Bachelor's thesis at the Institute for Information Systems and Society

An Introduction to Apache PDFBox Library: Nutshell Examples

Tiao Wang (h0553521)

Supervisor: ao.Univ.Prof. Dr. Rony G. Flatscher

Vienna,Select date

Table of contents

Abstract II
Table of figures III
1 Introduction 1
2 Background 2
2.1 Portable Document Format 2
2.2 Apache PDFBox 5
2.3 Java 7
2.4 Open Object Rexx 8
2.5 Bean Scripting Framework for ooRexx 10
3 Installation 11
3.1 Java 11
3.2 ooRexx 11
3.3 BSF4ooRexx 12
3.4 Apache PDFBox 12
4 Nutshell Examples 13
4.1 Creating a PDF Document 13
4.2 Adding Text to an Existing PDF Document 16
4.3 Use Different Fonts and Colors 18
4.4 Extracting Text from an Existing PDF Document 21
4.5 Drawing some Shapes 23
4.6 Creating a Table with Content 26
4.7 Converting a PDF Document To Image 29
4.8 Inserting Image to a PDF Document 31
4.9 Adding Multiple Pages to a PDF Document 33
4.10 Splitting a PDF Document with Multiple Pages 36
4.11 Merging Multiple PDF Documents 38
4.12 Setting the Document Metadata 40
4.13 Adding Watermark to a Document 42
4.14 Encrypting a PDF Document 45
4.15 Creating a PDF-A Document 48
4.16 Validating a PDF-A Document 52
4.17 Creating a Digital Signature 54
4.18 Verifying a Digital Signature 58
5 Conclusio 61
6 References 62
Appendix 64

Abstract

This thesis presents a collection of 18 nutshell examples demonstrating the use of Apache PDFBox, a Java library for creating and manipulating PDF documents. The examples are implemented using ooRexx, a high-level object-oriented scripting language, and BSF4ooRexx, a bridge between ooRexx and Java. The thesis provides an overview of the PDF format and the features of PDFBox, followed by the implementation of the examples, which cover a range of functionality such as creating a new PDF document, adding text, images, and annotations, manipulating existing PDF documents, and extracting data from them. The examples are designed to be concise and easy to follow, allowing users to quickly understand how to use PDFBox to accomplish various tasks related to PDF document creation and manipulation.

Table of figures

Figure 1: *"01. Creating a PDF Document.rex"* 14
Figure 2: Output of *"01. Creating a PDF Document.rex"* 15
Figure 3: *"02. Adding Text to an Existing PDF Document.rex"* 16
Figure 4: Warning for the overwrite mode 17
Figure 5: Output of *"02. Adding Text to an Existing PDF Document.rex"* 17
Figure 6: *"03. Use Different Fonts and Colors.rex"*... 19
Figure 7: Output of *"03. Use Different Fonts and Colors.rex"*.......................... 20
Figure 8: *"04. Extracting Text from an Existing PDF Document.rex"* 21
Figure 9: Output of *"04. Extracting Text from an Existing PDF Document.rex"*... 22
Figure 10: *"05. Drawing some Shapes.rex"* 24
Figure 11: Output of *"05. Drawing some Shapes.rex"* 25
Figure 12: *"06. Creating a Table with Content.rex"*....................................... 27
Figure 13: Output of *"06. Creating a Table with Content.rex"*.......................... 28
Figure 14: *"07. Converting a PDF Document To Image.rex"* 29
Figure 15: Output of *"07. Converting a PDF Document To Image.rex"*............. 30
Figure 16: *"08. Inserting Image to a PDF Document.rex"* 31
Figure 17: Output of *"08. Inserting Image to a PDF Document.rex"* 32
Figure 18: *"09. Adding Multiple Pages to a PDF Document.rex"* 34
Figure 19: Output of *"09. Adding Multiple Pages to a PDF Document.rex"* 35
Figure 20: *"10. Splitting a PDF Document with Multiple Pages.rex"* 36
Figure 21: Output of *"10. Splitting a PDF Document with Multiple Pages.rex"* 37
Figure 22: *"11. Merging Multiple PDF Documents.rex"*................................... 38
Figure 23: Output of *"11. Merging Multiple PDF Documents.rex"* 39
Figure 24: *"12. Setting the Document Metadata.rex"*.................................... 40
Figure 25: Output of *"12. Setting the Document Metadata.rex"*...................... 41
Figure 26: *"13. Adding Watermark to a Document.rex"*................................. 43
Figure 27: Output of *"13. Adding Watermark to a Document.rex"*................... 44
Figure 28: *"14. Encrypting a PDF Document.rex"* 45
Figure 29: Output 1 of *"14. Encrypting a PDF Document.rex"*.......................... 46
Figure 30: Output 2 of *"14. Encrypting a PDF Document.rex"*.......................... 46
Figure 31: Output 3 of *"14. Encrypting a PDF Document.rex"*.......................... 47
Figure 32: *"15. Creating a PDF-A Document.rex"* 49
Figure 33: Output of *"15. Creating a PDF-A Document.rex"* 51
Figure 34: *"16. Validating a PDF-A Document.rex"* 53
Figure 35: Output of *"16. Validating a PDF-A Document.rex"* 53
Figure 36: Keystore creation 54

Figure 37: *"17. Creating a Digital Signature.rex"*.. 55
Figure 38: Output of *"17. Creating a Digital Signature.rex"*............................ 57
Figure 39: *"18. Verifying a Digital Signature.rex"* 59
Figure 40: Output of *"18. Verifying a Digital Signature.rex"* 60

1 Introduction

In recent years, the use of PDF files has become ubiquitous in a wide range of industries and applications. The Portable Document Format (PDF) provides a versatile and reliable means of sharing and distributing documents across different devices and platforms. However, working with PDF files can often be challenging, especially when it comes to manipulating and customizing them to specific needs.

This Bachelor's thesis explores the use of PDFBox, an open-source Java library, to create and manipulate PDF documents. Specifically, we investigate how the combination of PDFBox with the programming languages of Java and ooRexx, along with the BSF4ooRexx bridge, can be used to create powerful, flexible, and efficient PDF manipulation scripts.

The thesis is structured into four main sections:

The first section introduces the topic, explaining the importance of PDF files and the challenges associated with working with them. This section also outlines the goals and objectives of the thesis.

The second section offers background information on PDFBox, Java, and ooRexx, including their features and capabilities. This section also explains how these components can be used together to create PDF manipulation scripts.

The third chapter will provide a step-by-step installation guide for the necessary components, including ooRexx, Java, and BSF4ooRexx. This chapter aims to ensure that readers can install the components and start working with PDFBox effectively.

The fourth section comprises 18 Nutshell examples that demonstrate the extensive capabilities of PDFBox for creating and manipulating PDF documents. These examples cover a range of practical applications, such as creating PDF documents, extracting text and images from PDF files, and merging and splitting PDF documents.

Finally, the thesis will conclude with a summary of the main findings and contributions of the study. Overall, this bachelor’s thesis offers valuable insights into the potential of PDFBox and ooRexx for creating and manipulating PDF documents. The Nutshell examples provided demonstrate the power and flexibility of this platform and provide a valuable resource for developers and users alike.

2 Background

To implement the nutshell examples presented in this bachelor thesis, the use of several technologies and components is required. This chapter will provide information about the relevant components in order to understand the implementing process.

2.1 Portable Document Format

Portable Document Format (PDF) is a widely recognized file format that has revolutionized the way documents are shared and accessed digitally. Developed by Adobe Systems in the early 1990s, PDF was created to provide a way to share documents across different computer platforms while preserving the document's layout, fonts, and images. Today, PDF has become one of the most popular and widely used document formats in the world, used across a wide range of industries, including publishing, printing, legal, and government documents [1].

The beauty of PDF lies in its ability to maintain document formatting and layout, regardless of the platform or device used to open it. This means that PDFs can be shared, viewed, and printed with ease, regardless of the operating system, software program, or device used [2]. In addition, PDFs can be secured with passwords and digital signatures, ensuring document confidentiality and authenticity [3].

PDF was created by Adobe Systems in the early 1990s, and its development was driven by the need for a universal file format that could be used across different computer systems. The idea for PDF came from John Warnock, the co-founder of Adobe Systems. In 1991, Warnock wrote a paper called "The Camelot Project" that proposed a new way of working with documents. The paper described a system that would allow documents to be viewed and printed on any computer, regardless of the software used to create them. The system would also allow documents to be stored electronically and distributed easily [4]. Warnock's vision was to create a document format that would be as reliable and consistent as paper, but more versatile and portable. He believed that by creating a universal format for documents, it would be possible to improve the way people worked with information, making it easier to share and collaborate on documents.

The first version of PDF was released in 1993, and it quickly gained popularity among businesses and organizations that needed a reliable way to share and distribute documents. The format became especially popular in industries such as publishing, where it was used to create electronic versions of books and other printed materials. In 2008, Adobe Systems released the PDF 1.7 specification as an ISO standard, making PDF an open standard that could be used by anyone [5]. This move helped to further establish PDF as a universal document format, and it paved the way for the development of a wide range of PDF-related software tools and applications. Today, PDF is one of the most widely used document formats in the world, with millions of PDF documents created and shared every day. It has become an essential tool for businesses, governments, and individuals who need a reliable way to share and exchange information. And with the rise of mobile devices, PDF has become even more important, as it allows users to access and view documents on a wide range of devices, including smartphones and tablets.

PDF has gone through several versions since its inception. Each version has introduced new features and improvements that have made PDF an increasingly powerful and versatile format for creating, sharing, and exchanging documents. Below, we will explore the version history of PDF [6].

- PDF 1.0 - The first version of PDF was released by Adobe Systems in 1993. It was based on PostScript, a page description language developed by Adobe. PDF 1.0 introduced basic features such as the ability to embed fonts, images, and other media into documents. It also allowed for documents to be viewed and printed on any computer, regardless of the software used to create them.
- PDF 1.1 - This version, released in 1996, introduced support for interactive form elements, such as text boxes and radio buttons. It also included support for annotations, which allowed users to add comments and notes to PDF documents.
- PDF 1.2 - Released in 1998, introduced support for digital signatures and encryption, making PDF documents more secure. It also included support for multimedia elements, such as audio and video.
- PDF 1.3 - This version, released in 2000, introduced support for layers, which allowed users to control the visibility of different elements in a document. It also included support for color management, making it easier to create accurate color representations in PDF documents.
- PDF 1.4 - Released in 2001, PDF 1.4 introduced support for transparency, allowing for more sophisticated graphics and designs. It also included support for tagged PDF, which made PDF documents more accessible for users with disabilities.

- PDF 1.5 - This version, released in 2003, introduced support for JPEG2000 compression, which made it possible to create smaller PDF files without sacrificing image quality. It also included support for 3D graphics and interactive multimedia elements.
- PDF 1.6 - Released in 2004, PDF 1.6 introduced support for layers that could be nested within one another, making it easier to organize complex documents. It also included support for live transparency, which allowed for more complex and dynamic designs.
- PDF 1.7 - The final version of PDF to be released by Adobe Systems, PDF 1.7 was introduced in 2006. It included a range of new features, such as support for the Adobe XML architecture, enhanced support for digital signatures, and improved handling of complex graphics and fonts.
- PDF 2.0 - In 2017, PDF 2.0 was released by the International Organization for Standardization (ISO). This version introduced a range of new features, including support for hybrid PDFs that can include both PDF and HTML content, improved support for 3D graphics and annotations, and enhanced security features.

PDF has gained popularity for various reasons. It offers a multitude of benefits over other file formats, making it an ideal choice for sharing and exchanging documents. One of the most significant advantages of PDF is its ability to preserve formatting. Unlike other file formats, PDF documents retain their formatting and layout regardless of the software or device used to view them. This means that the original document's fonts, colors, images, and graphics are preserved, ensuring that the document looks the same on any device. This is particularly important for documents like contracts, where formatting is crucial [7]. PDF also offers robust security features, such as password protection and encryption, that help safeguard sensitive documents from unauthorized access or modification. These features ensure that only authorized users can access or edit the document, making it an ideal choice for confidential documents such as financial reports, legal contracts, or medical records [8]. Another benefit of PDF is its smaller file size. PDF files can be compressed to reduce their size without compromising the document's quality. This means that PDF documents take up less storage space, making them easier to share, store, and transfer over the internet. This is particularly important for large documents that would otherwise be difficult to email or upload to a website [9]. PDF documents are also searchable, making it easy to find specific information within a document quickly. This feature is especially useful for large documents like textbooks or research papers. The search function allows users to find information quickly without having to read through the entire document [10]. PDF is also cross-platform compatible, meaning that it can be viewed and printed on any device or operating system. This makes it easy to share documents with others, regardless of their device or software. PDF files are also easy to create, edit, and share using various software applications, such as Adobe Acrobat, Microsoft Word, or Google Docs.

2.2 Apache PDFBox

Apache PDFBox is an open-source Java library for working with PDF documents. It allows developers to create, manipulate, and extract data from PDF files programmatically. Apache PDFBox was first released in 2008 and has since become a popular tool among developers due to its extensive features and ease of use. It is licensed under the Apache License 2.0, which means that it can be used for both commercial and non-commercial purposes [11].

Apache PDFBox was initially developed by Ben Litchfield in 2002 as a personal project. At that time, PDFBox was a simple Java-based tool that could extract text from PDF documents. In 2008, PDFBox was accepted as an Apache Incubator project and officially became part of the Apache Software Foundation [12]. It was designed to be a lightweight and easy-to-use library for Java developers who needed to work with PDF documents. Over time, PDFBox continued to evolve and mature, adding new features and capabilities. In 2010, version 1.0.0 was released, marking a major milestone in the project's development. This version included several significant improvements, including enhanced support for digital signatures, font embedding, and color spaces [13]. In 2012, PDFBox 1.7 was released, and it introduced support for the PDF/A standard. PDF/A is a subset of the PDF format that is specifically designed for long-term archiving of electronic documents, ensuring their preservation and accessibility over time. With the release of PDFBox 1.7, users were able to create PDF/A documents that met the ISO standard for archiving. PDFBox 1.7 offered features such as embedding fonts, ensuring all necessary metadata was present in the document, and validating PDF/A compliance. This made it a valuable tool for organizations that needed to create and manage long-term digital archives. One of the key milestones in the history of PDFBox was the release of version 2.0 in 2016. This major release included significant changes and improvements, such as better support for PDF 2.0, improved font handling, and enhanced text extraction capabilities. It also introduced a new modular architecture, making it easier for developers to use and extend the library [14]. PDFBox 3.0 is the latest major release of the Apache PDFBox library, and it was released in January 2021. This release introduced many new features and improvements, including significant performance enhancements, Java 11 compatibility, and new functionality for digital signature handling and document creation [15].

The Apche PDFBox library is made up of four main components: PDFBox, FontBox, XMPBox, and Preflight [12].

- PDFBox is the core component of the library and provides the main functionality for working with PDF files, including parsing, creation, and manipulation of PDF documents. It allows developers to extract text, images, and other content from PDF files, as well as add, remove, or modify elements within a PDF document.
- FontBox is a component of PDFBox that provides support for font manipulation. It allows developers to extract information about the fonts used in a PDF document, as well as embed or subset fonts to reduce file size and ensure that the document is displayed correctly.
- XMPBox is a component of PDFBox that provides support for Extensible Metadata Platform (XMP) metadata. It allows developers to extract, modify, and add XMP metadata to PDF documents, which can be used to provide additional information about the document, such as author, date, and copyright information.
- Preflight is a component of PDFBox that provides preflighting functionality. It allows developers to check whether a PDF document conforms to certain standards, such as the PDF/A or PDF/X standards and provides detailed reports of any issues that are found.

One of the most significant advantages of PDFBox is that it is entirely free to use, making it accessible to everyone, regardless of their budget. Additionally, PDFBox is designed to be compatible with multiple platforms, including Windows, Mac, and Linux, which means users can access and use it from any device. PDFBox's comprehensive functionality is another significant benefit. It can perform a variety of tasks, including creating, modifying, and extracting content from PDF files. Additionally, it supports advanced features such as encryption and digital signatures, which makes it an ideal tool for businesses that deal with sensitive information.

Another important benefit of PDFBox is its high performance. It can handle large files and perform complex operations quickly, saving users valuable time and resources. Finally, PDFBox benefits from an active community of developers and users, who continuously work to improve and update the tool, making it an excellent choice for anyone looking for a reliable and versatile PDF solution [12].

2.3 Java

Java is a high-level, class-based and object-oriented programming language that has become a popular choice for software development since it was first released in 1995. Developed by Sun Microsystems (now owned by Oracle Corporation), Java was designed to be platform-independent, meaning that it can run on multiple operating systems without requiring recompilation [16]. This makes Java a highly versatile language, widely used for developing everything from desktop applications to mobile apps, web applications, and enterprise software. Java's popularity can be attributed to its simplicity, robustness, and security features [17]. It has a vast library of built-in classes and functions, making it easy to write complex programs quickly. Additionally, Java has a vast community of developers, who contribute to open-source projects, share code snippets, and provide support to new programmers. Java is an object-oriented language, which means that it is based on the concept of objects. Objects are instances of classes, which are templates that define the properties and methods of an object. Java's object-oriented programming model enables developers to write modular, reusable code that is easy to maintain and update [18].

