. To modify the dimensions of the image, use the method `~drawXObject(PDXObject xobject, float x, float y, float width, float height).`

Close the content stream, then save and close the file according to the previous example after completing the operation.

References for this example:

[PDFBox PDDocument]
[PDFBox PDPage]
[PDF Box PDPageContentStream]
[PDFBox PDJpeg]
[Java FileInputStream]

6.3. Merge PDF Documents

This nutshell example demonstrates a simple, jet very effective way to merge two or more PDF files. In preparation for the example, two new documents are created by running the first nutshell example presented. ‘doc1.pdf’ contains the text “This is page ONE of TWO”, and, accordingly, ‘doc2.pdf’ contains the text “This is page TWO of TWO”. This is relevant since we will use the created documents in further examples.

```
--add source documents to merge
merger=.bsf~new("org.apache.pdfbox.util.PDFMergerUtility")
merger~addSource("doc1.pdf")
merger~addSource("doc2.pdf")

--set destination file name and perform operation
merger~setDestinationFileName("newdoc.pdf")
merger~mergeDocuments

::requires BSF.cls
```

Listing 6: Merging documents

The class PDFMergerUtility (fully qualified Java name: org.apache.pdfbox.util.PDFMergerUtility) is capable of receiving a list of documents and merge them, saving the result in a new document. In the example shown above, the two documents’ names are the arguments of the method `~addSource`, while the new document is specified as a parameter of `~setDestinationFileName`.

Another way to use the class PDFMergerUtility is by instantiating the merger object and using the method `~appendDocument (PDDocument destination, PDDocument source)`. This way, the content of one document will be appended to another one without changing the destination file.

References for this example:

[PDFBox PDFMergerUtility]

6.4. Extract Text from PDF File

This example demonstrated how text can be extracted from an existing PDF file. Since the deployed class is not entirely compatible with the ooRexx stream object for writing into destination files, the objective of this example is performed through two different approaches: First, the ooRexx stream object is deployed, and then the stream classes of Java are used.

```
--load the document
source="newdoc.pdf"
importdoc=BSF.loadClass("org.apache.pdfbox.pdmodel.PDDocument")
doc=importdoc~load(source)

--get content
stripper=.bsf~new("org.apache.pdfbox.util.PDFTextStripper")
msge=stripper~getText(doc)

--print out content
say "content of" source ":"
say msge

--create and write content to destination file
dest="newdoc.txt"
output=.stream~new(dest)
output~lineout(msge)
say "content saved in " dest
call syssleep 5

::requires BSF.cls
```

Listing 7: Text extraction - ooRexx stream object

The source file for this example is the previously created “newdoc.pdf”. The text content of the document is spread on two separate pages. To get access to the document’s content, the class PDFTextStripper si implemented. Using this class allows importing the text from any PDF file, regardless of its formatting and page arrangement.

To simply import the text to ooRexx, the method `~getText(PDDocument doc)` is implemented. After doing so, the character string is sent to the ooRexx stream object to write in the destination file. For illustration purpose, the complete text is printed out in the Command Prompt window and inserted into a new TXT file.

The current example is designed to use ooRexx’ own stream object. Therefore the full functionality of the PDFTextStripper class is not available due to the absence of a Java

output writer as requested for some methods' argument. For example, the method `~writeText(PDDocument doc, Writer outputStream)` demands for the java.io subclass OutputStreamWriter. Delivering the ooRexx stream object as an argument results into an error, pointing out an invalid method argument.

To bypass this problem, we loaded the PDF file's content to the main program as an intermediary step and then proceeded to printing it out to the destination file.

However, another option and a way to access full functionality of the **`org.apache.pdfbox.util.PDFTextStripper`** class is the usage of Java stream classes as shown below:

```
--load the source document and create the output stream
source="newdoc.pdf"
importdoc=BSF.loadClass("org.apache.pdfbox.pdmodel.PDDocument")
doc=importdoc~load(source)
dest="newdoc2.txt"
outputstream=.bsf~new("java.io.FileOutputStream", dest)

--get content and write it to the output stream
stripper=.bsf~new("org.apache.pdfbox.util.PDFTextStripper")
outputwr=.bsf~new("java.io.OutputStreamWriter", outputstream)

--create buffer
buffwr=.bsf~new("java.io.BufferedWriter", outputwr)

--write from source document to destination file
stripper~writeText(doc, buffwr)
buffwr~close

::requires BSF.cls
```

Listing 8: Text extraction - Java stream object

The following Java classes are used to ensure full functionality:

- `java.io.FileOutputStream` - establish connection to destination file
- `java.io.OutputStreamWriter` - writing to destination file through the previous created connection
- `java.io.BufferedWriter` - using a buffer for efficient writing (deployment optional, but recommended)

By importing these classes via BSF4ooRexx, writing directly from the source t to the destination file can be achieved, without any need for intermediary steps.

Which one of the presented practices will be used depends to some extent on the action to be performed, but is mainly up to the user and his preferences and precognition.

References for this example:

[PDFBox PDDocument]

[PDFBox PDFTextStripper]

[Java FileOutputStream]

[Java OutputStreamWriter]

[Java BufferedWriter]

[ooRexx, page 84 – Writing a text file]

6.5. Search for String in PDF File

In previous examples, we have demonstrated how text can be extracted out of a PDF File. The current example aims at searching for specific text within a PDF Document. For this purpose, the file 'doc2.pdf' is used as the source file. The file contains the string "This is document TWO of TWO" and was created by implementing the actions described in the first nutshell example. The assignment for the current example is to search for the word "TWO" inside the PDF document and return the number of times the string has been identified.

```
--import text from source document, get text
stripper=.bsf~new("org.apache.pdfbox.util.PDFTextStripper")
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")
stry = stripper~getText(doc~load("DOC2.pdf"))
say "Given string is: "
say stry

--define string to search for
strx="Two"
say "Searching for string: "
say strx

--define variable to memorize position of match
stamp=pos(strx,stry)

count=0 –set up the match counter
do while stamp > 0 --enter loop if match was found
    count=count+1 --increment counter
    say "Match number "count" found at position "stamp
    stamp=pos(strx,stry,stamp+length(strx))--define position to start next search
end

call syssleep 5
::requires BSF.cls
```

Listing 9: Searching for a string

The current example presents the search procedure by importing the text out of the PDF document and deploying some ooRexx built-in functions.

First of all, we need to import the character string from inside the source document. The procedure has already been discussed and the task accomplished by using the class PDFTextStripper. The sting obtained is labeled 'stry' for 'string Y'. The string we are searching for, in our case "TWO", is named 'strx' for 'string X'. Additionally, a counter is set up to detain the number of matches. This counter is labeled 'count' and initially set to 0.

Next, a stamp is defined to memorize the position of the match. To fulfill this task, the function `POS(needle,haystack)`is implemented.

