Apache POI (OOXML4):

Create and Process Microsoft Office Files, Cookbook with Nutshell Examples

Fabian Fuchs

11705016

Vienna University of Business and Economics
Business Information Systems Seminar (4152)
Summer Semester 2021

Declaration of Authorship

I assure:

to have individually written, to not have used any other sources or tools than referenced and to not have used any other unauthorized tools for the writing of this report.

to never have submitted this report topic to an advisor neither in this, nor in any foreign country.

that this report matches the report reviewed by the advisor.

Date: June 1st, 2021,

X

Fabian Fuchs

Table of Contents

List of Figures 4
1. Introduction 5
1.1 Apache POI 5
1.2 ooRexx and BSF4ooRexx 6
2. Methodology 6
3. Installation 7
3.1 Java 7
3.2 BSF4ooRexx 7
3.3 Apache POI 8
4. Microsoft Excel 10
4.1 Example 1 – Date.xlsx 10
4.1.1 Create Sheet and Cell 11
4.1.2 Edit Cell and Font 11
4.1.3 Change Cell Type and Size 12
4.1.3 Colour and Cell Border 13
4.1.4 Change Specific Cells and Save the File 13
4.2 Example 2 – Graph.xlsx 16
4.2.1 Position the Graph 16
4.2.2 Create Axes 17
4.2.3 Add Data to the Graph 17
4.2.4 Combine the Parts 18
5. Microsoft Word 19
5.1 Example 1 – Table.docx 19
Create Document and Text Run 19
5.1.1 Edit the Text Run 20
5.1.2 Create Table and Fill Cells 20
5.1.3 Edit Cell Text and Save the File 21
5.2 Example 2 – ImageAndReplace.docx 23
5.2.1 Add an Image 23
5.2.2 Replace a Word 24
6. Microsoft PowerPoint 26
6.1 Example 1 – Basics.pptx 26
6.1.1 Create PowerPoint and Slides 27
6.1.2 Create a Text Box and Edit the Text 27

6.1.3 Create Bullet Points 28
6.1.4 Add a Layout and Save the File 29
6.2 Example 2 – Read.pptx 31
6.2.1 Open the PowerPoint Presentation 31
6.2.2 Change Slide Order and Remove Slides 32
6.2.3 Add a Hyperlink 32
6.2.4 Add an Image 32
7. Conclusion 34
References 35
Appendix 36
ExcelExample1.rexx 36
ExcelExample2.rexx 38
WordExample1.rexx 40
WordExample2.rexx 40
PowerPointExample1.rexx 43
PowerPointExample2.rexx 45

List of Figures

Figure 1: Environment Variable 8
Figure 2: CLASSPATH 9
Figure 3: Directory and Name of the .jar files 9
Figure 4: Output of ExcelExample1.rexx 10
Figure 5: Output of ExcelExample2.rexx 16
Figure 6: Output of WordExample1.rexx 19
Figure 7: Output of Image+Replace.rexx 23
Figure 8: Output of PowerPointExample1.rexx (Slide 1 -3) 27
Figure 9: Output of PowerPointExample2.rexx (Slide 1 - 3) 31

1. Introduction

This paper focuses on providing an overview over the functionality of Apache Poi, with the help of nutshell examples. Even though Apache POI is a Java API for Microsoft Office files, the programming language of choice is not Java, but ooRexx with the help of the BSF4ooRexx function package. The reason being that there are already many Java nutshell examples available for Apache POI and that the syntax of ooRexx is easy to learn for programming beginners (Flatscher, 2012).

1.1 Apache POI

Apache POI provides Java libraries that enable the manipulation of Office Open XML files (OOXML) and Object Linking & Embedding files (OLE2). This means that files of the main Microsoft Office applications, Word, PowerPoint, and Excel can be read and written in Java (Wikipedia.org, 2021). The use of Java allows applications to run independent on Linux, Windows and macOS (Wikipedia.org, 2021).