For Java applications to run on a computer or device, they require the Java Runtime Environment (JRE), which is a software package that includes the Java Virtual Machine (JVM) and other necessary components. The JVM is a key component of the JRE, and it is responsible for executing Java bytecode [19]. Bytecode is a low-level, machine-independent code that is generated by the Java compiler when a program is compiled [20]. The JVM interprets this bytecode and executes it on the underlying system, providing a platform-independent runtime environment for Java applications. One of the key benefits of the JVM is that it provides a layer of abstraction between the Java code and the underlying operating system. This means that developers can write code that runs on any system that has a compatible JVM installed, without needing to worry about the specific details of the operating system or hardware [21]. This makes it much easier to develop cross-platform applications that can run on a variety of devices and operating systems. The JRE also includes a number of other components, such as class libraries, that are necessary for running Java applications. Class libraries are collections of pre-built classes and functions that developers can use to speed up development and reduce the amount of code they need to write. These class libraries cover a wide range of functionality, from basic data types and control structures to more advanced features such as network programming and graphical user interfaces [22]. While the JRE is an essential component of the Java ecosystem, it is important to note that it is separate from the Java Development Kit (JDK), which includes additional tools such as the Java compiler and debugger. The JRE is primarily used for running Java applications, while the JDK is used for developing and compiling Java code [23].

2.4 Open Object Rexx

Open Object Rexx (ooRexx) is an open-source programming language that is a modern implementation of the Rexx programming language. It was created by the Rexx Language Association and is available for a wide range of platforms, including Windows, Linux, and macOS [24]. ooRexx is based on the original Rexx language created by Mike Cowlishaw in the late 1970s. Rexx, short for "REstructured eXtended eXecutor," was designed to be an easy-to-learn and easy-to-use language that could be used for a wide range of tasks, including scripting, systems programming, and more. Over the years, Rexx was implemented on a variety of platforms, including mainframe computers, personal computers, and even some embedded systems. It gained popularity as a scripting language for various applications, including the OS/2 operating system [25]. In the 1990s, IBM released an updated version of Rexx called Object Rexx, which added support for object-oriented programming [24]. In 2004, the Open Object Rexx project was started to create an open-source implementation of Object Rexx. The goal was to make the language more accessible to developers and to encourage its adoption in the open-source community. ooRexx is maintained by a group of volunteers who are dedicated to improving the language and making it available on a wide range of platforms, including Windows, Linux, macOS, and various Unix operating systems [26].

ooRexx offers several benefits that make it an attractive option for programmers. Here are some of the key advantages of ooRexx [27]:

- English-like language: ooRexx instructions use common English words, making the language more intuitive and easier to learn for beginners. Unlike some programming languages that use abbreviations and symbols, ooRexx instructions are easy to understand and remember.

- Fewer rules: ooRexx has relatively few rules about format, allowing programmers to write code in their preferred style. For example, instructions can span multiple lines, be typed in uppercase or lowercase, and include multiple instructions on a single line.
- Interpreted, not compiled: ooRexx is an interpreted language, which means that programs can be executed immediately without the need for compilation. This makes it faster to develop and test code, and also easier to modify and update programs.
- Built-in functions and methods: ooRexx comes with a rich set of built-in functions and methods that perform various processing, searching, and comparison operations for text and numbers. This saves programmers time and effort, as they don't have to write these functions from scratch.
- Typeless variables: ooRexx treats all data as objects of different kinds, allowing variables to hold any kind of object without the need for explicit type declarations. This flexibility simplifies programming and enables programmers to write more concise and readable code.
- String handling: ooRexx includes powerful functionality for manipulating character strings, enabling programs to read and separate characters, numbers, and mixed input. ooRexx also performs arithmetic operations on any string that represents a valid number, including those in exponential formats.
- Decimal Arithmetic: ooRexx uses decimal arithmetic, which is more accurate and natural for humans, instead of the binary arithmetic used by many other programming languages. This makes it easier to perform precise calculations without rounding errors or other inaccuracies.
- Clear error messages and powerful debugging: ooRexx provides clear error messages and a powerful debugging tool (TRACE instruction) to help programmers quickly identify and fix errors in their code. This saves time and effort during the development and testing phases.

2.5 Bean Scripting Framework for ooRexx

Bean Scripting Framework (BSF4ooRexx) is a bridge technology that enables the ooRexx scripting language to interact with Java classes and libraries. This technology provides an easy way to call Java code from ooRexx scripts, allowing developers to leverage the power of Java libraries and components from within their ooRexx scripts.

BSF4ooRexx was developed as an extension of the Bean Scripting Framework (BSF), which is a Java-based framework that provides a standardized way of integrating scripting languages with Java. The original BSF was developed by IBM and was later released as an open-source project under the Apache license [28]. BSF4ooRexx was developed as an extension of the original BSF project to add Java support for ooRexx.

The key benefit of BSF4ooRexx is the ability to leverage existing Java libraries and components from within ooRexx scripts. This enables developers to take advantage of the vast number of Java libraries available and provides an easy way to add scripting support to Java applications.

3 Installation

This chapter presents a comprehensive installation guide for various programming languages and components. This guide is designed to provide a step-by-step process for installing the software and tools necessary to replicate the nutshell examples presented in this thesis. It is important to follow each step carefully to ensure that the testing system is properly configured and able to run the necessary software.

3.1 Java

Firstly a Java Runtime Environment (JRE) is required to be installed on your computer.

- Step 1: Download Azul JDK
  Visit the Azul website at https://www.azul.com/downloads/ and select the appropriate version of Azul JDK for your operating system. Click on the download link to begin the download process.
- Step 2: Install Azul JDK
  Once the download is complete, locate the installation file and run it. Follow the prompts to install Azul JDK on your system. During the installation process, you may be prompted to select the installation directory and other configuration options.
- Step 3: Verify the installation
  To verify that Azul JDK has been installed successfully, open a command prompt or terminal and enter the following command: java -version If Azul JDK has been installed correctly, the output should display the version of Azul JDK that you have installed.

3.2 ooRexx

The next thing to do is to install the ooRexx environment. To do that, follow these three steps:

- Download ooRexx from the official website at https://sourceforge.net/projects/oorexx/ and navigate to the Download section and select the appropriate version for your operating system. Click on the download link to start the download process.
- Once the download is complete, locate the installation file and run it. Follow the installation prompts to install ooRexx on your system. You may need to select the installation directory and configure other options during the installation process.

- To verify the installation, open a command prompt or terminal and enter the following command: rexx -v. If ooRexx has been installed correctly, the output should display the version of ooRexx that you have installed.

3.3 BSF4ooRexx

Before proceeding with the installation, make sure to uninstall any previous versions of BSF4ooRexx on your system. To do this, use the menu "BSF4ooRexx -> Installation -> Uninstall BSF4ooRexx".

Next, follow these steps to install BSF4ooRexx:

- Download the installation archive "BSF4ooRexx_install-*.zip" from the BSF4ooRexx website https://sourceforge.net/projects/bsf4oorexx/ .
- Unzip the downloaded installation archive. This will also unzip the Apache OpenOffice/LibreOffice scripting-support for ooRexx.
- Change into the unzipped subdirectory "BSF4ooRexx/install/windows".
- Run the installation file in this directory to start the installation process.

3.4 Apache PDFBox

To install PDFBox, follow these steps:

1. Download PDFBox: Visit the PDFBox website at https://pdfbox.apache.org/ and navigate to the Download section. Select the latest version of PDFBox and download the binary distribution of the following libraries:
    - pdfbox-app-3.0.0-alpha3.jar
    - preflight-3.0.0-alpha3.jar
    - xmpbox-3.0.0-alpha3.jar

2. Unzip the downloaded archive: Once the download is complete, locate the downloaded archive and extract its contents to a directory on your computer.

3. Copy the PDFBox jar files: Navigate to the directory where you extracted the contents of the archive and copy the .jar files to the "BSF4ooRexx/lib" directory.

4 Nutshell Examples

In this chapter, 18 nutshell examples are presented. The Objective of these examples is to give an insight into the functionality and working concept of the Java library PDFbox. Each example will start with the code, which will be explained. At the end a screenshot of the result will be showed. It is highly recommended to run the examples in the right order, because many examples will build on the results of previous examples. It's also important to note that a "resources" folder exists in the root directory, and certain examples require files from this folder to run successfully. Please ensure that these files are present in the folder before running the codes. To gain a better understanding of the code presented in the nutshell examples, it is highly recommended to go through the slides of Business Programming 1 & 2 by Professor Rony Flatscher [29] [30].

4.1 Creating a PDF Document

The first nutshell example demonstrates how to create a writeable PDF document and add some text to it. Afterwards the document is saved with a chosen file name. For these tasks some simple steps need to be followed as the code showed below.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- create a new document 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
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)

-- define font type

fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font =.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert content
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
```

```
cont~newLineAtOffset(100, 700)
cont~showText("Hello World")
cont~endText
cont~close

-- save and close the document file
doc~save("01-new doc.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 1: *"01. Creating a PDF Document.rex"*

As one can see the code is very clearly structured and easy to understand. The very first thing to do is to get Java support within the ooRexx environment. For that we use the directive statement "::requires", which is executed before all other non-directive statements. The last line from the code loads the ooRexx module "BSF.CLS", which camouflages Java as ooRexx. So, the whole functionality of Java is fully granted.

The next thing we need to do is to change the directory to the location where the program is saved. The first line retrieves the location of the program and names it as "pgm". The dots in the command disregard other additional information, which is not needed in this example. The following line uses the retrieved program location and defines the current root directory, the parameter "L" here stands for location. The program can be moved to any new location without rewriting the code.

Now we can focus on the actual tasks of this example. At first a PDF document needs to be created. Therefore we use the statement ".bsf~new" to import the Java class "org.apache.pdfbox.pdmodel.PDDocument" and create a new instance of that class. At the same time the name "doc" is given to the newly created instance of this class. It is important to use the fully qualified name of the Java class. After that is done, the imported Java class can be treated as if it was an ooRexx class. For a PDF document to be valid it must contain at least one page. Therefore, an empty page needs to be created and added to the document. For that, a new 121instance of the Java class "PDPage" named "page" is created in the same way as "doc". The Java method "addPage", which was imported with the Java class, is used for adding the empty page to the document.

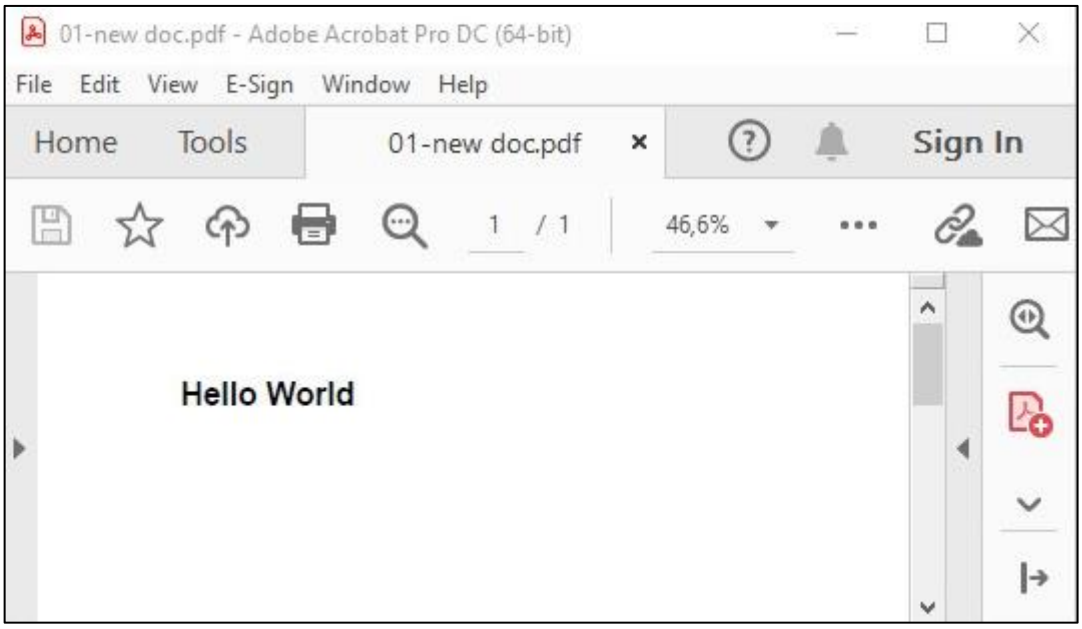
The PDFbox library uses the class " PDPageContentStream" for adding contents to the document. Once again, an instance of this class is created. For the creation of the cotentstream two parameters are needed, names of the document and the desired page. The next step is to set up the font type of the text we want to add. PDFbox comes with several build-in font types and one of them is selected and named "font".

For adding the text content to the document, the following methods are used:

- ~beginText: start of text input process
- ~setFont(font, 22): set up the default font type and size
- ~setLeading(25f): set up the default distance between two lines
- ~newLineAtOffset(100, 700): set up the start position of the text input
- ~showText("Hello World"): add desired text content to the document
- ~endText: end of the text input process
- ~close: close the contend stream

The final step is to save and close the PDF document with all the contents added. The methods "save" and "close" are used.

The following figure shows a screenshot of the resulting PDF document opened with Adobe Acrobat Pro.


01-new doc.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools
01-new doc.pdf
Sign In
1 / 1
46,6%

Hello World

Figure 2: Output of *"01. Creating a PDF Document.rex"*

4.2 Adding Text to an Existing PDF Document

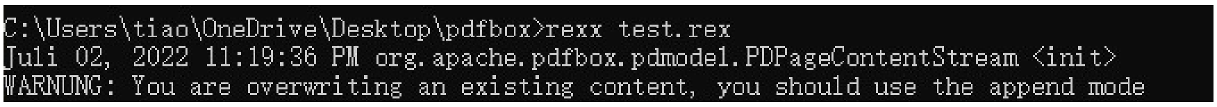
The second example demonstrates how to load an existing PDF document and put in additional contents.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "01-new doc.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)
page=doc~getPage(0)

-- set the append mode
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
append=BSF.loadClass(contclass)~AppendMode~APPEND

-- create a content stream
cont=.bsf~new(contclass,doc,page,append,"true")
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert content
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
cont~newLineAtOffset( 100, 600 )
cont~showText("Hello World, again")
cont~endText
cont~close

-- save and close the document file
doc~save("02-modified doc.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 3: *"02. Adding Text to an Existing PDF Document.rex"*

The code of the second example is similar to the code of the first example. First, we get Java support and define the directory of the program. Next step is to read the existing PDF Document created in the previous example. Therefore, the Java class “java.io.File” is used for providing access to the source file, which can be loaded now with the method “~loadPDF” imported with the Java class "org.apache.pdfbox.Loader". After the document is loaded the first page is selected for editing by the method “getPage”.

For adding contents to the document, the class “PDPageContentStream” is needed again. The tricky part here is the fact, that this class works in overwrite mode by default. The existing contents are replaced by the new content. The following figure shows the warning if the default setting is used.

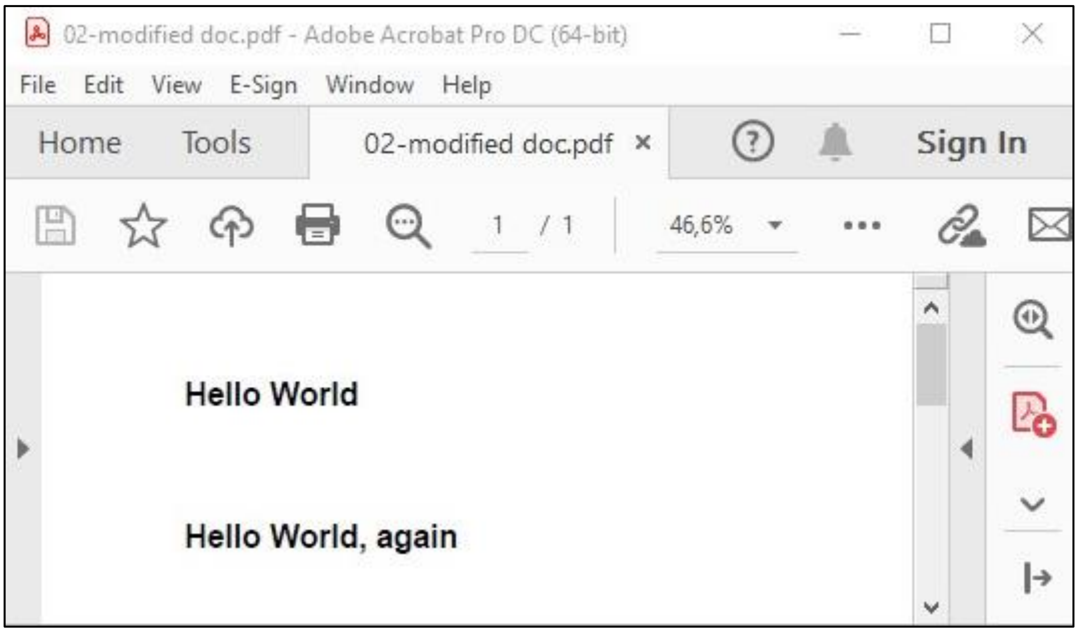


```
C:\Users\tiao\OneDrive\Desktop\pdfbox>rexx test.rex
Juli 02, 2022 11:19:36 PM org.apache.pdfbox.pdmodel.PDPageContentStream <init>
WARNUNG: You are overwriting an existing content, you should use the append mode
```

Figure 4: Warning for the overwrite mode

In this example, overwriting the existing contents is not desired. For that, the “AppendMode” needs to be set to “APPEND”. This allows us to create a content stream that works in the append mode.