"POS (Position) returns the position of one string, needle, in another, haystack. It returns 0 if needle is a null string or not found or if start is greater than the length of haystack. By default, the search starts at the first character of the receiving string (that is, the value of start is 1), and continues to the end of the string. You can override this by specifying start, the point at which the search starts, and length, the bounding limit for the search. If specified, start must be a positive whole number and length must be a non-negative whole number." [ooRexx2]

If the value of the created parameter is greater than 0, meaning a match has been found, a loop is entered for further processing of the initial string. First, the counter is incremented to store the number of times string X has been found. Second, to continue searching for string X inside string Y, the stamp parameter is modified to continue searching, starting at the position of the first letter of the previous match, plus the length of string X.

If, however, there is no need to retrieve the position where matches have been found, the ooRexx built-in function `COUNTSTR(needle,haystack)`can be applied. This function "returns a count of the occurrences of needle in haystack that do not overlap." Moreover, the string can be replaced by using the function `CHANGESTR(needle,haystack,newneedle)` can be implemented to achieve the desired outcome.
[ooRexx2]

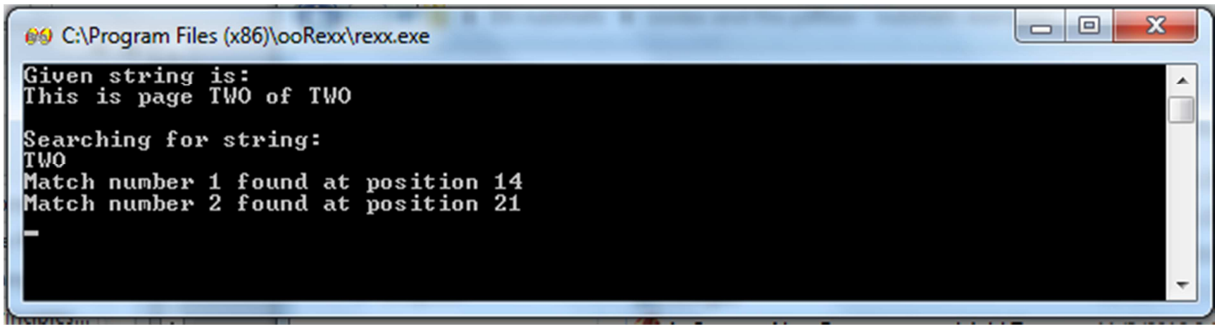

Figure 2: Searching for a string - outcome

Further development of the presented code can include features like highlighting of text sections by determining the coordinates of the section and drawing a colored box behind the text.

For the sake of completeness, the following annotation has to be discussed:

Importing the text from de PDF document brings about two extra characters. To be more precise, the string Y “This is page TWO of TWO” only consists of 23 letters. Jet 25 characters are imported, of which the last two are white-space characters and are not visually displayed in the Command Prompt interface.

References for this example:

[PDFBox PDDocument]

[PDFBox PDFTextStripper]

[oorexx2 – POS(position); countStr]

6.6. Split Up a PDF File by Handling Individual Pages

The following two examples illustrate how to split up a PDF document by using different approaches[1].

The current nutshell example is meant to split up a document into individual pages. Although this example does not involve deployment of any new classes, it shows how multiple pages of the same file can be appealed for further handling. This practice will be used whenever an action is performed to more than one pages of a PDF file, for instance adding a header or creating bookmarks (see example 'Creating new bookmarks').

```
--import source document
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")~load("newdoc.pdf")

--retrieve all pages
allpages = doc~getDocumentCatalog~getAllPages--returns List of PDPage objects

say "---"
say "OPTION 1 - Java ArrayList"
say "allpages:" allpages

a=doc~getNumberOfPages
a = a - 1
do i = 0 to a
   page = allpages~get(i)
   docx =.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")
   docx~addPage(page)
   name="split_01_"i"_JavaArrayList.pdf"
   say name
   docx~save(name)
end
say "---"

say "OPTION 2 - Java Array"
say "turning into a genuine Java array, that will be made into an ooRexx array:"
arr=allPages~toArray -- turn into an array

do i=1 to arr~items -- iterate like in ooRexx: first element has index # 1 !

   docx~addPage(arr[i])
   name="split_02_"i"_JavaArray.pdf"
   say name
   docx~save(name)
end
say "---"
call syssleep 5

::requires BSF.cls
```

Listing 10: Splitting up a PDF file: handling page objects

First, the source document is imported. This is the original file which will be separated into individual files. For this example, we use the document "newdoc.pdf" that has been created

[1] This example was completed with support provided by Prof. Rony G. Flatscher, to whom I am very thankful

in the previous example by merging two documents. By applying new measures, the action performed in the earlier example will be undone. Second, the document catalog is imported. For details on data content of the document catalog, please review the section "PDF – The Portable Document Format" of this paper.

Next, from the document catalog, enter the 'Page Tree' and retrieve all page objects available within the document. Note that the method `~getAllPages` returns a Java ArrayList holding PDPage objects.

It is important to be aware that the ArrayList currently storing information about nodes and pages is a Java type object. The main difference between collection classes in Java and in ooRexx is the index it starts with. In ooRexx, indexing of the collection class objects commence at 1, since this programming language was developed to be similar to the human communication and logic. However, indexing in Java starts at 0. This incompatibility can cause errors since the code is written in ooRexx and the index difference is easily overlooked.

Although getting the individual pages from the ArrayList for further handling is possible if considering the Java indexing scheme, an additional more convenient way of storing PDPage obejcts is presented in this example: turning the Java ArrayList into an Array which will be converted by BSF4ooRexx into an ooRexx Array. This method will help bypassing any errors connected to different index.

OPTION1: Java ArrayList

The first way of handling page objects is by working directly with the objects stored in the ArrayList. From the received page list, individual page objects can now be appealed for further processing by supplying the index value of the page in question, as shown below:

After retrieving the ArrayList containing the individual pages, a loop is built to create an individual file for each one of them. The loop counter is called 'a' and equals the number of pages of the document. Since the same action is applied to every page of the source document, the loop is set up to repeat itself 'a' times.

Although the number of pages is equal to the number of position in the Java array, the index difference needs to be considered. Page 1 of the document occupies the position at index 0 of the array. To obtain equality, 'a' and 'i' are matched by subtracting one unit from the counter 'a' (a = a - 1).

Now, the actual page handling can be performed. Every page object stored in the array is individually retrieved and assigned to a new document. Denotation and saving of each

document is itemized before repeating the chain of actions for the next PDPage object.

OPTION 2: Java Array

To avoid errors arising from differences in indexing, the method `~toArray` is applied to the ArrayList, resulting into a genuine Java array. The Java Array is automatically converted into an ooRexx-Array by BSF4ooRexx and therefore index of the collection class object commence at 1.

From here on, the procedure is similar to the one presented in the first option: a loop is set up to iterate for every item in the array. Inside the loop, every object is inserted into a new PDDocument object and saved under a different name.

Although both options return a valid outcome in terms of page handling and creation of separate documents for each page, the second option additionally provides a solution to issues arising out of index incompatibility between Java and ooRexx. The author therefor recommends deployment of the second alternative presented.