The project was first released in 2006. In the beginning of the project POI used to be an acronym for "Poor Obfuscation Implementation" which intended to mock Microsoft for obfuscating their file format but not good enough to be reverse engineered. However, the name has been removed from the official project website because of concerns that the project will be seen as untrustworthy by potential business clients (Wikipedia.org, 2021). The newest stable version namely 5.0.0[1] was released in January 2021 (Apache Software Foundation, 2021) .

[1] This particular Apache POI version leads to a bug in the second Excel example *"4.2 Example 2 – Graph.xlsx"*. It is strongly suggested to use a newer version where the bug is already resolved. However, it is also possible to use an older version like Apache POI 4.1.2. More information on the bug can be found in the official bug database entry here: https://bz.apache.org/bugzilla/show_bug.cgi?id=65016

1.2 ooRexx and BSF4ooRexx

The history of Open Object Rexx (ooRexx) starts in IBM, as the proprietary object-oriented successor of the programming language Rexx. After failed tries by IBM to sell it, the source code became public through the Rexx Language Association. Since then, the language has been continuously developed further. Overall, it is very a human centric programming language and thus easier to learn for beginners than Java.

BSF4ooRexx (Bean Scripting Framework for ooRexx) is a function package for ooRexx. It enables the use of java classes in ooRexx as if they were part of it and vice versa. The first successful implementation was achieved in the year 2000 (Flatscher, 2012).

2. Methodology

For this paper, I partly translated Java applications into ooRexx and partly created own applications with the help of the available Javadoc's for Apache POI and its different modules. Most of the pre-existing Java applications I used can be found on the following websites:

- https://poi.apache.org/index.html
- https://www.tutorialspoint.com/index.htm
- https://www.javatpoint.com/

Moreover, I used https://stackoverflow.com/ whenever something was unclear, in the Javadoc's or in the give Java applications. The full code for every application explained in the paper is available in the *Appendix*.

To ensure that all applications are easily applicable for readers I divided them into building blocks that are not completely independent. However, with the help of the code lines at the beginning of every application and the ones at the end most of the building blocks can be used independently.

3. Installation

This chapter is quick guide on which programs were used for this cookbook, where to find them and how to install them.

3.1 Java

First and foremost, Java has to be installed on the computer. To check if it is already installed it is possible to enter “java -version” in the cmd-box which yields the respective version. In the case of this paper “1.8.0_271-b09” 32bit. The download file can be found here: https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html. It is called JDK 8u271 for Windows.

3.2 BSF4ooRexx

Before we can use BSF4ooRexx the underlying programming language ooRexx has to be installed. The file can be found online under: https://sourceforge.net/projects/oorexx/files/oorexx/. For this project “ooRexx 5.0.0 r12142” was used. It is important to note that the java version and the ooRexx version should have the same bit rate.

In order to make use of the java classes BSF4ooRexx has to be installed. The respective file, in this work “BSF 641.20201217” can be found online under: https://sourceforge.net/projects/bsf4oorexx/files/. The installation is done in three steps after downloading the latest file:

1. Unzip the downloaded archive
2. Go to the subdirectory “bsf4oorexx/install/windows” if Windows is the operating system
3. Execute install.cmd

In my case the following error occurred: Executing install.cmd did not start the installation process but rather opened a new window which proposed several options on how to open the file. Subsequently BSF4ooRexx was not installed properly no matter which option was chosen for opening the file.

The solution for this problem was to either change kickoff.rex or elevate.rex to open with Open Object Rexx executable. This can be done as follows:

1. Right click on kickoff.rex or elevate.rex
2. Open properties

3. Select “open with” and change it to Open Object Rexx executable
4. Execute install.cmd again to properly install BSF4ooRexx

3.3 Apache POI

The last component that is needed for the nutshell examples is Apache POI, in order to use the Java classes necessary to manipulate MS office files. The latest version can be found on the official Apache POI website: https://poi.apache.org/download.html. For this paper “poi-bin-5.0.0-20210120.zip” was used. Since Apache POI will simply extend the Java classes available there is no installation necessary. Nevertheless, it is required to set the classpath of the required .jar files so that the Java Runtime Environment can access them. (Wikipedia.org, 2021).