For the new content the same methods are used. To prevent overlapping with the existing content the starting position of the new content has been changed. In the last step the document can be saved and closed. The following figure is a screenshot showing the resulting PDF document.


02-modified doc.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 02-modified doc.pdf ×
Sign In
1 / 1
46,6%

Hello World

Hello World, again

Figure 5: Output of *“02. Adding Text to an Existing PDF Document.rex”*

4.3 Use Different Fonts and Colors

This example shows how to insert text with different font types, size and colors on one single page.

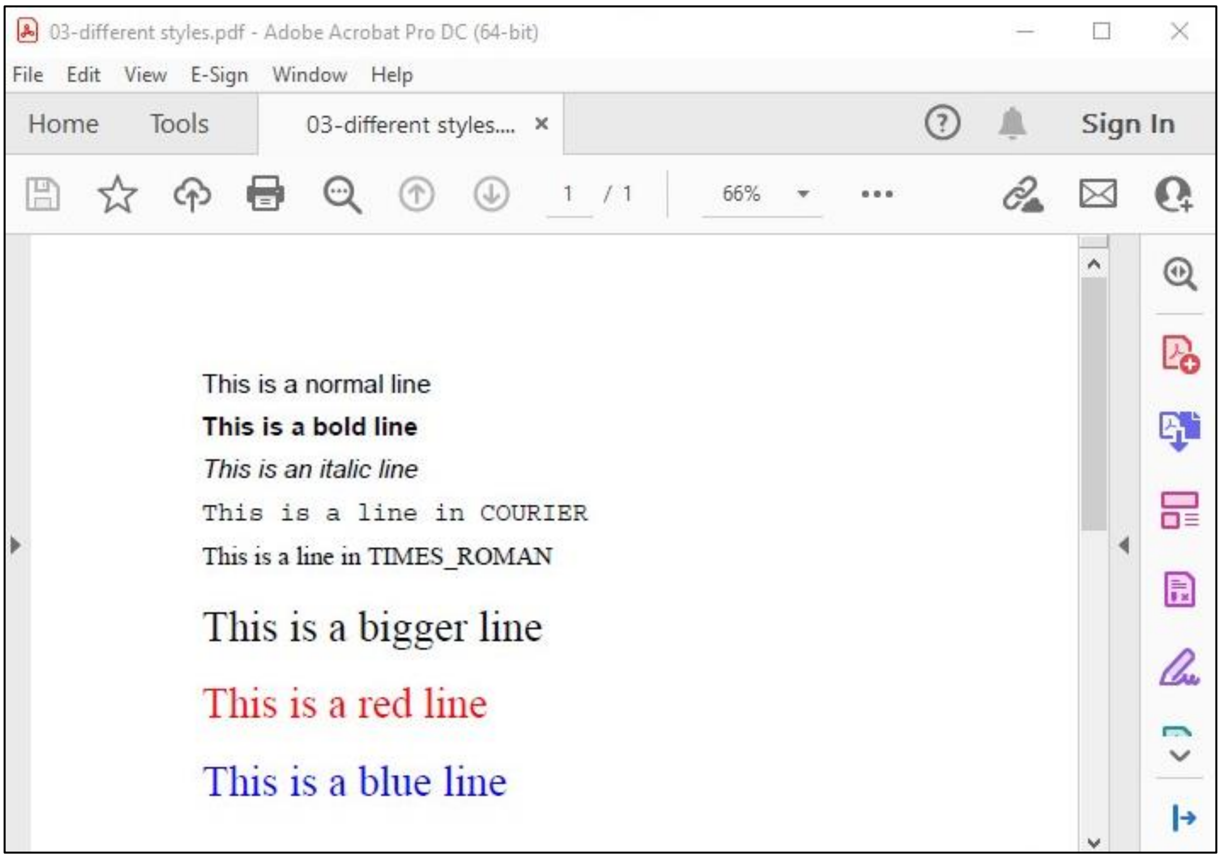
```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- create a new document 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
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)

-- define font type
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert content
cont~beginText
cont~setFont(font, 15)
cont~setLeading(25f)
cont~newLineAtOffset( 100, 700 )
cont~showText("This is a normal line")
cont~newLine

-- use a new font type
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)
cont~setFont(font, 15)
cont~showText("This is a bold line")
cont~newLine

-- use a new font type
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_OBLIQUE
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)
cont~setFont(font, 15)
cont~showText("This is an italic line")
cont~newLine

-- use a new font type
```

```
fname = BSF.loadClass(fontclass)~FontName~COURIER
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)
cont~setFont(font, 15)
cont~showText("This is a line in COURIER")
cont~newLine

-- use a new font type
fname = BSF.loadClass(fontclass)~FontName~TIMES_ROMAN
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)
cont~setFont(font, 15)
cont~showText("This is a line in TIMES_ROMAN")
cont~setLeading(45f)
cont~newLine

-- use a bigger font size
cont~setFont(font, 25)
cont~showText("This is a bigger line")
cont~newLine

-- use a new font color
col=BSF.loadClass("java.awt.Color")~red
cont~setNonStrokingColor(col)
cont~showText("This is a red line")
cont~newLine

-- use a new font color
col=BSF.loadClass("java.awt.Color")~blue
cont~setNonStrokingColor(col)
cont~showText("This is a blue line")
cont~endText
cont~close

-- save and close the document file
doc~save("03-different styles.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 6: *"03. Use Different Fonts and Colors.rex"*

First, a new PDF document with a blank page and the contentstream are created. Second, the font type "HELVETICA" and font size 15 are set as default. Then the text insert process is started. This process is repeated several times, each time with a different font type. The Java class "org.apache.pdfbox.pdmodel.font.Standard14Fonts" has to be reimported and set as default font type every time. For different font size it is sufficient to call the method "~setFont" with the desired parameter. Next step is to change the font color. Therefore, the Java class "java.awt.Color" is imported to provide color support. The method "~setNonStrokingColor" is used to set up default text color. Last step is to save and close the document.

The following figure shows the resulting PDF document.


03-different styles.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 03-different styles.... ×
Sign In
1 / 1
66%

This is a normal line
This is a bold line
This is an italic line
This is a line in COURIER
This is a line in TIMES_ROMAN
This is a bigger line
This is a red line
This is a blue line

Figure 7: Output of *"03. Use Different Fonts and Colors.rex"*

4.4 Extracting Text from an Existing PDF Document

In the last three examples we saw how to add text to a document under different circumstances. This example will demonstrate how to extract text content from an existing PDF document to a simple .txt file.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "03-different styles.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

-- use the stripper to get content
stripper=.bsf~new("org.apache.pdfbox.text.PDFTextStripper")
content=stripper~getText(doc)

-- write the content to the destination file
writer=bsf.import("java.io.FileWriter")
file= writer~new("04-extracted content.txt")
file~write(content)
file~close()

-- get java support
::requires "BSF.CLS"
```

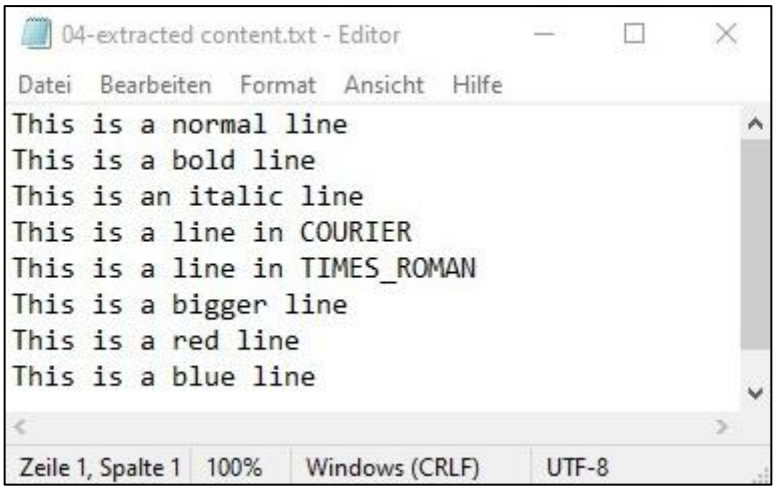
Figure 8: *"04. Extracting Text from an Existing PDF Document.rex"*

After getting Java support and setting up the directory location the source file is loaded. To read the content a new instance of the Java class "org.apache.pdfbox.text.PDFTextStripper" is created. Then the text content is saved by the method "~getText".

The next step is to create an empty text file and extract the content to it. Hereby the Java class "java.io.FileWriter" is needed. The following methods are used in the extract process:

- ~new: create a new file
- ~write: write content to the file
- ~close: close the writer

After the extract process the text content is saved without any formatting to a text file as the following figure shows.


04-extracted content.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

```
This is a normal line
This is a bold line
This is an italic line
This is a line in COURIER
This is a line in TIMES_ROMAN
This is a bigger line
This is a red line
This is a blue line
```

Zeile 1, Spalte 1 100% Windows (CRLF) UTF-8

Figure 9: Output of *"04. Extracting Text from an Existing PDF Document.rex"*

4.5 Drawing some Shapes

In the previous examples we saw how text processing with PDFbox works. This example will demonstrate how to use the content stream to draw different shapes.

In PDFbox, it is impossible to draw shapes directly, what we can do is to define a path from A to B in form of a line or a curve. A combination of these lines and curves can form complex shapes, which the following code will show.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- create a new document 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
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)
cont~setLineWidth(1)

-- draw a line
cont~moveTo(100,700)
cont~lineTo(250,700)

-- draw a curve
cont~moveTo(350,700)
cont~curveTo(380,720,420,750,500,700)

-- draw a rectangle
cont~addRect(120,600,100,-100)

-- draw a triangle
cont~moveTo(350,500)
cont~lineTo(450,500)
cont~lineTo(400,600)
cont~lineTo(350,500)

-- draw a star shape
cont~moveTo(120,360)
cont~lineTo(220,360)
cont~lineTo(140,300)
cont~lineTo(170,400)
```

```
cont~lineTo(200,300)
cont~lineTo(120,360)

-- draw a circle
cx=400
cy=350
r=50
k=0.552284749831
cont~moveTo(cx-r,cy)
cont~curveTo(cx-r,cy+k*r,cx-k*r,cy+r,cx,cy+r)
cont~curveTo(cx+k*r,cy+r,cx+r,cy+k*r,cx+r,cy)
cont~curveTo(cx+r,cy-k*r,cx+k*r,cy-r,cx,cy-r)
cont~curveTo(cx-k*r,cy-r,cx-r,cy-k*r,cx-r,cy)

-- make all the drawed shapes visible
cont~stroke
cont~close

-- save and close the document file
doc~save("05-shapes.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 10: *"05. Drawing some Shapes.rex"*

First, a new document with a blank page and the associated content stream are created. Then the method "~setLineWidth" is used to set up the line thickness of the path we want to draw.

The first shape to draw is a simple straight from one point to another point. The method "~moveTo" is used to define the coordinates of the starting point and the method "~lineTo" describes the path to the ending point.

Drawing a curve is slightly more complex. In addition to the starting point and ending point, 2 further control points are needed to describe the path of the curve. That is the reason why the method "~curveTo" has six parameters, which are the X/Y coordinates of the two control points and the ending point.

For drawing a rectangle, the method "~addRect" can be used to simplify the coding. With a starting point and the length/width this method automatically draws the four sides of the rectangle.

The next shape to draw is a triangle. Therefore, we must combine three lines to form the triangle. With a set of three Point chosen, the points are connected with each other. In the same way we can draw a star shape, by connecting five points with each other.

The last and the trickiest shape to draw is a circle. It is impossible to draw a perfect circle, but a very good approximation can be reached by combining several curves. In this example the circle is divided into four parts. First, the circle center and the radius are defined. Then the constant "k" is needed to determine the coordinates of the control points. So, these four curves are drawn to form the circle.

The final step is to use the method "~stroke" to make all the paths defined visible. Then the doc can be saved and closed.

The following figure shows the drawn shapes.

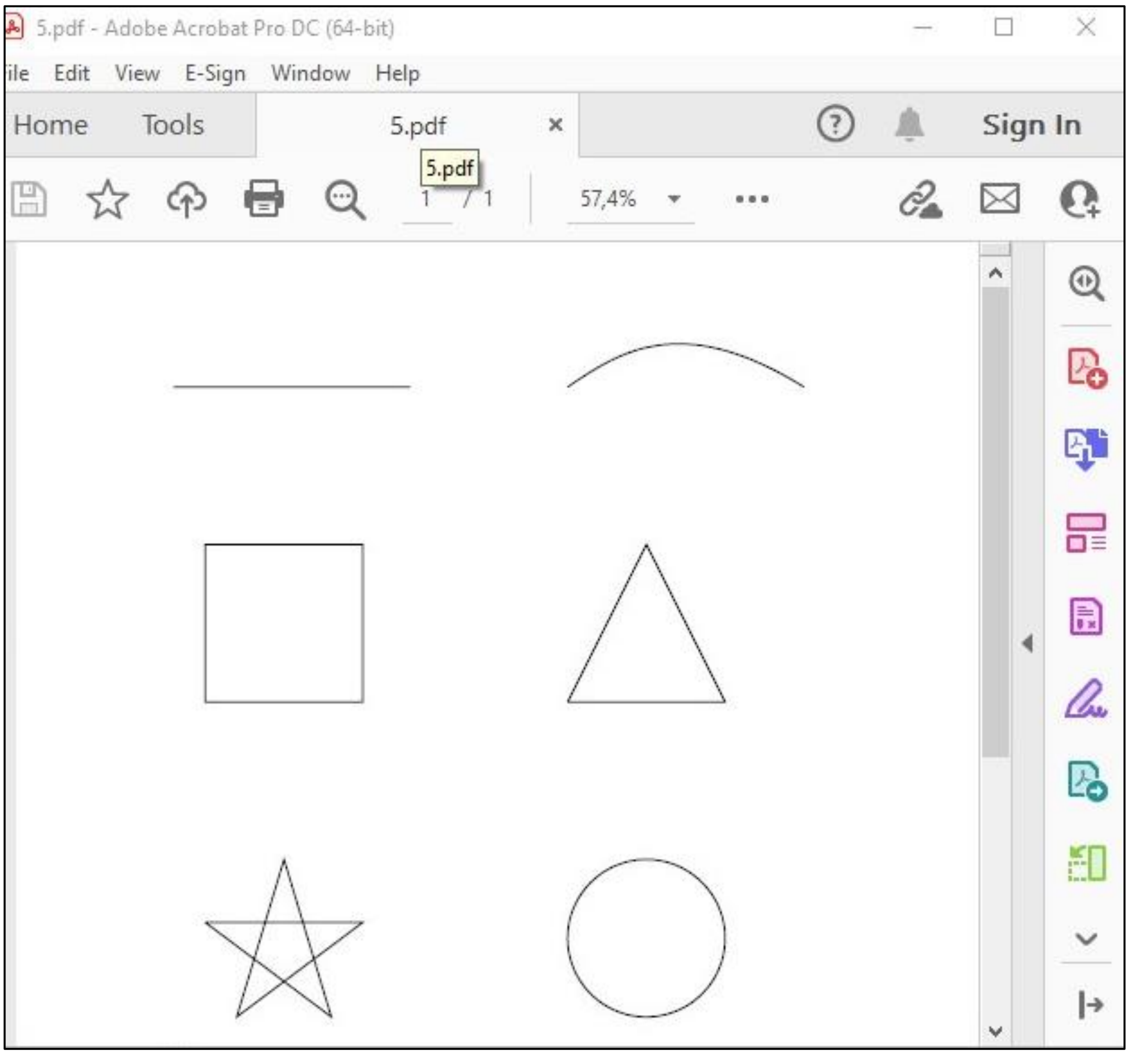

5.pdf - Adobe Acrobat Pro DC (64-bit)
ile Edit View E-Sign Window Help
Home Tools 5.pdf
Sign In
1 / 1
57,4%

Figure 11: Output of *"05. Drawing some Shapes.rex"*

4.6 Creating a Table with Content

This example demonstrates how to create a table and add contents to it. PDFbox does not include a build-in table editor. However, we can use the knowledge from the last example to draw the table manually.