References for this example:

[PDFBox PDDocument]
[PDFBox PDDocumentCatalog]
[ooRexx Array]
[Java Array]
[Java ArrayList]

6.7. Split Up a PDF File by Sections

This example illustrates how a document containing multiple pages can be split up into several smaller documents by using the class Splitter.

```
--load the source document
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")~load("newdoc.pdf")

--set the splitting algorithm
splitter=.bsf~new("org.apache.pdfbox.util.Splitter")
count = 1
splitter~setSplitAtPage(count)

--create list of results, turn into array
doclist=splitter~split(doc)~toArray

say "Creating files:"
--create individual documents
do i=1 to doclist~items
      docx = doclist~at(i)
      name="split_03_"i"_JavaArray.pdf"
      say name
      docx~save(name)
end

call syssleep 3
::requires BSF.cls
```

Listing 11: Splitting up a PDF file: creating a splitting algorithm

The document to be split up is loaded by using the class PDDocument. Again, we use the document 'newdoc.pdf' that has been created before and consists of two pages containing the text "This is page ONE of TWO" and "This is page TWO of TWO".

The only additional PDFBox class needed to perform the splitting operation is `org.apache.pdfbox.util.Splitter`, a direct subclass of `java.lang.Object`. The `~setSplitAtPage(int split)` method helps defining an algorithm for splitting the document into several other document. Note that the `int split` argument is zero based. Since 'newdoc.pdf' only consists of two pages, the variable needs to be set at a value of 1.

Next, a list is created to store the individual, newly created documents until they are saved. Same as seen in the previous example, the list is corresponding to a Java ArrayList. By

turning the Java ArrayList into a genuine Java Array object and by using BSF4ooRexx, we are able to work with the Array as an ooRexx collection class object. Appealing the Array object can now be performed by implementing the ooRexx methods for the Array class. In addition, index of the array corresponds to the ooRexx notation scheme and is no longer zero-based.

The loop is similar to the one created in the previous example, except for the fact that entire document sections are saved as new files instead of inserting individual pages into newly created document. This circumstance eliminates the need of importing and instantiating the PDDocument class for creating the individual files.

Alternatively, the method `~createNewDocument` can be used directly with the splitting object to create a new document to write the contents to. The splitting feature of the PDFBox can be personalized by creating an arbitrary splitting algorithm to be applied to a document.

References for this example:

[PDFBox PDDocument]
[PDFBox Splitter]
[Java Array]
[Java ArrayList]

6.8. Create New Bookmark

The current example illustrates the creation of bookmarks on every page of a document. To do so, we proceed using the PDF file ‘newdoc.pdf’ containing two pages, that has been created before.

```
--load document
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")~load("newdoc.pdf")

--create new outline for document
outline=.bsf~new("org.apache.pdfbox.pdmodel.interactive.documentnavigation.outline.PDDocumentOutline")
doc~getDocumentCatalog~setDocumentOutline(outline)

--set root element for the tree
root=.bsf~new("org.apache.pdfbox.pdmodel.interactive.documentnavigation.outline.PDOutlineItem")
root~setTitle( "All Pages" )
outline~appendChild(root)

--get list of all PDPage objects
allpages = doc~getDocumentCatalog~getAllPages~toArray

--loop
do i = 1 to allpages~items
   --creating bookmarks and setting their destination
   page = allpages~at(i)
   destination=.bsf~new("org.apache.pdfbox.pdmodel.interactive.documentnavigation.destination.PDPageFitDestination")
   destination~setPage(page)
   bookmark=.bsf~new("org.apache.pdfbox.pdmodel.interactive.documentnavigation.outline.PDOutlineItem")
   bookmark~setDestination(page)

   --name the node and add to outline
   title="page" i+1
   bookmark~setTitle(title)
   root~appendChild(bookmark)
end

--open nodes for display
root~openNode
outline~openNode
doc~save("newdoc2.pdf")

::requires BSF.cls
```

Listing 12: Creating new bookmarks

As a logical first step the PDDocument object ‘newdoc.pdf’ is loaded. After doing so, the document catalog can be accessed. This can easily be done by sending the command `~getDocumentCatalog` to the PDDocument object. After importing the document catalog, the developer is able to set the document outline storing future bookmark items.

“A PDF document may optionally display a document outline on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of outline items (sometimes called bookmarks), which serve as a visual table of contents to display the document’s structure to the user.” [PDF Reference, Page 554]

Having gained access to the document’s catalog, an outline directory can now be added to facilitate the navigation of the document. This can be accomplished by creating an object from the class PDDocumentOutline, a class representing the document outline for PDF documents. A document outline is comparable to a data tree structure or hierarchy, displaying name and/or position of the document sections.

To set the root element of the newly added outline, a first item will be added to the bookmark tree. Creation of corresponding elements can be complete by using the PDOutlineItem class. This class allows not only creation of elements, but also permits navigation and manipulation of outline items, as will be shown in subsequent examples. After having created the root element labeled “All Pages”, the item can be added to the outline. This task is perform by using method `~appendChild(PDOutlineItem outlineNode).`

As the next step, we need to access the indivual pages of the document. The PDDocument class provides a method named `~getAllPages,` which returns the hierarchical structure of PDPageNode and PDPages, storing this objects in a Java ArrayList. As shown before, the ArrayList will be transformed into a Java Array, converted to an ooRexx array object by BSF4ooRexx.

To automate the creation of bookmarks for every page, a loop has been created. To find out how many bookmarks will be created, we use the command line `allpages~items`. This method returns the numbers of items stored in the array to be used as a parameter for the loop to be created.

Inside the loop, every PDPage object is invoked from its position on the array called “allpages”. The destination for the bookmark is set individually for every existing page of the document. The following list displays the class hierarchy for the class PDPageDestination and its known subclasses:

`java.lang.Object`
 	
`org.apache.pdfbox.pdmodel.interactive.documentnavigation.destination.PDDestination`
 	
`org.apache.pdfbox.pdmodel.interactive.documentnavigation.destination.PDPageDestination`

Direct known subclasses:
`PDPageFitDestination`
`PDPageFitHeightDestination`
`PDPageFitRectangleDestination`
`PDPageFitWidthDestination`
`PDPageXYZDestination`

Table 5: Class PDPageDestination and subclasses

After setting the destination, a new bookmark can be created pointing at it. Additionally, the title of the bookmark is customized to mirror the page it points to. As final step, the created

outline items have to be inserted into the document outline. Once again, we use the command `~appendChild` to pin the created items to the root element.