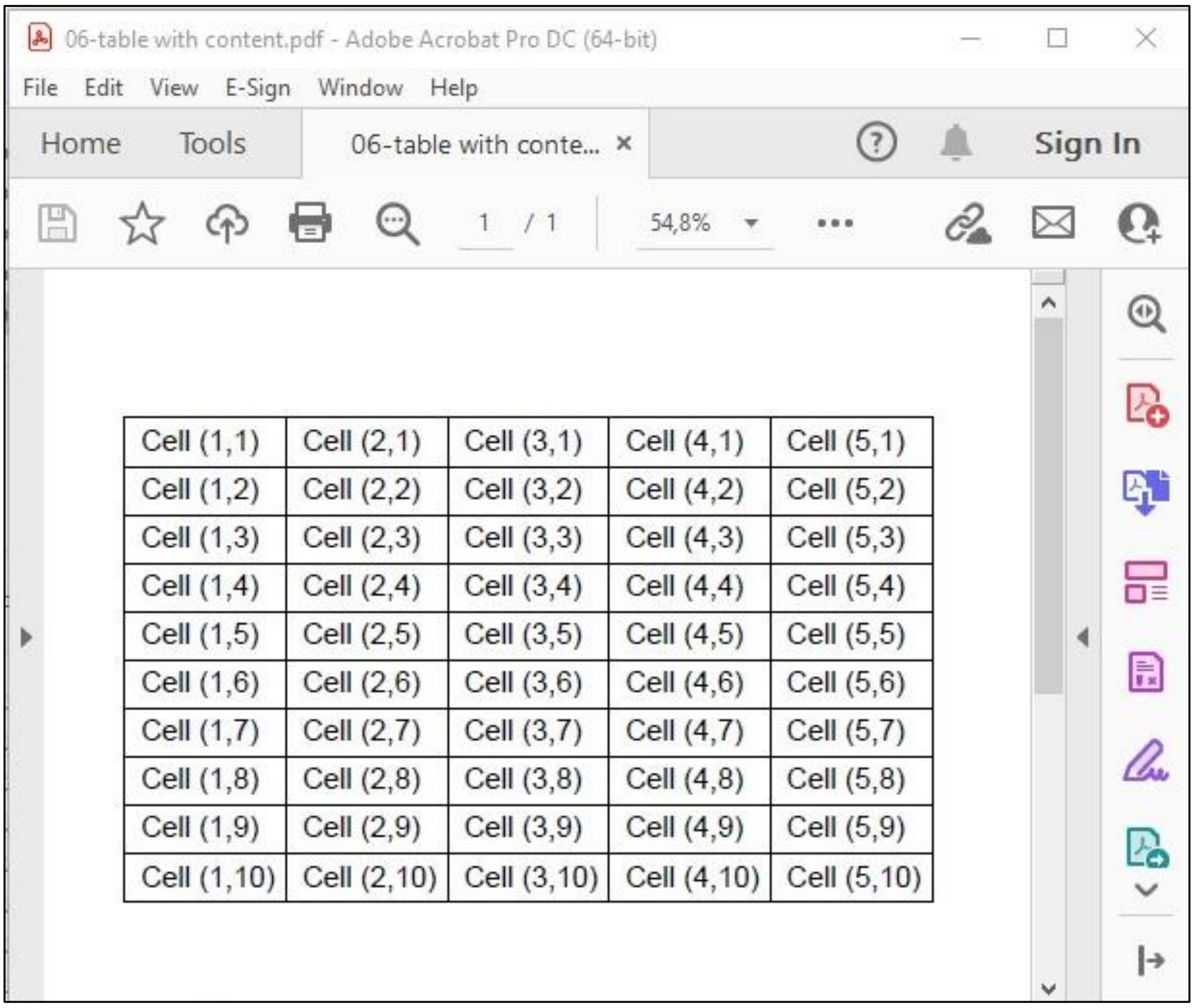
```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

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

-- define font type
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- create a content stream
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)
cont~setLineWidth(1)
cont~setFont(font, 18)

-- define the cell and table
initx=50
inity=700
cheight=30
cwidth=100
col=5
row= 10

-- start the loops to draw the table and insert contents
do i=0 to row-1
do j=0 to col-1
cont~addRect(initx+j*cwidth,inity-i*cheight,cwidth,-cheight)
cont~beginText
cont~newLineAtOffset(initx+j*cwidth+10,inity-i*cheight-cheight+10)
cont~showText("Cell" "(" || j+1 || "," || i+1 || ")")
cont~endText
end
end

-- make the table visible
```

```
cont~stroke
cont~close

-- save and close the document file
doc~save("06-table with content.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 12: *"06. Creating a Table with Content.rex"*

After the document and the content stream are created, the coordinates of the initial point and the size of the cells are defined. Then, the number of the columns and rows are defined. After the preparation is done, we can start to draw the table manually. Therefore, two do-loops are created, one each for the rows and columns. First, the method "~addRect" is used for drawing the table, cell by cell. Then, a text content is inserted to each of the cells. After the loops are done, the whole table is made visible by the methode "~stroke".

The following figure shows a screenshot of the resulting table.


06-table with content.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 06-table with conte... ×
Sign In
1 / 1
54,8%

Cell (1,1)	Cell (2,1)	Cell (3,1)	Cell (4,1)	Cell (5,1)
Cell (1,2)	Cell (2,2)	Cell (3,2)	Cell (4,2)	Cell (5,2)
Cell (1,3)	Cell (2,3)	Cell (3,3)	Cell (4,3)	Cell (5,3)
Cell (1,4)	Cell (2,4)	Cell (3,4)	Cell (4,4)	Cell (5,4)
Cell (1,5)	Cell (2,5)	Cell (3,5)	Cell (4,5)	Cell (5,5)
Cell (1,6)	Cell (2,6)	Cell (3,6)	Cell (4,6)	Cell (5,6)
Cell (1,7)	Cell (2,7)	Cell (3,7)	Cell (4,7)	Cell (5,7)
Cell (1,8)	Cell (2,8)	Cell (3,8)	Cell (4,8)	Cell (5,8)
Cell (1,9)	Cell (2,9)	Cell (3,9)	Cell (4,9)	Cell (5,9)
Cell (1,10)	Cell (2,10)	Cell (3,10)	Cell (4,10)	Cell (5,10)

Figure 13: Output of *"06. Creating a Table with Content.rex"*

4.7 Converting a PDF Document To Image

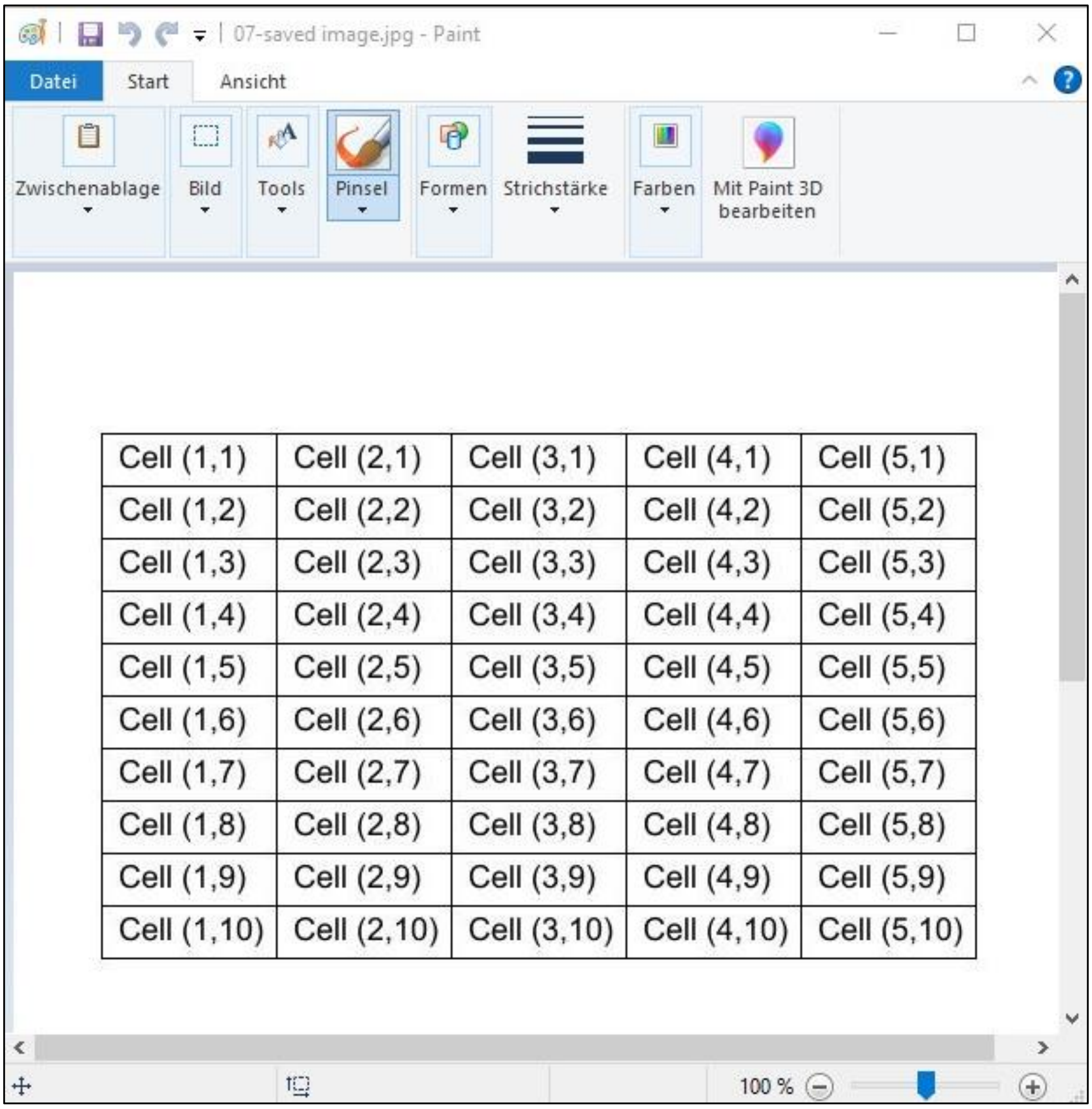
This Example demonstrates how to save an existing PDF document as an image file.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "06-table with content.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

-- use the renderer to buffer the document as an image
renderer=.bsf~new("org.apache.pdfbox.rendering.PDFRenderer",doc)
image=renderer~renderImage(0)

-- write the buffered image to the destination file
writer=bsf.import("javax.imageio.ImageIO")
file=.bsf~new("java.io.File","07-saved image.jpg")
writer~write(image, "jpeg", file)

-- get java support
::requires "BSF.CLS"
```

Figure 14: *"07. Converting a PDF Document To Image.rex"*

As one can see the code for this example is very simple. The Java class "Loader" is used to import the existing PDF document. Then, the Java class "PDFRenderer" is imported to save the whole page as an image to the buffer. For saving the Image file there are a few steps to do. First, the Java class "ImageIO" has to be imported. Then, the Java class "io.File" is used to create an empty image file. The last step is to use the method "~write" from the class "ImageIO" for writing the buffered image to the created image file.

The figure below shows a screenshot of the resulting image file opened with Microsoft Paint.



Cell (1,1)	Cell (2,1)	Cell (3,1)	Cell (4,1)	Cell (5,1)
Cell (1,2)	Cell (2,2)	Cell (3,2)	Cell (4,2)	Cell (5,2)
Cell (1,3)	Cell (2,3)	Cell (3,3)	Cell (4,3)	Cell (5,3)
Cell (1,4)	Cell (2,4)	Cell (3,4)	Cell (4,4)	Cell (5,4)
Cell (1,5)	Cell (2,5)	Cell (3,5)	Cell (4,5)	Cell (5,5)
Cell (1,6)	Cell (2,6)	Cell (3,6)	Cell (4,6)	Cell (5,6)
Cell (1,7)	Cell (2,7)	Cell (3,7)	Cell (4,7)	Cell (5,7)
Cell (1,8)	Cell (2,8)	Cell (3,8)	Cell (4,8)	Cell (5,8)
Cell (1,9)	Cell (2,9)	Cell (3,9)	Cell (4,9)	Cell (5,9)
Cell (1,10)	Cell (2,10)	Cell (3,10)	Cell (4,10)	Cell (5,10)

Figure 15: Output of *"07. Converting a PDF Document To Image.rex"*

4.8 Inserting Image to a PDF Document

In previous example the converting from PDF to image is showed. This example will demonstrate how to insert an existing image to a PDF document.

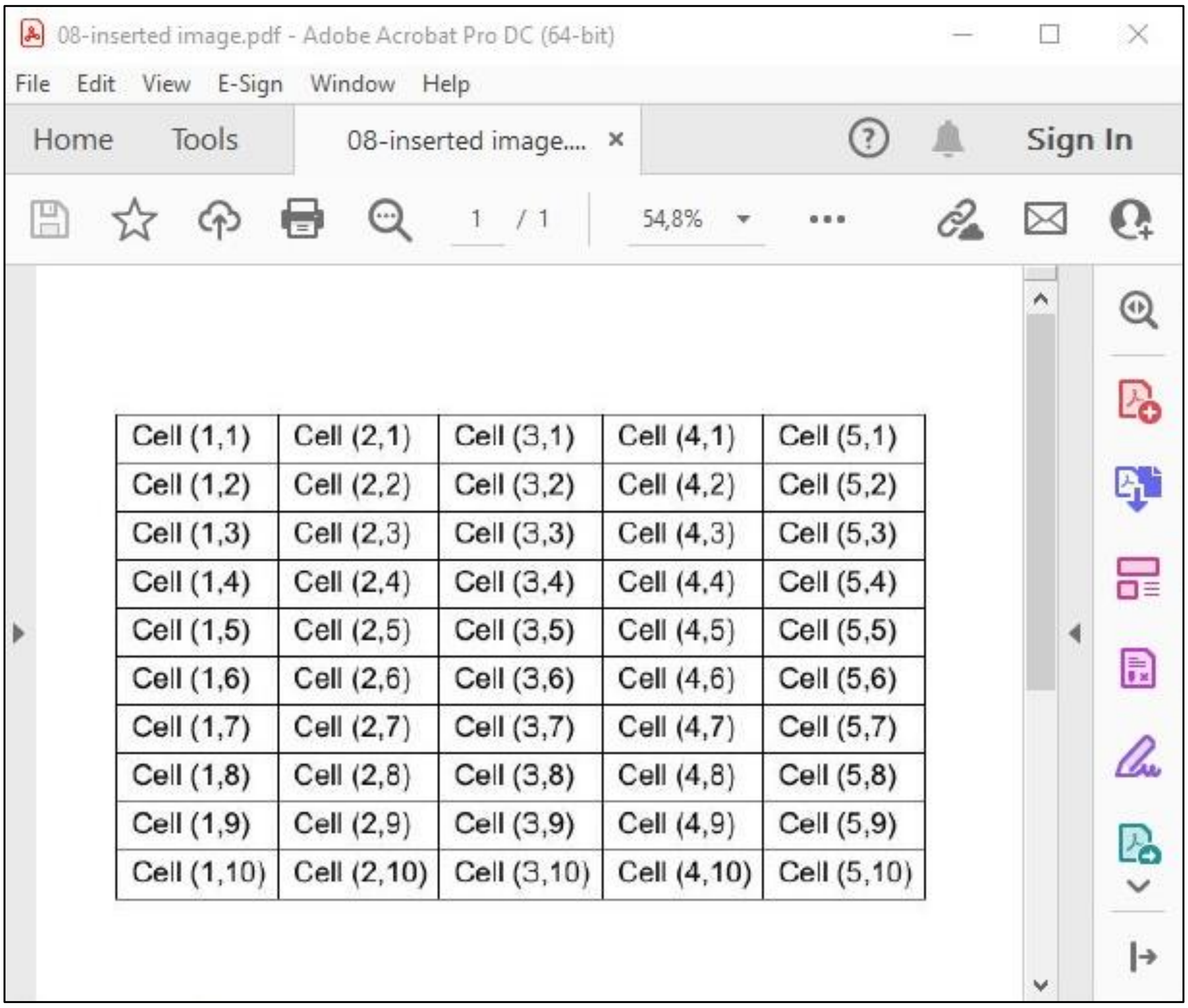
```
-- change directory to program location
parse source   . . pgm
call directory filespec('L', pgm)

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

-- import the image file
xclass = "org.apache.pdfbox.pdmodel.graphics.image.PDImageXObject"
xobject=BSF.loadClass(xclass)
image=xobject~createFromFile("07-saved image.jpg",doc)

-- create a content stream
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)

-- use the content stream to insert content
cont~drawImage(image,0,0)
cont~close

-- save and close the document file
doc~save("08-inserted image.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 16: *"08. Inserting Image to a PDF Document.rex"*

After creating the new document and a new blank page, the Java class "PDImageXObject" is imported to handle the image file we want to insert. The method "~createFromFile" is used to create an image object that can be used for further operations. To use this method, we need the path of the desired image file and the name of document object we created.

After the content stream is prepared, the method "~drawImage" is used to insert the image object we created from the original image file. With that done, the document can be saved and closed.

The following figure shows the resulting document with the image inserted.


08-inserted image.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 08-inserted image....
Sign In
1 / 1
54,8%

Cell (1,1)	Cell (2,1)	Cell (3,1)	Cell (4,1)	Cell (5,1)
Cell (1,2)	Cell (2,2)	Cell (3,2)	Cell (4,2)	Cell (5,2)
Cell (1,3)	Cell (2,3)	Cell (3,3)	Cell (4,3)	Cell (5,3)
Cell (1,4)	Cell (2,4)	Cell (3,4)	Cell (4,4)	Cell (5,4)
Cell (1,5)	Cell (2,5)	Cell (3,5)	Cell (4,5)	Cell (5,5)
Cell (1,6)	Cell (2,6)	Cell (3,6)	Cell (4,6)	Cell (5,6)
Cell (1,7)	Cell (2,7)	Cell (3,7)	Cell (4,7)	Cell (5,7)
Cell (1,8)	Cell (2,8)	Cell (3,8)	Cell (4,8)	Cell (5,8)
Cell (1,9)	Cell (2,9)	Cell (3,9)	Cell (4,9)	Cell (5,9)
Cell (1,10)	Cell (2,10)	Cell (3,10)	Cell (4,10)	Cell (5,10)

Figure 17: Output of *"08. Inserting Image to a PDF Document.rex"*

4.9 Adding Multiple Pages to a PDF Document

In the previous examples all the documents only contain one single page. PDFbox does certainly support multi-page documents. This example shows how to add contents to multiple pages within a single document.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- create a new document
doc=.bsf~new("org.apache.pdfbox.pdmodel.PDDocument")

-- start the loop for each page
num= 3
do i=1 to num

-- add a blank page the document
page.i=.bsf~new("org.apache.pdfbox.pdmodel.PDPage")
doc~addPage(page.i)

-- create a content stream
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page.i)

-- define font type
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert content
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
cont~newLineAtOffset( 100, 700 )
cont~showText("This is Page" i)
cont~newLine
cont~newLine
cont~newLine
cont~showText("This is the first line")
cont~newLine
cont~newLine
cont~showText("This is the second line")
cont~newLine
cont~newLine
cont~showText("This is the third line")
```