As a result of the performed actions, the edited document now comprises an outline containing the root element “All Pages” and two bookmarks pointing at the two PDPage objects. In order to display the bookmark nodes, the outline, as well as the root element, is opened using the method `~openNode`. We proceed to saving the document as “newdoc2.pdf”

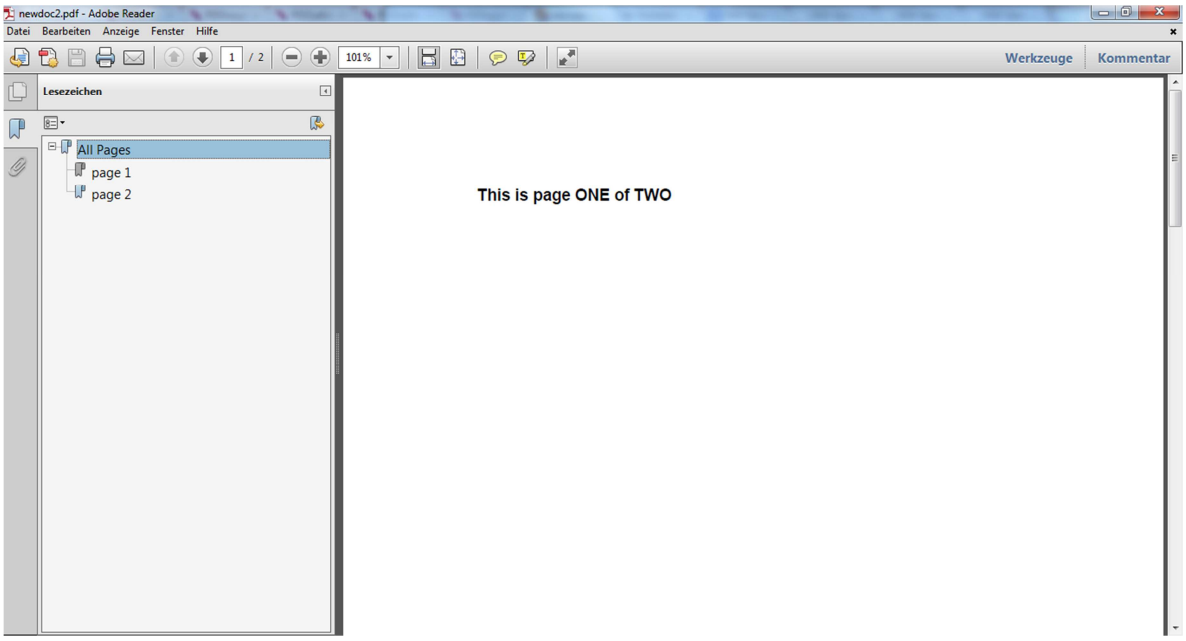

Figure 3: Creating new bookmarks - outcome

References for this example:

[PDFBox PDDocument]

[PDFBox PDDocumentCatalog]

[PDFBox PDDocumentOutline]

[PDFBox PDOutlineItem]

[PDFBox PDDestination]

[PDFBox PDPageFitDestination]

6.9. Get Bookmarks on a Single Level

This nutshell example describes the procedure for obtaining the bookmark elements for the upper level of the bookmark structure. Preceding the algorithm is possible in order to retrieve the entire outline structure of the document. However, the basic principle of entering and appending the nodes of the outline is hereby presented.

```
--load source document
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")~load("rexxpg.pdf")

--get catalog and outline
outline=doc~getDocumentCatalog~getDocumentOutline

--start navigation at first node
itm=outline~getFirstChild

--loop
a=0
do while a<outline~getOpenCount --find number of same level nodes
      say "Item= " itm~getTitle --extract node name
      itm=itm~getNextSibling --move to next node
      a=a+1
end

say "number of bookmarks on first level: "a

call syssleep 5
::requires BSF.cls
```

Listing 13: Extracting outline information: bookmarks on a single level

First we import a source document. To demonstrate the magnitude of this example, we use a different PDF file with a vast structure of bookmark elements. Accordingly, we employ the Open Object Rexx Programming Guide, named 'rexxpg.pdf'. This document can be found in the documentation folder for ooRexx. Given the current computer setting, the file was found in 'C:\Program Files (x86)\ooRexx\doc'. The programming guide is also available for download at http://www.oorexx.org/docs/rexxpg/rexxpg.pdf

The first step is importing the source file, as seen before. Next, the document catalog is imported for obtaining access to the PDF file's objects. From the document catalog, get the document outline. This is a directory containing all bookmark nodes of the document.

Interestingly enough, the only class that needs to be imported is the PDDocument class. By sending the message `~getDocumentCatalog` to the PDDocument object, a connection to

the PDDocumentCatalog class is established. Without importing this class, we are able to use the method `~getDocumentOutline` embedded in it. This is possible due to the basic characteristics of object oriented programming, where every object carries information about available methods. Once an object is imported, the user is able to handle it by using the class' methods.

After having acquired access to the document's outline, we can now appeal the first node element of the structure tree. We therefore select the first child of the outline by sending the message `~getFirstChild` to the PDOutlineNode object. The object returned is of the type PDOutlineItem – again, no import of this class is needed.

After connecting to the first node, a loop helps automating the next step for every other node of the same degree. As loop count, the counter 'a' is created. Using this approach not only helps determine the number of bookmark items of the level, but also eases exiting the loop after addressing each item. For the beginning, the counter is set to 0.

Next, the counter is compared to the number of bookmarks items of the first level. Therefore, we fetch the number of direct child elements of the outline by using the `~getOpenCount` method. Since we are examining the upper level of the document's outline, the nodes are likely to be open. Keep in mind that this is not necessarily true for other objects in the outline tree. In our example, there are 18 items to examine. The loop is set up to be traversed as long as the counter 'a' is smaller than 18.

Now, we retrieve the title of the bookmark item and move on to the next one. To do so, the methods `~getTitle` and `~getNextSibling` are used. After doing these two operations, the counter 'a' is augmented by 1 and the loop starts again, performing the same operations for another element.

The following screenshot displays the outcome of the program:

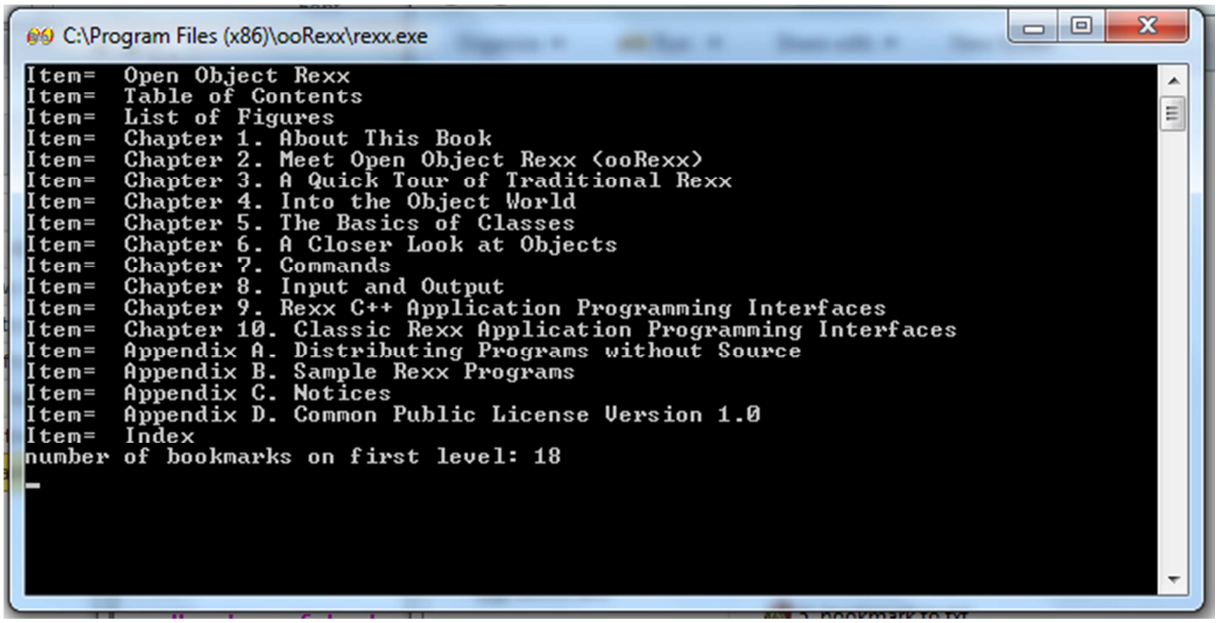

Figure 4: Extracting outline information: bookmarks on a single level - outcome

To continue collecting information about the outline tree, access child items of the once already inspected. Set up another loop by using the method `~getFirstChild` once again for every item and check if and how many items the branch of the outline tree holds. Move on to other items by using `~getNextSibling`.

The approach of this example is suited for a document with a limited number of outline tree levels, because querying the items count is necessary for every level of every branch. Another approach for getting the complete outline structure of the document is presented in the upcoming example.

References for this example:

[PDFBox PDDocument]

[PDFBox PDDocumentCatalog]

[PDFBox PDDocumentOutline]

6.10. Get the Complete Bookmark Structure of a PDF File

This example supplies a similar functionality as the previous one, displaying the bookmarks items of a PDF file's outline. For this example, however, a different approach was used. The previous example involved looking up the number of times the loops has to be traversed. We assumed that the document outline is limited in its complexity and all outline nodes are open. Although this procedure delivers a valid result, we are now using another approach to automate the search and display of bookmarks. Accordingly, the algorithm for navigation across an infinite tree structure is presented previous to its implementation in the code.

To simplify the understanding of the principle, we use the following bookmark structure to illustrate an ordered data tree:

1. Parent Item
   1.1. Child Item
      1.1.1. Grandchild Item
      1.1.2. Grandchild Item
      1.1.3. Grandchild Item
   1.2. Child Item
      1.2.1. Grandchild Item
      1.2.2. Grandchild Item

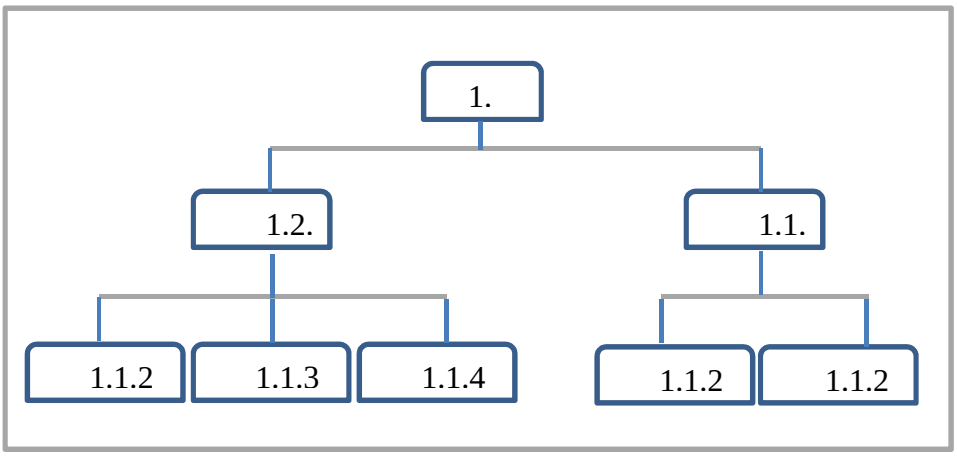

Figure 5: Graphical display of ordered data tree

Regarding the graphic above, we first need to figure out a way to access the nodes in the desired display order. Using the approach of the previous example, we could display all same level nodes at a time. Accessing another level and displaying it would lead up to the following outcome (unless manually adjusted and provided the nodes are open):

1. Parent Item
    1.1. Child Item
    1.2. Child Item
        1.1.1. Grandchild Item
        1.1.2. Grandchild Item
        1.1.3. Grandchild Item
        1.2.1. Grandchild Item
        1.2.2. Grandchild Item

This representation is not the desired outcome. Therefore, navigation of the tree needs to respect the following steps:

- Start at the root element
- Move to first child
- Continue moving down in rank by appending the first grandchild item, until there is no subsequent level left to navigate to
- On the lowest level, move to next same-ranked item
- Search for children of this item and, if available, repeat the steps above for as long as this is possible.

By now, we have queried the following nodes of the tree:

1. Parent Item
    1.1. Child Item
        1.1.1. Grandchild Item
        1.1.2. Grandchild Item
        1.1.3. Grandchild Item

Now we need to move on to node 1.2. To do so, we need to go up one level and search for another item of the same rank as 1.1. By following this step, we are now located on the 1.2. node. By repeating the same steps implemented from the beginning, we can now appeal inferior rank items of this branch. After completing this interrogation, we once again move to the upper level of “Child” items. Since there is no other node to apply these steps to and no node that has not been examine, the search for bookmark items is now complete.

Translating the presented search algorithm into code resulted into the following program:

```
--load document, enter outline tree and navigate to root item
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")~load("rexxpg.pdf")
root=doc~getDocumentCatalog~getDocumentOutline
itm=root~getFirstChild

--protection variables
lvl = 0
a = 1
q = 0
--create stream
outputobject = .stream~new("rexxpg.txt")

--loop
do while a \= 0
   if lvl = -1 then do
      a = 0
   end

   else do
      msge = ""—-insert new line to destination file
      do i = 0 to lvl
          msge = msge "->"
      end

      --write to stream
      msge = msge itm~getTitle--get node name
      outputobject~lineout(msge)--pass node name to stream

     --navigate down the outline tree
     if itm~getFirstChild \== .nil then do
        itm = itm~getFirstChild
        lvl=lvl+1
     end

     else do
         if itm~getNextSibling \== .nil then do
             itm = itm~getNextSibling
         end

         else do
             q = 1
             do while q = 1
                 if lvl = 0 then do
                     q = 0
                 end

                 else do
                 --navigare up the outline tree
                      if itm~getParent~getNextSibling \==.nil then do
                         q = 0
                      end

                      else do
                         itm = itm~getParent
                         lvl = lvl - 1
                      end
                 end
            end

            if itm~getParent \== .nil then do
                itm = itm~getParent
                lvl = lvl - 1
                    if lvl \= -1 then do
                        itm = itm~getNextSibling
                    end
                end

                else do
```

```
                    a = 1
                end
            end
        end
    end
end

::requires BSF.cls
```

Listing 14: Extracting outline information: the complete outline structure

To demonstrate the amplitude of the programs functionality, the Open Object Rexx Programming Guide, named 'rexxpg.pdf' is once again used as source file. The methods `~getDocumentCatalog` and `~getDocumentOutline` are cascaded to enter the outline tree.