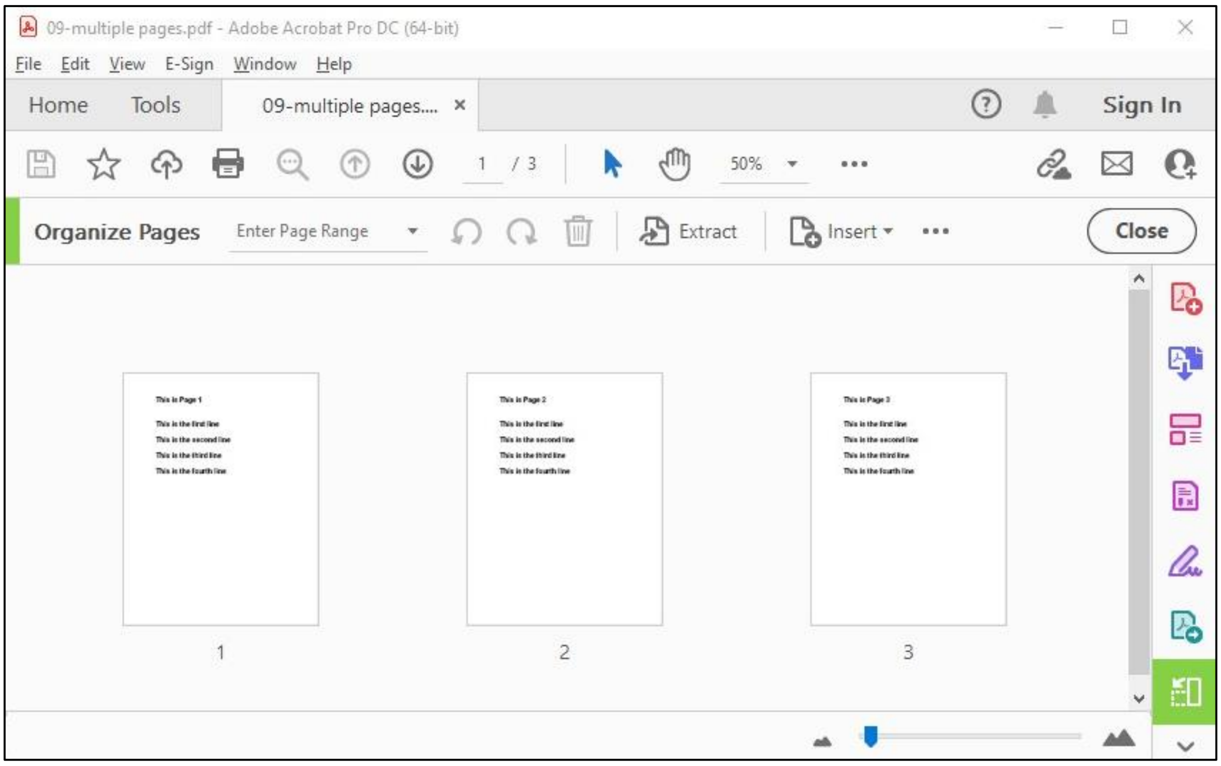
```
cont~newLine
cont~newLine
cont~showText("This is the fourth line")
cont~endText
cont~close
end

-- save and close the document file
doc~save("09-multiple pages.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 18: *"09. Adding Multiple Pages to a PDF Document.rex"*

After the document is created, we want to add several pages according to the goal of this example. In this code a do loop is used to simplify the code. For demonstration the number of loops is set to 3. It is of course possible to use another value instead or just do it manually without the loop function.

For each loop a page with own index is created and added to document. It is necessary to create a content stream for each of the pages. After inserting the desired content, the content stream is closed. After all loops are done, the document can be saved and closed.

The following figure shows the resulting multi-pages document.


09-multiple pages.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 09-multiple pages.... ×
Sign In
1 / 3
50%
Organize Pages Enter Page Range Extract Insert Close
1
2
3

This is Page 1

This is the first line
This is the second line
This is the third line
This is the fourth line

This is Page 2

This is the first line
This is the second line
This is the third line
This is the fourth line

This is Page 3

This is the first line
This is the second line
This is the third line
This is the fourth line

Figure 19: Output of *"09. Adding Multiple Pages to a PDF Document.rex"*

4.10 Splitting a PDF Document with Multiple Pages

The last example showed how to create an PDF document with multiple pages. This example demonstrates how to split a document containing several pages into separate documents.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "09-multiple pages.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

-- use the splitter to split the document
spl=.bsf~new("org.apache.pdfbox.multipdf.Splitter")
pages=spl~split(doc)

-- create an iterator to navigate through the pages
iterator=Pages~listIterator

-- save each page as a new document
do i= 1 to pages~size
pd=iterator~next
pd~save("10-split page"||i||".pdf")
pd~close
end

-- get java support
::requires "BSF.CLS"
```

Figure 20: *"10. Splitting a PDF Document with Multiple Pages.rex"*

The first thing to do is to import a document with several pages. The resulting document of the last example is used for that. For splitting the document a new instance of the Java class "org.apache.pdfbox.multipdf.Splitter" is created. First, the document is split into a list of separate documents by using the method "~split". Next, the method "~listIterator" is used to create an iterator for pointing every single document from the list. We use a loop statement to save those documents. The first step is to check the number of documents created, which defines the number of loops to run. The next step is to create the loops, in which we use the method "~next" to select a single document and save it. After the loops are done, every page from the original documents is saved as a new separate document.

The following figure shows the resulting documents.

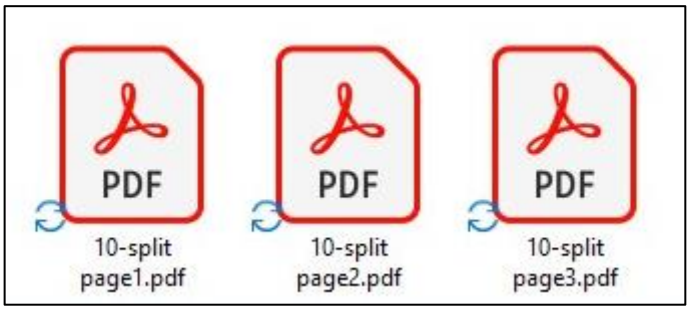


Figure 21: Output of *"10. Splitting a PDF Document with Multiple Pages.rex"*

4.11 Merging Multiple PDF Documents

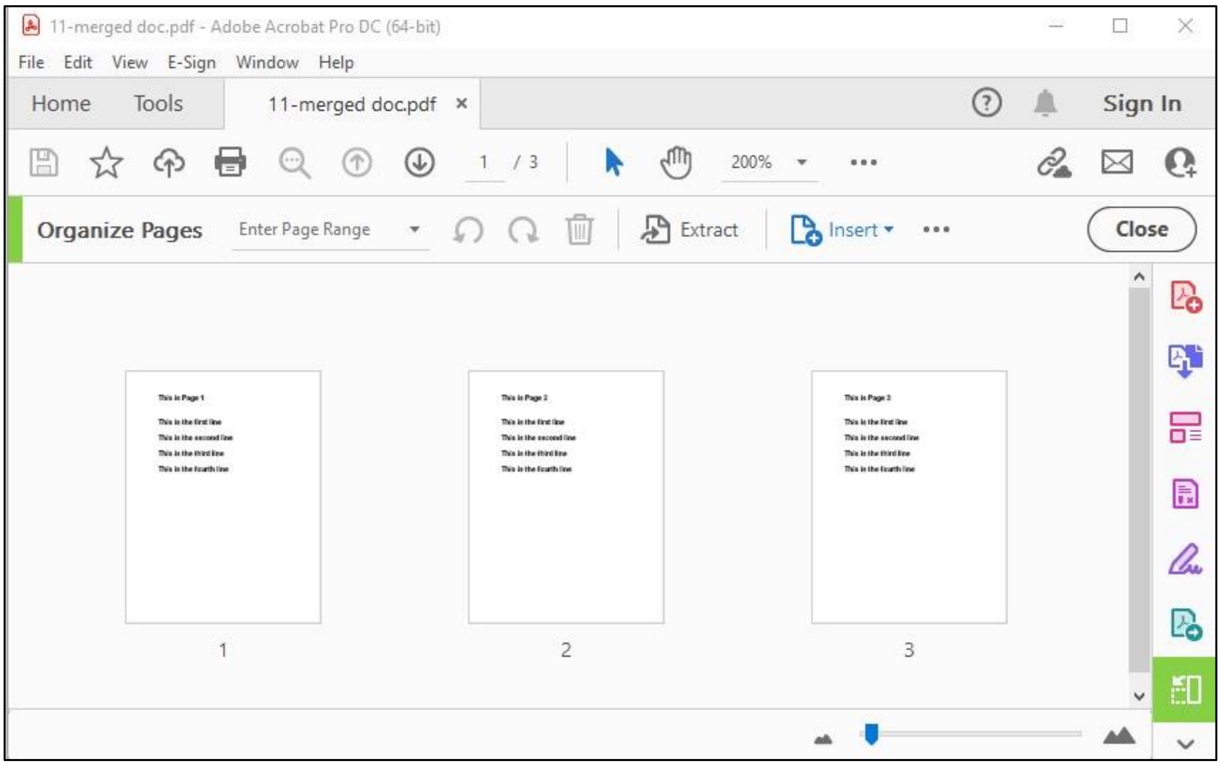
In the last example the splitting of a documents containing several pages is showed. This example will demonstrate the reverse process, namely the merging of several documents into a single one.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source files
file1=.bsf~new("java.io.File", "10-split page1.pdf")
file2=.bsf~new("java.io.File", "10-split page2.pdf")
file3=.bsf~new("java.io.File", "10-split page3.pdf")

-- import the merger
merger=.bsf~new("org.apache.pdfbox.multipdf.PDFMergerUtility")

-- set the destination file
merger~setDestinationFileName("11-merged doc.pdf")

-- add the source files and save as a new document
merger~addSource(file1)
merger~addSource(file2)
merger~addSource(file3)
mus=BSF.loadClass("org.apache.pdfbox.io.MemoryUsageSetting")
merger~mergeDocuments(mus~setupMainMemoryOnly)

-- get java support
::requires "BSF.CLS"
```

Figure 22: *"11. Merging Multiple PDF Documents.rex"*

The first thing to do is to use the Java class "java.io.File" to provide access to the source files. Next, a new instance of the Java class "PDFMergerUtility" needs to be created. The file name of the new document is set by the method "~setDestinationFileName". Then, the method "~addSource" is used to import source file we want to merge. This needs to be done for every file. Next, the memory usage setting needs to be defined. Therefore, the Java class "MemoryUsageSetting" is imported and set to "~setupMainMemoryOnly". The last step is to use the method "~mergeDocuments" with the defined memory usage setting to merge all the imported documents.

The following figure shows a screenshot of the resulting document opened with Adobe Acrobat Pro.


11-merged doc.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 11-merged doc.pdf ×
Sign In
1 / 3
200%
Organize Pages
Enter Page Range
Extract
Insert
Close
1
2
3

This is Page 1
This is the first line
This is the second line
This is the third line
This is the fourth line

This is Page 2
This is the first line
This is the second line
This is the third line
This is the fourth line

This is Page 3
This is the first line
This is the second line
This is the third line
This is the fourth line

Figure 23: Output of "*11. Merging Multiple PDF Documents.rex*"

4.12 Setting the Document Metadata

This example demonstrates how to set or modify the metadata of a PDF document. Metadata consists information about the document itself such as Author, Title, Creation date, Subject…

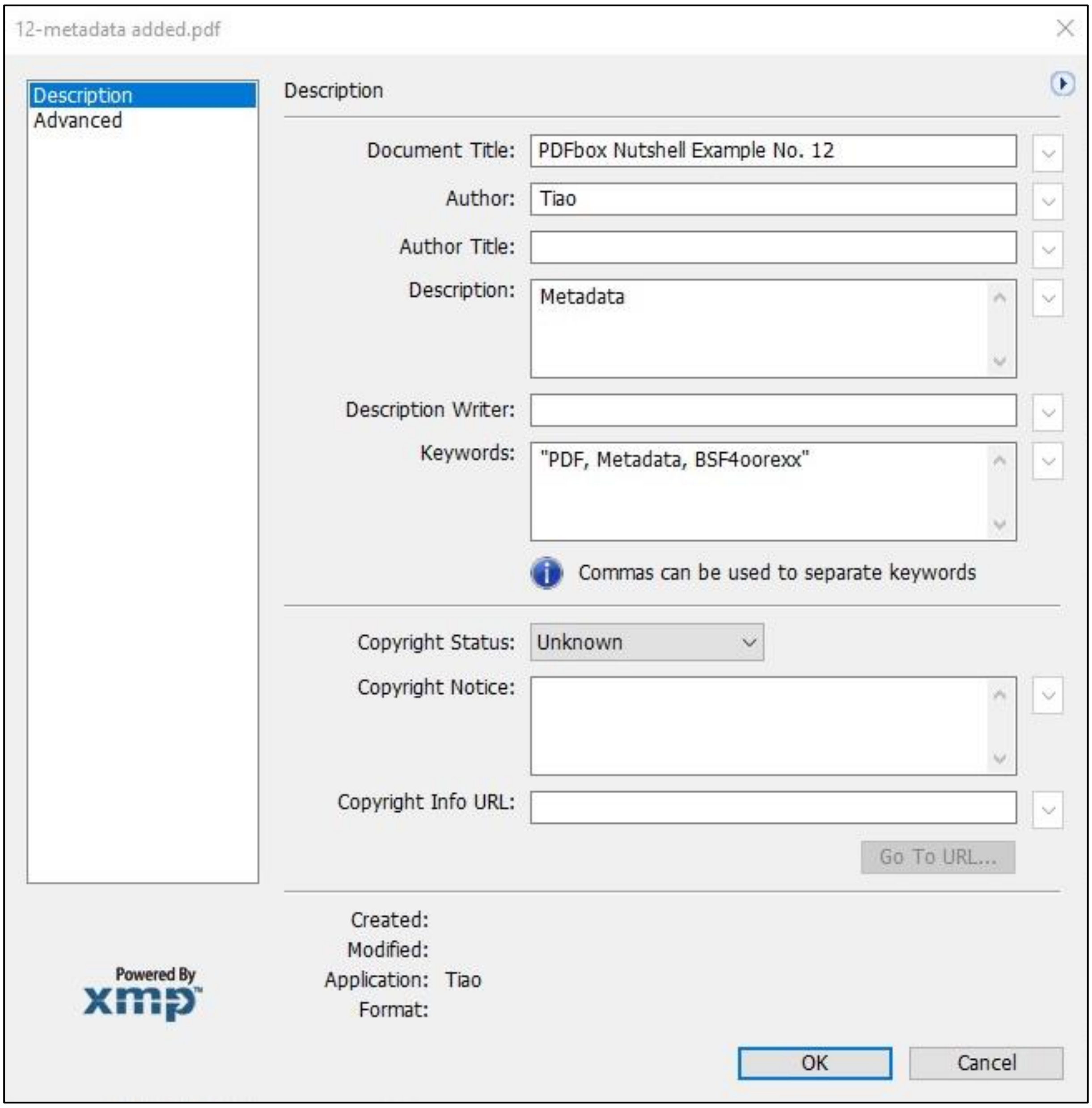
```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "11-merged doc.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

-- load the document information and set different metadata
info=doc~getDocumentInformation
info~setAuthor("Tiao")
info~setTitle("PDFbox Nutshell Example No. 12")
info~setCreator("Tiao")
info~setSubject("Metadata")
info~setKeywords("PDF, Metadata, BSF4oorexx")

-- save and close the document file
doc~save("12-metadata added.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 24: *"12. Setting the Document Metadata.rex"*

With PDFbox it is very easy to edit the metadata of a document. First, a document is loaded using the Java class "org.apache.pdfbox.Loader". Then, the method "~getDocumentInformation" is used to load the metadata of the document. Now it is possible to modify it. In this example these following methods are used:

- ~setAuthor: set the author of the document
- ~setTitle: set the title of the document
- ~setCreator: set the Creator of the document
- ~setSubject: set the Subject of the document
- ~setKeywords: set the Keywords of the document

After setting the metadata the document is saved and closed. The following figure shows a screenshot of the modified metadata.


12-metadata added.pdf

Document Title:	PDFbox Nutshell Example No. 12
Author:	Tiao
Author Title:	
Description:	Metadata
Description Writer:	
Keywords:	"PDF, Metadata, BSF4oorexx"

Commas can be used to separate keywords

Copyright Status:	Unknown
Copyright Notice:	
Copyright Info URL:	

Created:	
Modified:	
Application:	Tiao
Format:	

Figure 25: Output of "*12. Setting the Document Metadata.rex*"

4.13 Adding Watermark to a Document

This example shows how to create a watermark on every page of a document.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "12-metadata added.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

-- count the number of pages
pages=doc~getDocumentCatalog~getPages

-- start the loop for each page
do i= 1 to doc~getNumberOfPages
page=pages~get(i-1)

-- set the append mode
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
append=BSF.loadClass(contclass)~AppendMode~APPEND

-- create a content stream
cont=.bsf~new(contclass,doc,page,append,"true")

-- define font type

fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert content
cont~beginText
cont~setLeading(25f)
cont~setFont(font, 70)

-- set the transparency of the watermark
gsclass="org.apache.pdfbox.pdmodel.graphics.state.PDExtendedGraphicsState"
gs=.bsf~new(gsclass)
gs~setNonStrokingAlphaConstant(0.2)
cont~setGraphicsStateParameters(gs)

-- set the color of the watermark
col=BSF.loadClass("java.awt.Color")~red
```

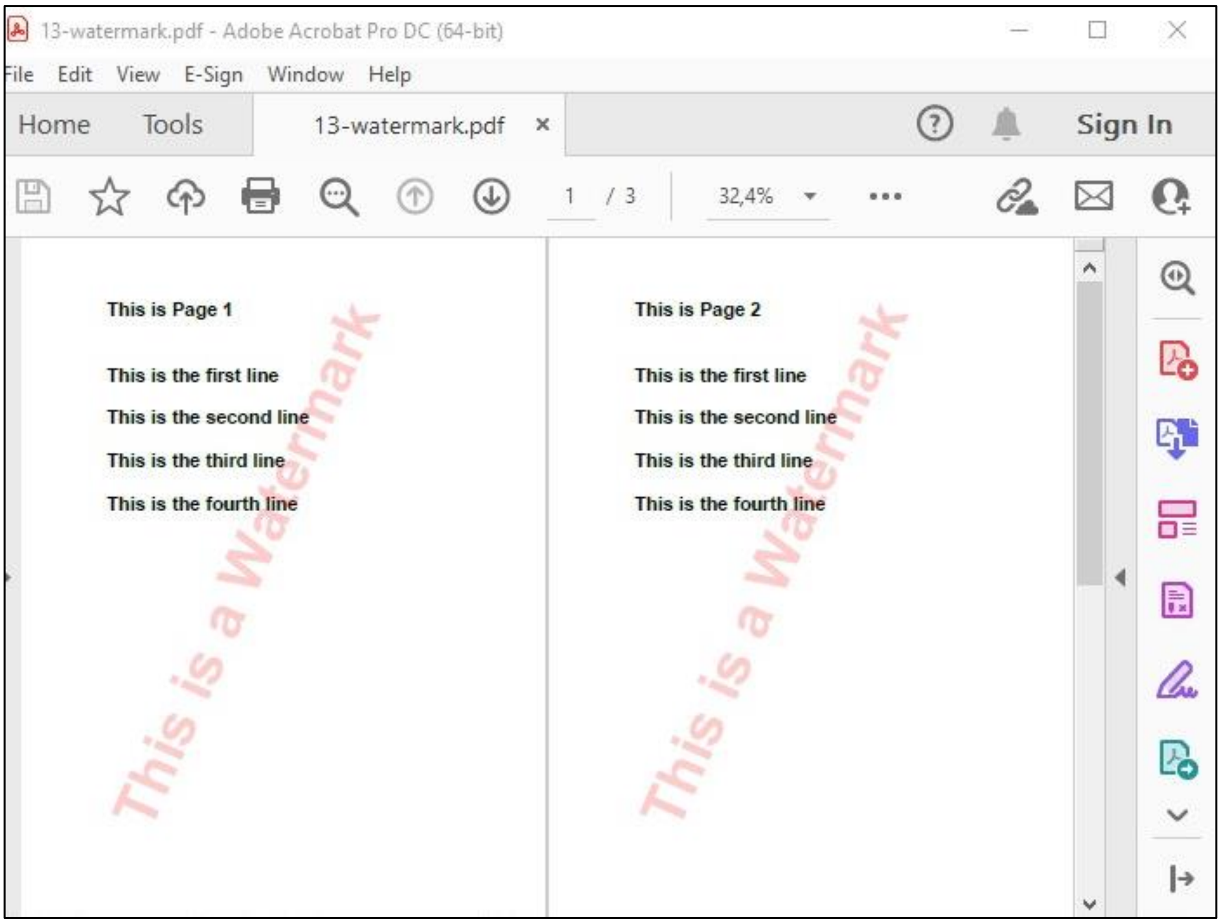
```
cont~setNonStrokingColor(col)

-- set the rotation of the watermark
matrix=.bsf~new("org.apache.pdfbox.util.Matrix")
cont~setTextMatrix(matrix~getRotateInstance(20,150,100))

-- insert the watermark
cont~showText("This is a Watermark")
cont~endText
cont~close
end

-- save and close the document file
doc~save("13-watermark.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 26: *"13. Adding Watermark to a Document.rex"*

The first thing to do is to import a document with several pages. We can use the resulting document of last example. Next, the number of pages is counted to determine how many loops are needed to add the watermark. In this example, we will use a short text as watermark.

Within the loop a content stream is created for the current page. To avoid overwriting existing content the append mode must be activated. After setting the font type the content stream can be started with the method "~beginText". The next thing to do is to set the transparency of the watermark. Therefore, the Java class "PDExtendedGraphicsState" needs to be imported. The alpha constant, which defines the transparency, is modified by the method "~setNonStrokingAlphaConstant". This setting needs to be saved to the content stream using the method "~setGraphicsStateParameters". Then, the font color of the text is set to red and the font size to 70. The Java class "org.apache.pdfbox.util.Matrix" is imported to rotate the watermark. Therefore, the method "getRotateInstance" is used to define the rotation parameters. The first parameter is the rotation angle and the next two parameters are coordinates of the rotation point. This rotation setting needs to be saved to the content stream as well. The last step of the loop is to insert the text as watermark and close the content stream.

After all loops are done, the document can be saved and closed. The following figure shows the resulting document.


13-watermark.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 13-watermark.pdf
Sign In
1 / 3
32,4%

This is Page 1

This is the first line

This is the second line

This is the third line

This is the fourth line

This is a Watermark

This is Page 2

This is the first line

This is the second line

This is the third line

This is the fourth line

This is a Watermark

Figure 27: Output of "*13. Adding Watermark to a Document.rex*"

4.14 Encrypting a PDF Document

This example illustrates how to secure a PDF Document by setting the user access permissions and encrypting the whole document.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "13-watermark.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(source)

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

-- define the standard protection policyencrypt the document
sppclass = "org.apache.pdfbox.pdmodel.encryption.StandardProtectionPolicy"
spp=.bsf~new(sppclass,"ownerpw","userpw",ap)
spp~setEncryptionKeyLength(128)
spp~setPermissions(ap)

--encrypt the document
doc~protect(spp)

-- save and close the document file
doc~save("14-encrypted.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 28: *"14. Encrypting a PDF Document.rex"*

The first thing to do is to create or load a PDF document, which should be secured. In this example the resulting document of last example is used to demonstrate how the securing process works.

After the document is loaded, the first part of the securing process can be started. A standard PDF document can be opened, copied, printed or modified by anyone. To avoid that, the Java class "org.apache.pdfbox.pdmodel.encryption.AccessPermission" is imported to define access permissions for the user. The following methods are used to define if the user has certain permission to the document.

- ~setcanModify: Permission to modify the document.
- ~setCanExtractContent: Permission to extract content from the document.
- ~setCanPrint: Permission to print the document.

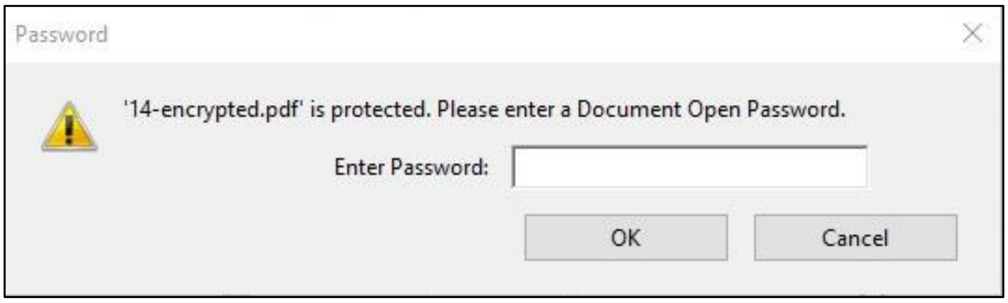
In this example these permissions are set to false, which means that the user is not allowed to process these operations.

The next part of the securing process is to encrypt the document and adopt the access permission settings we defined in the first part of the process. Therefore, the Java class "org.apache.pdfbox.pdmodel.encryption.StandardProtectionPolicy" needs to be imported. This class requires three parameters, the first two are owner and user password, the third parameter is the access permission setting. Owner password and user password are strings that can be chosen freely. With the user password the document can be opened and all operations, that aren't forbidden by the defined access permissions, can be processed. The owner password is needed to gain full access to the document and is not subject to any restriction.

The next step is to set the length of the secret key used to encrypt the document by using the method "~setEncryptionKeyLength". The access permission setting needs to be activated by the method "~setPermissions". Finally, the document is encrypted by the method "~protect".

After the securing process the document can be saved and closed.

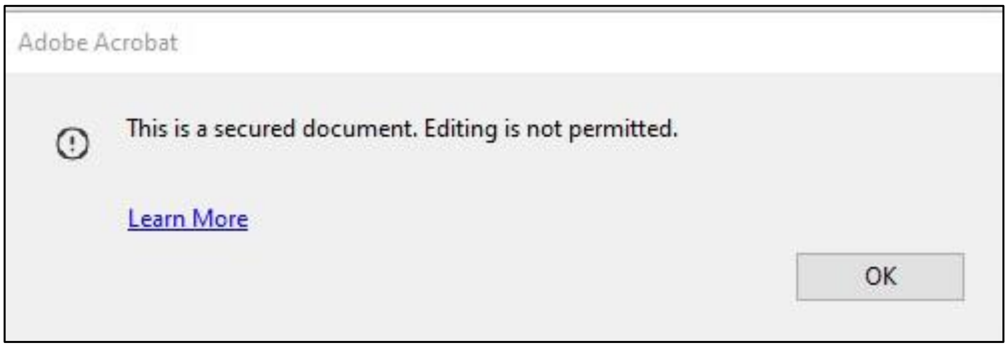
The following figures will show the effects of the securing process.



'14-encrypted.pdf' is protected. Please enter a Document Open Password.

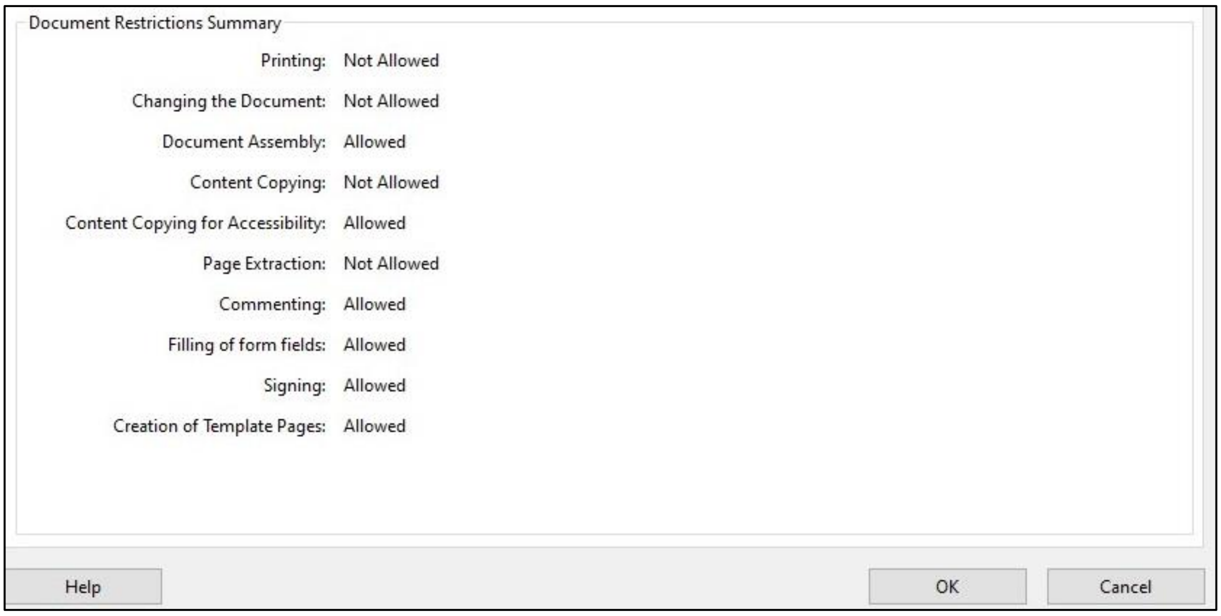
Enter Password:

Figure 29: Output 1 of "*14. Encrypting a PDF Document.rex*"



This is a secured document. Editing is not permitted.

Figure 30: Output 2 of "*14. Encrypting a PDF Document.rex*"


Document Restrictions Summary

Printing:	Not Allowed
Changing the Document:	Not Allowed
Document Assembly:	Allowed
Content Copying:	Not Allowed
Content Copying for Accessibility:	Allowed
Page Extraction:	Not Allowed
Commenting:	Allowed
Filling of form fields:	Allowed
Signing:	Allowed
Creation of Template Pages:	Allowed

Figure 31: Output 3 of *"14. Encrypting a PDF Document.rex"*

4.15 Creating a PDF-A Document

This example demonstrates how to create a PDF/A document. PDF/A is a specialized version of the Portable Document Format (PDF) used for long-term archiving of digital contents and is standardized by the International Organization for Standardization (ISO).

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

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

-- load the font file
fontfile=.bsf~new("java.io.File", "resources\arial.ttf")
fontclass = "org.apache.pdfbox.pdmodel.font.PDType0Font"
font=BSF.loadClass(fontclass)~load(doc,fontfile)

-- set the xmp metadata
xmp=BSF.loadClass("org.apache.xmpbox.XMPMetadata")~createXMPMetadata
dc=xmp~createAndAddDublinCoreSchema
dc~setTitle("15-PDFa.pdf");
id=xmp~createAndAddPDFAIdentificationSchema
id~setPart(1);
id~setConformance("B");
serializer=.bsf~new("org.apache.xmpbox.xml.XmpSerializer")
baos=.bsf~new("java.io.ByteArrayOutputStream")
serializer~serialize(xmp, baos,"true")
metadata=.bsf~new("org.apache.pdfbox.pdmodel.common.PDMetadata",doc)
metadata~importXMPMetadata(baos~toByteArray)
doc~getDocumentCatalog~setMetadata(metadata)

-- define the color profile
colorprofile=.bsf~new("java.io.FileInputStream", "resources\sRGB.icc")
intentclass = "org.apache.pdfbox.pdmodel.graphics.color.PDOutputIntent"
intent=.bsf~new(intentclass,doc,colorprofile)
intent~setInfo("sRGB IEC61966-2.1")
intent~setOutputCondition("sRGB IEC61966-2.1")
intent~setOutputConditionIdentifier("sRGB IEC61966-2.1")
intent~setRegistryName("http://www.color.org")
doc~getDocumentCatalog~addOutputIntent(intent)

```

```
-- create a content stream
cont=.bsf~new("org.apache.pdfbox.pdmodel.PDPageContentStream",doc,page)

-- use the content stream to insert content
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
cont~newLineAtOffset( 100, 700 )
cont~showText("This is a PDF/A Document")
cont~endText
cont~close
nocclass="org.apache.pdfbox.pdfwriter.compress.CompressParameters"
noc=BSF.loadClass(nocclass)

-- save and close the document file
doc~save("15-PDFa.pdf",noc~NO_COMPRESSION)
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 32: *"15. Creating a PDF-A Document.rex"*

The first thing to do is to create a new PDF document and add a blank page to it. The PDF/A format has some requirements that need to be fulfilled. The first requirement concerns the fonts. All fonts used in the document must be embedded in the file, because of this the usage of the build-in standard font types is not suitable. For this example, the font file "arial.ttf" has been downloaded and put into the folder "resources". The Java class "java.io.File" is used to provide access to the file and the class "org.apache.pdfbox.pdmodel.font.PDType0Font" is imported to load the font file.