In this example, ooRexx' own stream is implemented to write to a TXT file. This measure is deployed since the collection of nodes is too extensive to be depicted in the Command Prompt interface.

Some protection variables are used to keep track of the outline levels examined and help breaking the looping at some point. Accordingly, the counter 'lvl' is set up to memorize the current position on the tree. Since the displayed bookmark elements of the outline are not same-level as the root element, we set the counter 'lvl' to 0. As seen on the example data tree, we first navigate down the outline and slowly go back up in rank. Search is completed when the counter 'lvl' is equal to -1, since navigation as arrived back at the root element.

The 'itm' object is an object of the type PDOutlineItem. At the beginning, the root element is set to be named 'itm'. As we move from one node to another, 'itm' is the label for the node we are situated on at the time. To write the name of the node to the TXT file, we apply the `~getTitle` method to the item and pass the data on to the stream. To keep track of the items rank even after being written in the destination file, the symbol "->" is used for every level preceding the actual node rank. This means passing the symbol "->" times 'lvl' to the stream.

As demonstrated on the simple data tree at the beginning of this section, navigating down the tree using the method `~getFirstChild` is pursued for as long as this is possible. The statement "for as long as this is possible" is translated into object oriented programming code by comparing the sighted object to 'the Nil Object'. This is one of the basic primitive objects and contains no data. Arriving at the Nil Object means that there are no child elements left to inspect. The same technique is used when navigating up in rank or to related same-level objects.

To apply the procedure described at the top of this section, loops are built inside other loops.

Whenever navigation jumps down in level, the counter 'lvl' is increased by one. This event is undone by the time we travel back up on the precedent level. In addition to 'lvl', the protection parameters 'a' and 'q' are used to break the looping whenever the search algorithm leads to the Nil Object.

For every loop, there have been implemented two branches: one if the condition is fulfilled and one if it is not. The visual arrangement of the IF-ELSE pairs is meant to facilitate the reading of the code. After having gained knowledge of the principles applied, comprehension of the presented code should be easy.

The following screenshot displays a part of the retrieved results written into the destination TXT file.

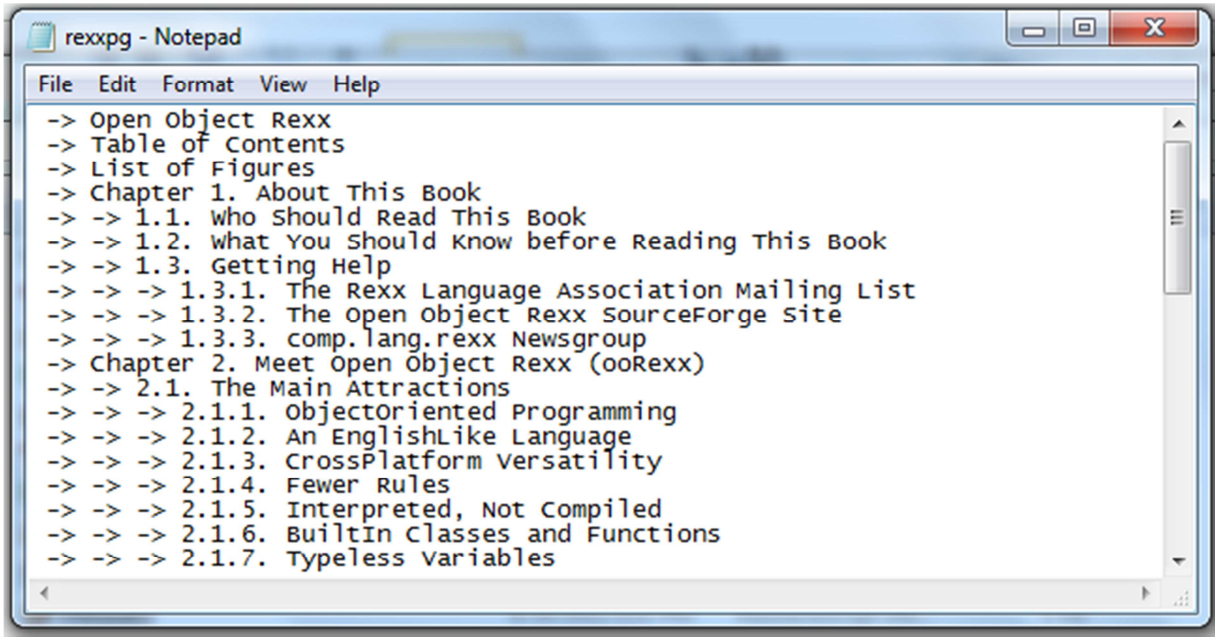

Figure 6: Extracting outline information: the complete outline structure – outcome (fragment)

By using this program, automated search for bookmark items can be accomplished for vast outline trees effortless, regardless of the number of items contained. Also, helpful features can be built on top of this program by connecting the bookmark item to the underlying page object and finding the location of the bookmark. Possible applications include extracting selected section of the document by setting the text extraction to only process pages between two chosen bookmarks, or searching for a string at a predefined section of the document.

References for this example:

[PDFBox PDDocument]

[PDFBox PDDocumentCatalog]

[PDFBox PDDocumentOutline]

[ooRexx, page 84 - Writing a Text File]

[ooRexx, page 59 - The NIL Object (.nil)]

6.11. Encrypt a PDF Document

The last example delivers the simplest method for securing a document by deploying encryption and equipping the file with special user permissions.

```
--create document, add one page
doc = .bsf~new("org.apache.pdfbox.pdmodel.PDDocument")
page = .bsf~new("org.apache.pdfbox.pdmodel.PDPage")
doc~addPage(page)

--set access permissions
ap =.bsf~new("org.apache.pdfbox.pdmodel.encryption.AccessPermission")
ap~setCanExtractContent(.false)
ap~setCanPrint(.false)
ap~setCanModify(.false)

--set access passwords
ownerpass = "master"
userpass = "user"

--apply policy to document
stnd = .bsf~new("org.apache.pdfbox.pdmodel.encryption.StandardProtectionPolicy",ownerpass, userpass, ap)
stnd~setEncryptionKeyLength(128)
doc~protect(stnd)
doc~save("encryted.pdf")
doc~close

::requires BSF.cls
```

Listing 15: Encrypting a PDF document

For demonstration purpose, a new blank file is created. Providing customized access permissions for a PDF document can be accomplished by following a two-step procedure: defining security features and applying the security policy to the document.