Another requirement of the PDF/A format is to have metadata defined in the document. The tricky part here is the fact, that the use of the ISO-standardized metadata format Extensible Metadata Platform (XMP) is required. The PDFbox library does not support this format by default, so the sub-library xmpbox needs to be implemented to handle XMP metadata. After this is done, the Java class "org.apache.xmpbox.XMPMetadata" can be imported to provide XMP metadata support. First, the method "~createXMPMetadata" is used to create a new XMP metadata. A PDF/A document requires at least two entries in the metadata, the title and the PDF/A version of the document. For these two entries we must use different standardized schemas to define them. The title requires the use of the Dublin Core Metadata Element Set, which is done by using the methods "~createAndAddDublinCoreSchema" and "~setTitle". The PDF/A Identification Schema defines the entries, which indicate that the file is a PDF/A document and its version. Therefore, the methods "~createAndAddPDFAIdentificationSchema", "~setPart" and "~setConformance" are used. In this example, the PDF/A basic version 1B is used. The next step is to serialize the XMP metadata. Therefore, the Java class "org.apache.xmpbox.xml.XmpSerializer" is imported. For the serializing process a destination output stream is needed, which can be created by importing the Java class "java.io.ByteArrayOutputStream". After the serializing process the XMP metadata as an output stream can be imported to the document. First, the Java class "org.apache.pdfbox.pdmodel.common.PDMetadata" is imported and the method "~importXMPMetadata" is used to import the XMP metadata we created. Finally, the metadata is implemented by the method "~setMetadata".

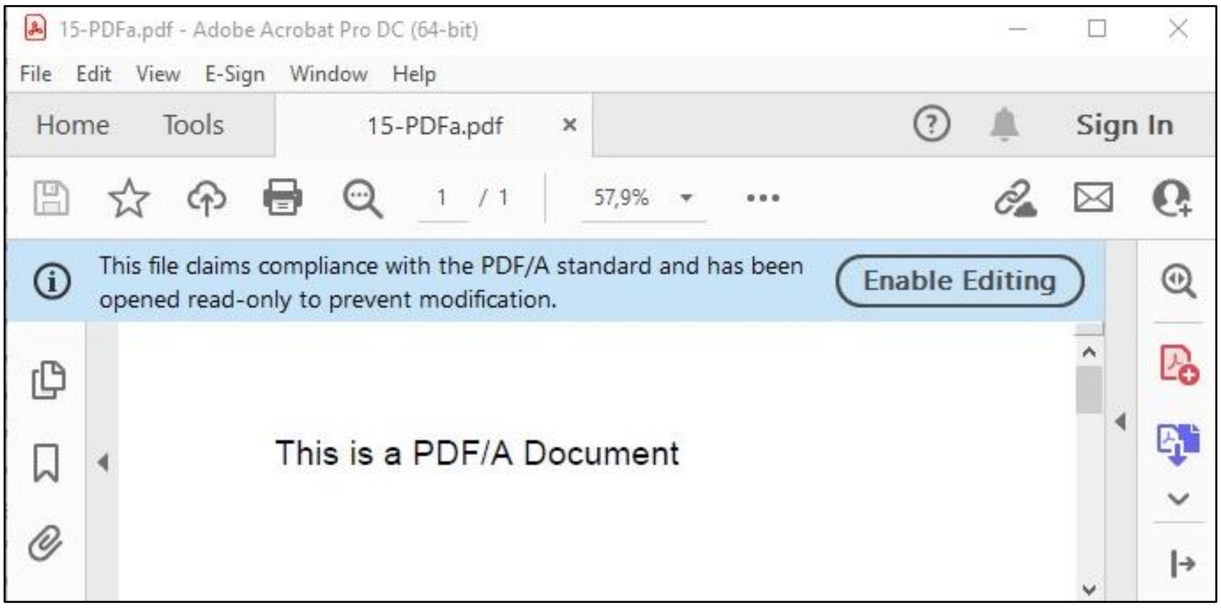
The last requirement of PDF/A standard is to include the color space profile used into the document. For this example, the color space profile "sRGB.icc" is used and saved to the resources folder. First, the color profile is converted to an input stream by importing the Java class "java.io.FileInputStream". Next, the Java class "org.apache.pdfbox.pdmodel.graphics.color.PDOutputIntent" is imported to create an output intent using the converted color profile. The color profile needs to be defined in document by the following methods:

- ~setInfo
- ~setOutputCondition
- ~setOutputConditionIdentifier
- ~setRegistryName

After inserting these entries the color profile can be implemented by the method "~addOutputIntent".

The next step is to create a content stream and insert some text in the document. Then, the document can be saved and closed.

The following figure shows the resulting document.


15-PDFa.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 15-PDFa.pdf
Sign In
1 / 1 57,9%
Enable Editing

This file claims compliance with the PDF/A standard and has been opened read-only to prevent modification.

This is a PDF/A Document

Figure 33: Output of *"15. Creating a PDF-A Document.rex"*

4.16 Validating a PDF-A Document

This example shows how to validate an existing PDF/A document.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file
file="15-PDFa.pdf"

-- use the parser to validate the document and save the result
pclass = "org.apache.pdfbox.preflight.parser.PreflightParser"
parser=.bsf~new(pclass,file)
doc=parser~parse
if doc~validate~isValid=1
then content=file " is a valid PDF/A-1b document"
else content=file " is not a valid PDF/A-1b document"

-- create a new document 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
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)

-- define font type
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert validation result
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
cont~newLineAtOffset(100, 700)
cont~showText(content)
cont~endText
cont~close

-- save and close the document file
doc~save("16-validation result.pdf")
doc~close

```

```
-- get java support
::requires "BSF.CLS"
```

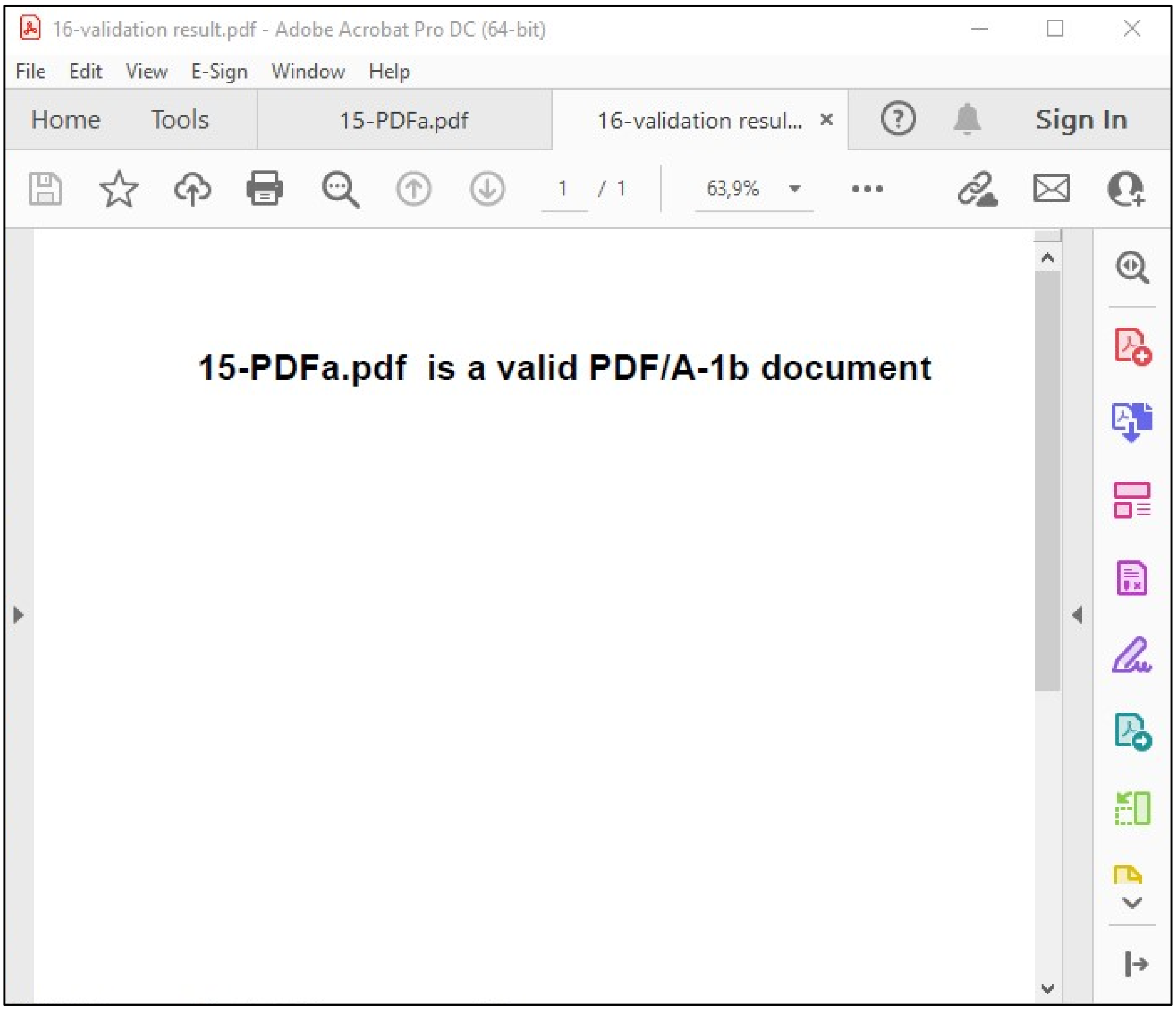
Figure 34: "*16. Validating a PDF-A Document.rex*"

The PDFbox library doesn't support PDF/A validation by default. The sub-library preflight needs to be implemented.

First, the Java class "org.apache.pdfbox.preflight.parser.PreflightParser" is imported and the method "~parse" is used to read the source document. Next, the method "~validate" is used to validate, whether the given document meets the requirements of the PDF/A standard. The method "~isValid" is used to read the validation result. If the value 1 is returned, then the document is a valid PDF/A document. Otherwise the document doesn't meet the requirements.

The next step is to save the validation result to a new document. Therefore, a new PDF document is created, and a blank page is added. Then, a content stream is created to insert the validation result. In the end, the document is saved and closed.

The following figure shows the resulting document.

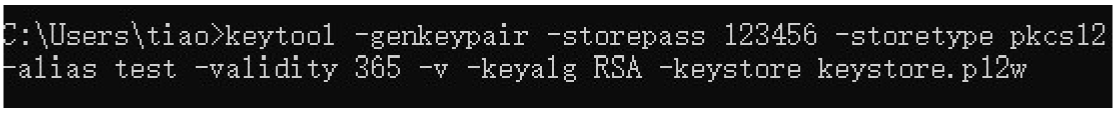

16-validation result.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 15-PDFa.pdf 16-validation resul... Sign In
1 / 1 63,9%

15-PDFa.pdf is a valid PDF/A-1b document

Figure 35: Output of "*16. Validating a PDF-A Document.rex*"

4.17 Creating a Digital Signature

This example demonstrates how to create a digital signature with PDFbox. The signing process is done with bouncy castle, which is a Java library for cryptography. Bouncy castle is included in the standalone version of PDFbox Library. For the signing process a keystore is needed, it can be created with keytool using command line. Keystore contains public keys, private keys and certificates, which are used during the signing process. In this example a keystore is already created and saved in the resources folder. The following figure shows the command used for the keystore creation.



```
C:\Users\tiao>keytool -genkeypair -storepass 123456 -storetype pkcs12
-alias test -validity 365 -v -keyalg RSA -keystore keystore.p12w
```

Figure 36: Keystore creation

The following parameters are used:

- -genkeypair: Generates a key pair
- -storepass: Sets the password of the keystore
- -storetype: Sets the type of the keystore
- -alias: Sets the name of the keystore
- -validity: Sets the validity time (in days) of the keystore
- -v: Verbose Mode
- -keyalg: Sets algorithm used for the keystore
- -keystore: Sets the file name of the keystore

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
infile=.bsf~new("java.io.File", "16-validation result.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
doc=importdoc~loadPDF(infile)

-- configure the signature settings
sigc="org.apache.pdfbox.pdmodel.interactive.digitalsignature.PDSignature"
sig=.bsf~new(sigc)
filter=BSF.loadClass(sigc)~FILTER_ADOBE_PPKLITE
sf=BSF.loadClass(sigc)~SUBFILTER_ADBE_PKCS7_DETACHED
sig~setFilter(filter)
sig~setSubFilter(sf)
sig~setName("Tiao Wang")
sig~setLocation("Vienna, Austria")
sig~setReason("Testing")
cal=BSF.loadClass("java.util.Calendar")~getInstance
```

```
sig~setSignDate(cal)
doc~addSignature(sig)

-- prepare the signature generator
password=.bsf~new("java.lang.String", "123456")~toCharArray
keystore=BSF.loadClass("java.security.KeyStore")~getInstance("PKCS12")
ksfis=.bsf~new("java.io.FileInputStream", "resources\keystore.p12")
keystore~load(ksfis,password)
pk=keystore~getKey(test,password)
certchain=keystore~getCertificateChain(test)
certList=BSF.loadClass("java.util.Arrays")~asList(certchain)
cert =certlist~get(0)
certstore=.bsf~new("org.bouncycastle.cert.jcajce.JcaCertStore",certlist)

-- create the signature generator
gen=.bsf~new("org.bouncycastle.cms.CMSSignedDataGenerator")
sbclass="org.bouncycastle.operator.jcajce.JcaContentSignerBuilder"
sha1signer=.bsf~new(sbclass,"SHA256WithRSA")~build(pk)
pbclass="org.bouncycastle.operator.jcajce.JcaDigestCalculatorProviderBuilder"
calc=.bsf~new(pbclass)~build
igclass="org.bouncycastle.cms.jcajce.JcaSignerInfoGeneratorBuilder"
infogen=.bsf~new(igclass,calc)~build(sha1signer, cert)
gen~addSignerInfoGenerator(infogen)
gen~addCertificates(certstore)

-- set the destination file
outfile=.bsf~new("java.io.File", "17-signed.pdf")
fos=.bsf~new("java.io.FileOutputStream", outfile)
exsign=doc~saveIncrementalForExternalSigning(fos)

-- convert the signing data to the cms format
x=exsign~getContent
io=BSF.loadClass("org.apache.commons.io.IOUtils")
ba= io~toByteArray(x)
cmsdata=.bsf~new("org.bouncycastle.cms.CMSProcessableByteArray",ba)

-- generate the signature
signed=gen~generate(cmsdata,"false")
cmssig=signed~getEncoded

-- add the signature to the document
exsign~setSignature(cmssig)

-- get java support
::requires "BSF.CLS"
```

Figure 37: *"17. Creating a Digital Signature.rex"*

The first step is to load an existing document that need to be signed. In this example, the resulting document of the last example is used. The next step is to create and configure the signature. Therefore, a new instance of the Java class "org.apache.pdfbox.pdmodel.interactive.digitalsignature.PDSignature" is created. For configuring the signature, the following methods are used:

- ~setFilter
- ~setSubFilter
- ~setName
- ~setLocation
- ~setReason
- ~setSignDate

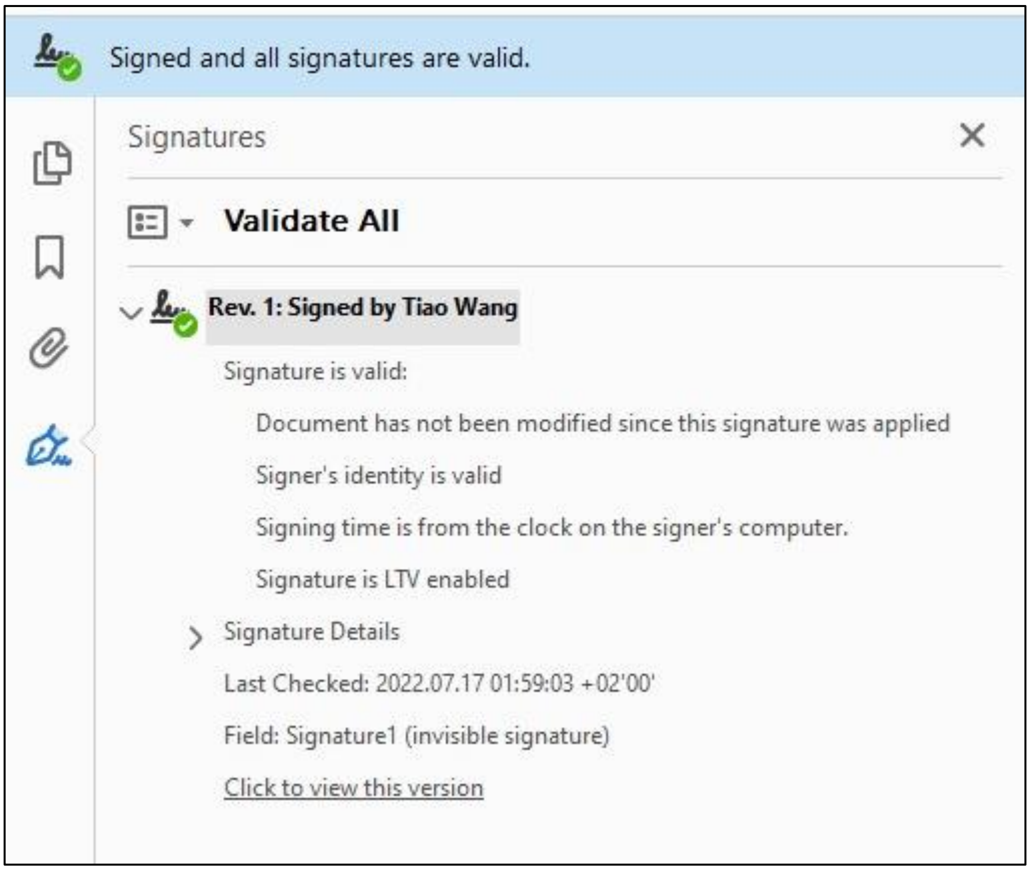
The configured signature is added to the document by the method "addSignature". The next part is to create and prepare the signature generator. The first step is to load the keystore and extract the certificates and the private key. Therefore, the Java class "java.security.KeyStore" is imported to provide keystore support. Next, the keystore source file is converted into an input stream using the Java class "java.io.FileInputStream". For loading the keystore input stream we must use the chosen password, which needs to be converted to a character array. After the kestore is loaded, the private key and the certificate chain can be extracted by the methods "~getKey" and "~getCertificateChain". For extracting the private key, the password is used again. The certificate chain is converted to a list and used to create a new instance of the Java class "org.bouncycastle.cert.jcajce.JcaCertStore". The certificate we will use is the first item from the certificate chain.

The next thing to do is to create the signature generator by creating a new instance of the Java class "org.bouncycastle.cms.CMSSignedDataGenerator". Then, an info generator is needed. For the info generator can be created, the Java classes "org.bouncycastle.operator.jcajce.JcaContentSignerBuilder" and "org.bouncycastle.operator.jcajce.JcaDigestCalculatorProviderBuilder" need to be imported and defined. The info generator is created by importing the Java class "org.bouncycastle.cms.jcajce.JcaSignerInfoGeneratorBuilder" and added to the signature generator by the method "~addSignerInfoGenerator". The certificate store needs to be added too.

After the signature generator is created and configured, the signing process can be started. First, the destination file for the signed document needs to be defined. Then, an output stream is created for the destination file. The output stream will be used by the method "saveIncrementalForExternalSigning", which will write the signing data. The next step is to make the signing data suitable for the Cryptographic Message Syntax (CMS) Standard, which is used for digital signatures and encryption. The content is read by the method "~getContent" and converted to a byte array by importing the Java class "org.apache.commons.io.IOUtils" and using the method "~toByteArray". The Java class "org.bouncycastle.cms.CMSProcessableByteArray" is imported to provide CMS support.

The method "~generate" generates the signature, which can be retrieved by the method "~getEncoded". The last step is to use the method "~setSignature" to save the signed signature to the document.

The following figure show the resulting document with the digital signature.



Signed and all signatures are valid.

Signatures

Validate All

Rev. 1: Signed by Tiao Wang

Signature is valid:

Document has not been modified since this signature was applied

Signer's identity is valid

Signing time is from the clock on the signer's computer.

Signature is LTV enabled

Signature Details

Last Checked: 2022.07.17 01:59:03 +02'00'

Field: Signature1 (invisible signature)

Click to view this version

Figure 38: Output of "*17. Creating a Digital Signature.rex*"

4.18 Verifying a Digital Signature

This example demonstrates how to verify a digital signature in a PDF document.

```
-- change directory to program location
parse source  . . pgm
call directory filespec('L', pgm)

-- define the source file and import it
source=.bsf~new("java.io.File", "17-signed.pdf")
importdoc=BSF.loadClass("org.apache.pdfbox.Loader")
fis =.bsf~new("java.io.FileInputStream","17-signed.pdf")
doc=importdoc~loadPDF(source)

-- extract the signature and the signed content
sig=doc~getSignatureDictionaries~get(0)
sigdata=sig~getContents
signeddata=sig~getSignedContent(fis)

-- convert the signed content to the cms format
pbclass="org.bouncycastle.cms.CMSProcessableByteArray"
cmsdata=.bsf~new(pbclass,signeddata)
cms=.bsf~new("org.bouncycastle.cms.CMSSignedData", cmsdata,sigdata)

-- load the certificate
signerinfo=cms~getSignerInfos~getSigners~iterator~next
cert=cms~getCertificates~getMatches(signerInfo~getSID)~iterator~next

-- create the verifier
vbclass="org.bouncycastle.cms.jcajce.JcaSimpleSignerInfoVerifierBuilder"
vbuilder=.bsf~new(vbclass)
provider=.bsf~new("org.bouncycastle.jce.provider.BouncyCastleProvider")
vbuilder~setProvider(provider)
verifier=vbuilder~build(cert)

-- verify the signature and save the result
result=signerinfo~verify(verifier)
if result=1 then content=source~getName "has a valid signature"
else content=source~getName "has a invalid signature"

-- create a new document 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
```

```
contclass = "org.apache.pdfbox.pdmodel.PDPageContentStream"
cont=.bsf~new(contclass,doc,page)

-- define font type
fontclass = "org.apache.pdfbox.pdmodel.font.Standard14Fonts"
fname = BSF.loadClass(fontclass)~FontName~HELVETICA_BOLD
font=.bsf~new("org.apache.pdfbox.pdmodel.font.PDType1Font",fname)

-- use the content stream to insert the verify result
cont~beginText
cont~setFont(font, 22)
cont~setLeading(25f)
cont~newLineAtOffset(100, 700)
cont~showText(content)
cont~endText
cont~close

-- save and close the document file
doc~save("18-signature result.pdf")
doc~close

-- get java support
::requires "BSF.CLS"
```

Figure 39: *"18. Verifying a Digital Signature.rex"*

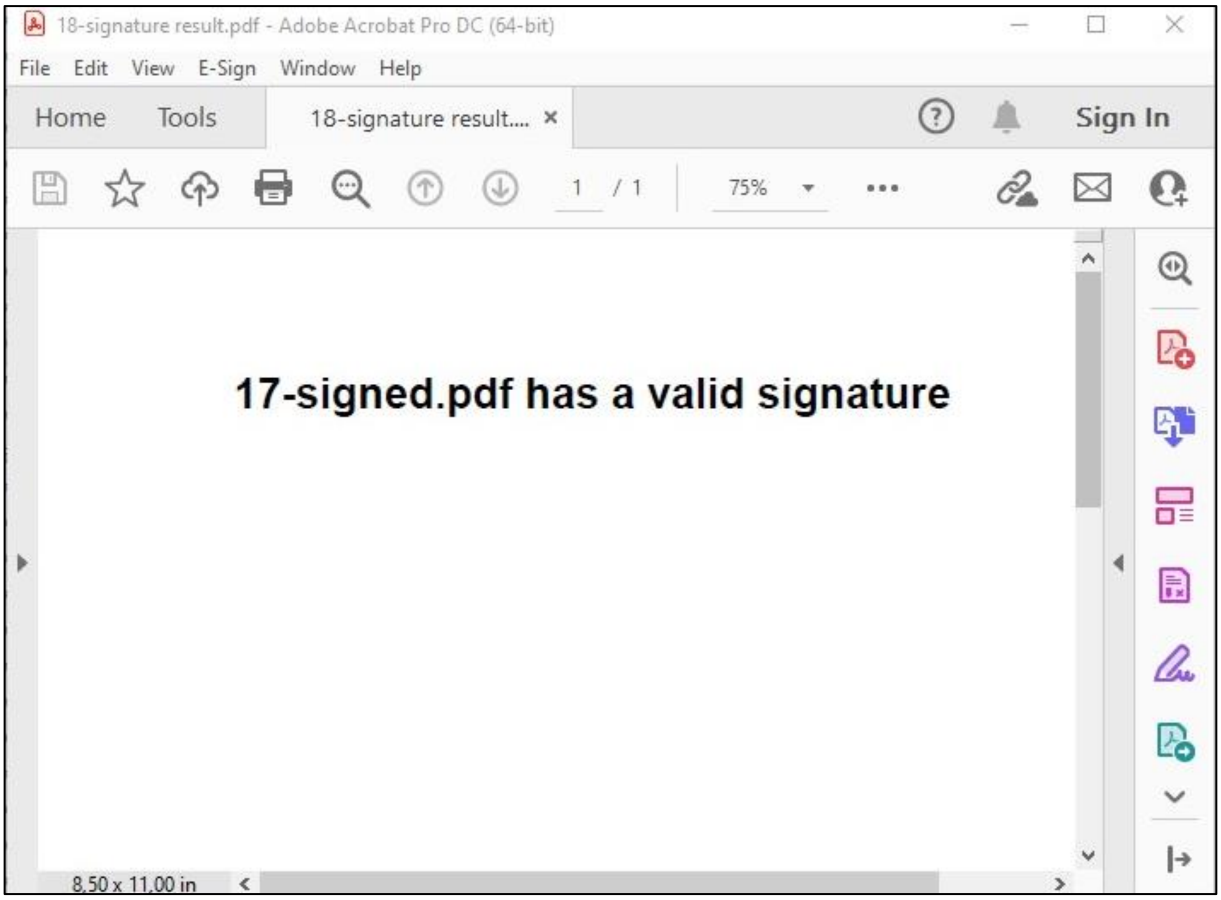
The basic idea of the verifying process is to check, whether the signature matches the signed content. So, the first step is to extract them from the signed document. After the document has been loaded, the method "~getSignatureDictionaries" is used to get all signature dictionaries. Then, the method "~get(0)" is used to get the first signature dictionary. The signature is extracted from the signature dictionary by the method "~getContents". The next step is to extract the signed content of the document. Therefore, an input stream converted from the source document file is needed. The method "~getSignedContent" extracts the signed content from the input stream.

The next step is to convert the extracted signed content to a new byte array, that is suitable for the Cryptographic Message Syntax (CMS) Standard. The Java class "org.bouncycastle.cms.CMSProcessableByteArray" is imported for this task. Next, the converted signed content and the signature are used to create a new object of the Java class "org.bouncycastle.cms.CMSSignedData", which can checked, whether the signature matches the signed content. For the verifying process a verifier needs to be created. The first step is to get the signer info by using the method "~getSignerInfos" and certificate that matches the signer info. Next, a verifier builder is created by the

Java class "org.bouncycastle.cms.jcajce.JcaSimpleSignerInfoVerifierBuilder". Then, the Java class "org.bouncycastle.jce.provider.BouncyCastleProvider" is set as the security provider by using the method "~setProvider". The method "~build" uses the retrieved certificate and creates the verifier, which is used to check the signature. If the value 1 is returned, the signature is valid. Otherwise the signature doesn't match the signed content.

The last thing to do is to create a new document and insert the verifying result. After that, the document can be saved and closed.

The following figure shows the resulting document.


18-signature result.pdf - Adobe Acrobat Pro DC (64-bit)
File Edit View E-Sign Window Help
Home Tools 18-signature result....
Sign In
1 / 1 75%
8,50 x 11,00 in

17-signed.pdf has a valid signature

Figure 40: Output of "*18. Verifying a Digital Signature.rex*"

5 Conclusio

In conclusion, this thesis has explored the use of Apache PDFBox in combination with ooRexx to create and manipulate PDF documents. Through the 18 Nutshell examples provided, we have demonstrated the extensive capabilities of this platform, highlighting the power and flexibility of PDFBox for a range of practical applications.

The use of BSF4ooRexx as a bridge between Java and ooRexx has provided a valuable toolkit for working with Java Libraries such as Apache PDFbox, which has allowed developers and users to create custom PDF manipulation solutions tailored to their specific needs. The platform has opened new possibilities for PDF document creation and manipulation, increasing efficiency and productivity.

Through the Nutshell examples provided, we have demonstrated the ability of PDFBox to create and manipulate PDF documents for a range of practical applications, including creating PDF documents, generating contents, and extracting information from PDF files. The examples have highlighted the ease of use and flexibility of PDFBox, making it an attractive platform for developers and users alike.

Looking towards future research, there is potential to explore the use of more complex programs in conjunction with PDFBox, Java, and ooRexx to handle even more complex document structures. Additionally, deeper exploration of the PDFBox library could help identify more advanced features and capabilities, providing even greater opportunities for developers and users.

Overall, this thesis has shown the potential of PDFBox and ooRexx for creating and manipulating PDF documents. The Nutshell examples provided in this thesis serve as a foundation for further research and development in this area, highlighting the importance of exploring the potential of PDFBox to advance the capabilities of PDF document manipulation. The combination of PDFBox and ooRexx represents a valuable resource for anyone working with PDF documents, and the potential for further development in this area is significant.

6 References

[1] D. Johnson, “PDF: The document format for everything.” https://www.pdfa.org/pdf-the-document-format-for-everything-2/

[2] “What is a PDF? Portable Document Format | Adobe Acrobat.” https://www.adobe.com/acrobat/about-adobe-pdf.html

[3] “Choosing a security method for PDFs.” https://helpx.adobe.com/acrobat/using/choosing-security-method-pdfs.html

[4] J. Warnock, “The Camelot Project.” https://www.pdfa.org/norm-refs/warnock_camelot.pdf

[5] “Document Management — Portable Document Format — Part 1: PDF 1.7,” 2008. https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf

[6] Wikipedia contributors, “History of PDF,” *Wikipedia*, Jan. 29, 2023. https://en.wikipedia.org/wiki/History_of_PDF#:~:text=The%20Portable%20Document%20Format%20(PDF,an%20open%20standard%20in%202008.

[7] Kodhodbanaan, “Benefits of Using PDF Files In Your Office - Tishare,” *newzworldmagazine.com*, Jan. 24, 2023. https://worldtimemagazine.com/benefits-of-using-pdf-files-in-your-office-tishare/

[8] “Overview of security in Acrobat and PDFs.” https://helpx.adobe.com/acrobat/using/overview-security-acrobat-pdfs.html

[9] “How to compress a PDF file.” https://helpx.adobe.com/acrobat/how-to/compress-pdf.html#:~:text=To%20reduce%20the%20size%20of,from%20the%20drop%2Ddown%20menu.

[10] “How to make PDF searchable: Make PDF text searchable | Adobe Acrobat.” https://www.adobe.com/acrobat/hub/how-to/make-a-pdf-searchable

[11] “Apache License, Version 2.0.” https://www.apache.org/licenses/LICENSE-2.0

[12] Wikipedia contributors, “Apache PDFBox,” *Wikipedia*, Oct. 02, 2022. https://en.wikipedia.org/wiki/Apache_PDFBox

[13] “Index of /dist/pdfbox/pdfbox/1.0.0.” https://archive.apache.org/dist/pdfbox/pdfbox/1.0.0/

[14] “Apache PDFBox | PDFBox 2.0.0 Migration Guide.” https://pdfbox.apache.org/2.0/migration.html

[15] “Apache PDFBox | PDFBox 3.0 Migration Guide.” https://pdfbox.apache.org/3.0/migration.html

[16] Wikipedia contributors, “Java (programming language),” *Wikipedia*, Mar. 09, 2023. https://en.wikipedia.org/wiki/Java_(programming_language)

[17] “Features of Java - Javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/features-of-java

[18] “Java OOP (Object-Oriented Programming).” https://www.w3schools.com/java/java_oop.asp#:~:text=Java%20%2D%20What%20is%20OOP%3F,contain%20both%20data%20and%20methods.

[19] “Java JRE | Java Run-time Environment - Javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/java-jre

[20] “Java Bytecode - Javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/java-bytecode

[21] W. I. a J. V. M.-D. F. Techopedia, “Java Virtual Machine (JVM),” *Techopedia.com*, May 01, 2013. https://www.techopedia.com/definition/3376/java-virtual-machine-jvm

[22] Wikipedia contributors, “Java Class Library,” *Wikipedia*, Jan. 13, 2023. https://en.wikipedia.org/wiki/Java_Class_Library

[23] “Difference between JDK, JRE and JVM - javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/difference-between-jdk-jre-and-jvm

[24] Wikipedia contributors, “Object REXX,” *Wikipedia*, Dec. 30, 2022. https://en.wikipedia.org/wiki/Object_REXX

[25] Wikipedia contributors, “Rexx,” *Wikipedia*, Feb. 08, 2023. https://en.wikipedia.org/wiki/Rexx

[26] “Charter of the Open Object Rexx Project.” https://www.oorexx.org/charter.html

[27] “About Open Object Rexx.” https://www.oorexx.org/about.html

[28] Wikipedia contributors, “Bean Scripting Framework,” Wikipedia, Aug. 21, 2020. https://en.wikipedia.org/wiki/Bean_Scripting_Framework

[29] R. Flatscher, “Business Programming 1.” https://wi.wu-wien.ac.at/rgf/wu/lehre/autowin/material/foils/

[30] R. Flatscher, “Business Programming 2.” http://wi.wu-wien.ac.at/rgf/wu/lehre/autojava/material/foils/

Appendix

Prerequisites to execute the nutshell examples:

Software:

- Azul Zulu JDK -> https://www.azul.com/downloads/?package=jdk#zulu
- ooRexx https://sourceforge.net/projects/oorexx/
- BSF4ooRexx https://sourceforge.net/projects/bsf4oorexx/
- Apache PDFBox https://www.apache.org/dyn/closer.lua/pdfbox/3.0.0-alpha3/pdfbox-app-3.0.0-alpha3.jar
- xmpbox https://www.apache.org/dyn/closer.lua/pdfbox/3.0.0-alpha3/xmpbox-3.0.0-alpha3.jar
- preflight https://www.apache.org/dyn/closer.lua/pdfbox/3.0.0-alpha3/preflight-3.0.0-alpha3.jar

Installation Guide:

1. Download the latest versions of the required software.
2. Install Azul Zulu JDK.
3. Install ooRexx.
4. Install BSF4ooRexx.
5. Copy the downloaded .jar files into the lib folder of the Bsf4ooRexx850 directory.

How to use:

1. Verify that the resource folder is present.
2. Execute the Nutshell examples in the correct order.
3. Check the results.