A standard configured PDF file has no special access permissions and is vulnerable to changes and interference. To secure a document, first the desired security features need to be defined. Therefore, several objects are instantiated from the class **`org.apache.pdfbox.pdmodel.encryption.AccessPermission.`** The main features available within the class for security configuration are:

- “print the document
- modify the content of the document
- copy or extract content of the document
- add or modify annotations
- fill in interactive form fields
- extract text and graphics for accessibility to visually impaired people

- assemble the document
- print in degraded quality

This class can be used to protect a document by assigning access permissions to recipients. In this case, it must be used with a specific ProtectionPolicy. When a document is decrypted, it has a currentAccessPermission property which is the access permissions granted to the user who decrypted the document." [Class AccessPermission]

After defining the access permission to be applied, the protection features are attached to the document by using the StandardProtectionPolicy class. Aside from the access permission, the StandardProtectionPolicy object requires a master password and a user password as arguments.

"Opening the document with the correct owner password (assuming it is not the same as the user password) allows full (owner) access to the document. This unlimited access includes the ability to change the document's passwords and access permissions.

Opening the document with the correct user password (or opening a document that does not have a user password) allows additional operations to be performed according to the user access permissions specified in the document's encryption dictionary." [PDF Reference, page 96]

Additionally, setting the encryption key length in bits is required.

After applying the protection policy, an interactive graphical user interface will appear on attempting to open the document in question, prompting the user to insert either the user password or the master password.

The screenshot below demonstrates the efficient application of security restriction to the holder of the user password:

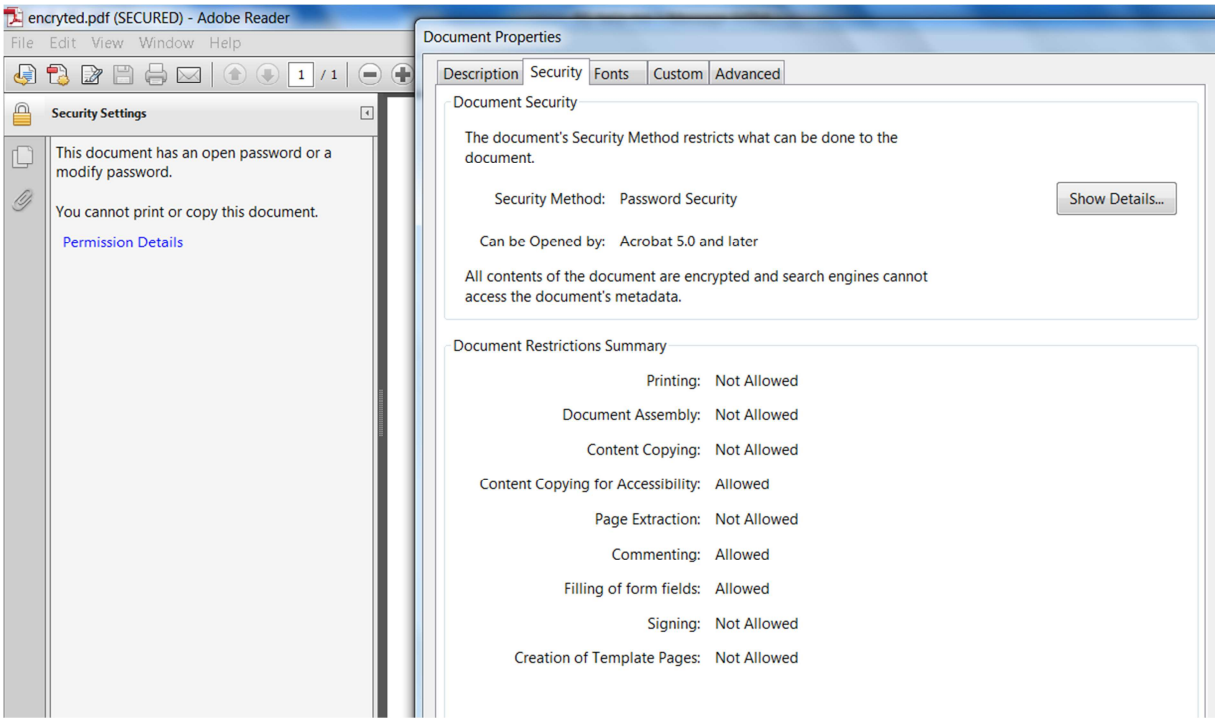

Figure 7: Encrypting a PDF document - outcome

When trying to handle a PDF file out of the command line, an encrypted document will return an error due to insufficient access permissions. Therefore, the following two methods need to be applied directly to the PDDocument object when importing the file:

- `isEncrypted` – use this method to verify if document requires decryption
- `decrypt(String password)` – use this method to provide a password for decryption

Furthermore, the method `~getCurrentAccessPermission` will return the current access permission for the provided password.

References for this example:

[PDFBox PDDocument]

[PDFBox PDPage]

[PDFBox AccessPermission]

[PDFBox StandardProtectionPolicy]

7. Conclusion and Outlook

The nutshell examples presented in this paper are meant to ensure an easy way of creating and managing a PDF file. The main purpose of this paper was to demonstrate the practicability of the PDFBox package. Therefore eleven nutshells have been developed as a solution to different issues a user is often confronted with.

The examples can be applied in arbitrary order or combined for greater impact. However, the benefit that a developer will gain by using the Apache PDFBox library consists in the incorporation of a PDF file in constructing programs.

The PDF file format is widely spread. Most computer users are likely to have access to a PDF reader tool, which is available free of charge. By storing data in a PDF file, the content, as well as the formatting, can be shared without alterations caused by hardware or software setting.

Using a PDF file when developing a program, can disclose new possibilities. For instance, the outcome of data processing can be stored inside a PDF file and sent out for review or further discussions. Compatibility issues can be completely avoided. Another option is to automatically import data from a form, by retrieving the information in interactive elements of a PDF file. In this scenario, an electronic form can be mailed to participant users, regardless of the platform the individuals are using.

Operating the PDFBox library by using the ooRexx programming language assures for the full range of functions available with the classes of the PDFBox and still provides the advantages of the simple, human-centric language ooRexx. The presented nutshell examples demonstrated the easy usage of the Java classes imported to ooRexx by the Bean Scripting Framework. The examples are short and very powerful and also confer a detail understanding of actions performed ‘behind the scenes’ to a PDF file and its components.

Although only limited capacity of the BSF library has been shown, the reader should now be aware of the reduction of complexity BSF4ooRexx introduces in creating source code.

After carefully reading the information and the source code provided in this paper and studying the PDFBox package, there are no limitations for managing a PDF file. If this is the case, the target of this paper has been achieved.

8. Bibliography

[Adobe1] Acrobat and PDF Library API Reference: All Acrobat Layers
http://livedocs.adobe.com/acrobat_sdk/9.1/Acrobat9_1_HTMLHelp/API_References/Acrobat_API_Reference/package-summary.html
Retrieved on 2012-07-25

[Adobe2] Developer Support: Legal Notices for Developers
http://partners.adobe.com/public/developer/support/topic_legal_notices.html
Retrieved on 2012-08-03

[Flatscher] Resurrecting REXX, Introducing Object Rexx, 2006
http://wi.wu-wien.ac.at/rgf/rexx/misc/ecoop06/ECOOP2006_RDL_Workshop_Flatscher_Paper.pdf
Retrieved on 2012-08-30

[Husband et al.] Husband, Mark; Nguyen, Dung; Wong, Stephen: Principles of Object-Oriented Programming, 2008.
http://florida.theorangegrove.org/og/file/33bf62f3-8ad1-7dde-e23f-6f17aca953c7/1/OOProgramming.pdf
Retrieved on 2012-08-10

[Java Array] Oracle: Class java.util.Arrays
http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html
Retrieved on 2012-11-13

[Java ArrayList] Oracle: Class **`java.util.ArrayList<E>`**
http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html
Retrieved on 2012-11-13

[Java BufferedWriter] Oracle: Class java.io.BufferedWriter
http://docs.oracle.com/javase/1.4.2/docs/api/java/io/BufferedWriter.html
Retrieved on 2012-08-26

[Java FileInputStream] Oracle: Class java.io.FileInputStream
http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileInputStream.html
Retrieved on 2012-08-26

[Java FileOutputStream] Oracle: Class java.io.FileOutputStream
http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileOutputStream.html
Retrieved on 2012-08-26

[Java OutputStreamWriter] Oracle: Class java.io.OutputStreamWriter
http://docs.oracle.com/javase/1.4.2/docs/api/java/io/OutputStreamWriter.html
Retrieved on 2012-08-26

[Nirosh] Nirosh: Introduction to Object Oriented Programming Concepts (OOP) and More, 25.01.2011
http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep
Retrieved on 2012-08-10

[Nørmark] Nørmark, Kurt: Functional Programming in Scheme, Programming Paradigms, 07.07.2010
http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html
Retrieved on 2012-08-10

[ooRexx] Ashley, W. David; Flatscher, Rony G.; Hessling, Mark; McGuire, Rick; Miesfeld, Mark; Peedin, Lee; Wolfers, Jon: ooRexx Programming Guide, Version 4.1.0 Edition, 2010.
www.oorexx.org/docs/rexxpg/rexxpg.pdf
Retrieved on 2012-08-03

[ooRexx1] What is Open Object Rexx?
http://www.oorexx.org/about.html
Retrieved on 2012-08-30

[ooRexx2] Built-In Functions. ooRexx Reference
http://www.oorexx.org/docs/rexxref/x23579.htm
Retrieved on 2012-11-21

[Parker] Parker, Thom: Navigating the Internal Structure of a PDF Document. http://www.planetpdf.com/developer/article.asp?ContentID=navigating_the_internal_struct
Retrieved on 2012-07-19

[PDF Box PDPageContentStream] Apache: Class org.apache.pdfbox.pdmodel.edit.PDPageContentStream
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/edit/PDPageContentStream.html
Retrieved on 2012-08-16

[PDF Reference] Adobe Systems Incorporated: PDF Reference fifth edition – Adobe® Portable Document Format, Version 1.6
http://stuff.mit.edu/afs/sipb/contrib/doc/specs/software/adobe/pdf/PDFReference16-v4.pdf
Retrieved on 2012-08-26

[PDFBox AccessPermission] Apache: Class org.apache.pdfbox.pdmodel.encryption.AccessPermission
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/encryption/AccessPermission.html
Retrieved on 2012-09-16

[PDFBox PDDestination] Apache: Class org.apache.pdfbox.pdmodel.interactive.documentnavigation.destination.PDDestination http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/interactive/documentnavigation/destination/PDDestination.html Retrieved on 2012-09-06

[PDFBox PDDocument] Apache: Class org.apache.pdfbox.pdmodel.PDDocument http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/PDDocument.html Retrieved on 2012-08-16

[PDFBox PDDocumentCatalog] Apache: Class org.apache.pdfbox.pdmodel.PDDocumentCatalog http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/PDDocumentCatalog.html Retrieved on 2012-08-30

[PDFBox PDDocumentOutline] Apache Class org.apache.pdfbox.pdmodel.interactive.documentnavigation.outline.PDDocumentOutline http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/interactive/documentnavigation/outline/PDDocumentOutline.html Retrieved on 2012-08-30

[PDFBox PDFMergerUtility] Apache: Class org.apache.pdfbox.util.PDFMergerUtility http://pdfbox.apache.org/apidocs/org/apache/pdfbox/util/PDFMergerUtility.html Retrieved on 2012-09-02

[PDFBox PDFTextStripper] Apache: Class org.apache.pdfbox.util.PDFTextStripper http://pdfbox.apache.org/apidocs/org/apache/pdfbox/util/PDFTextStripper.html Retrieved on 2012-08-20

[PDFBox PDJpeg] Apace: Class org.apache.pdfbox.pdmodel.graphics.xobject.PDJpeg http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/graphics/xobject/PDJpeg.html Retrieved on 2012-08-18

[PDFBox PDOutlineItem] Apache: Class org.apache.pdfbox.pdmodel.interactive.documentnavigation.outline.PDOutlineItem http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/interactive/documentnavigation/outline/PDOutlineItem.html Retrieved on 2012-09-06

[PDFBox PDPage] Apache: Class org.apache.pdfbox.pdmodel.PDPage http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/PDPage.html Retrieved on 2012-08-16

[PDFBox PDPageFitDestination] Apache: Class org.apache.pdfbox.pdmodel.interactive.documentnavigation.destination.PDPageFitDestination
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/interactive/documentnavigation/destination/PDDestination.html
Retrieved on 2012-09-06

[PDFBox PDType1Font] Apache: Class org.apache.pdfbox.pdmodel.font.PDType1Font
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/font/PDType1Font.html
Retrieved on 2012-08-16

[PDFBox Splitter] Apache: Class org.apache.pdfbox.util.Splitter
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/util/Splitter.html
Retrieved on 2012-09-05

[PDFBox Standard 14 Fonts] Apache: Developers Guide – Fonts
http://pdfbox.apache.org/userguide/fonts.html
Retrieved on 2012-10-02

[PDFBox StandardProtectionPolicy] Apache: Class org.apache.pdfbox.pdmodel.encryption.StandardProtectionPolicy
http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/encryption/StandardProtectionPolicy.html
Retrieved 2012-09-16

[PDFBox1] Apache PDFBox - Java PDF Library
http://pdfbox.apache.org/
Retrieved on 2012-07-12

[RexxLA] The Rexx Language Association: What is Rexx?
http://www.rexxla.org/rexxlang/
Retrieved on 2012-08-30

[Tiobe] TIOBE Programming Community Index for October 2012
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
Retrieved on 2012-10-27

[Wiki1] Wikipedia: The Portable Document Format
http://de.wikipedia.org/w/index.php?title=Portable_Document_Format&oldid=109758542
Retrieved on 2012-07-16

[Wiki2] Wikipedia: Java (programming language)
http://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=519946505
Retrieved on 2012-10-27

List of Abbreviations

API	Application programming interface
BSF	Bean Scripting Framework
BSF4ooRexx	Bean Scripting Framework for object oriented Rexx
cls	class
CPL	Common Public License
int	integer
OOP	Object oriented programming
ooRexx	Object oriented Rexx
PDF	Portable Document Format
Rexx	Restructured Extended Executor