The classpath can be set as follows:

1. Unpack the zip archive to access the necessary .jar files. A list of which .jar files are necessary for the respective MS office application can be found under: https://poi.apache.org/components/ All .jar files used in this paper can be found in the *Appendix.*
2. Type “env” into the search field.
3. Open “Change Environment Variable”
4. Click on Environment Variable (Figure 1)

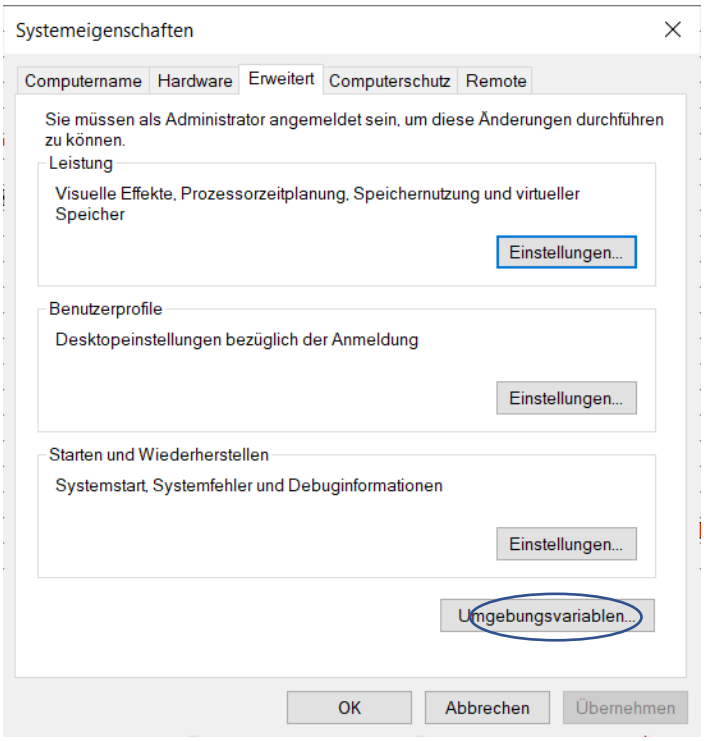

Figure 1: Environment Variable

5. Double Click on CLASSPATH (Figure 2)

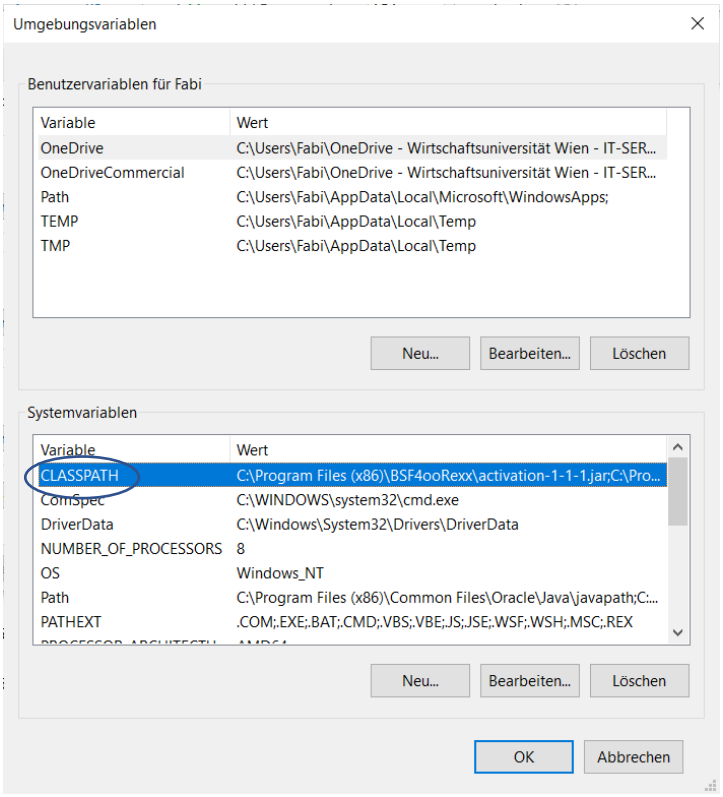

Figure 2: CLASSPATH

6. Click "New" and enter the directory and the name of the .jar file (Figure 3)

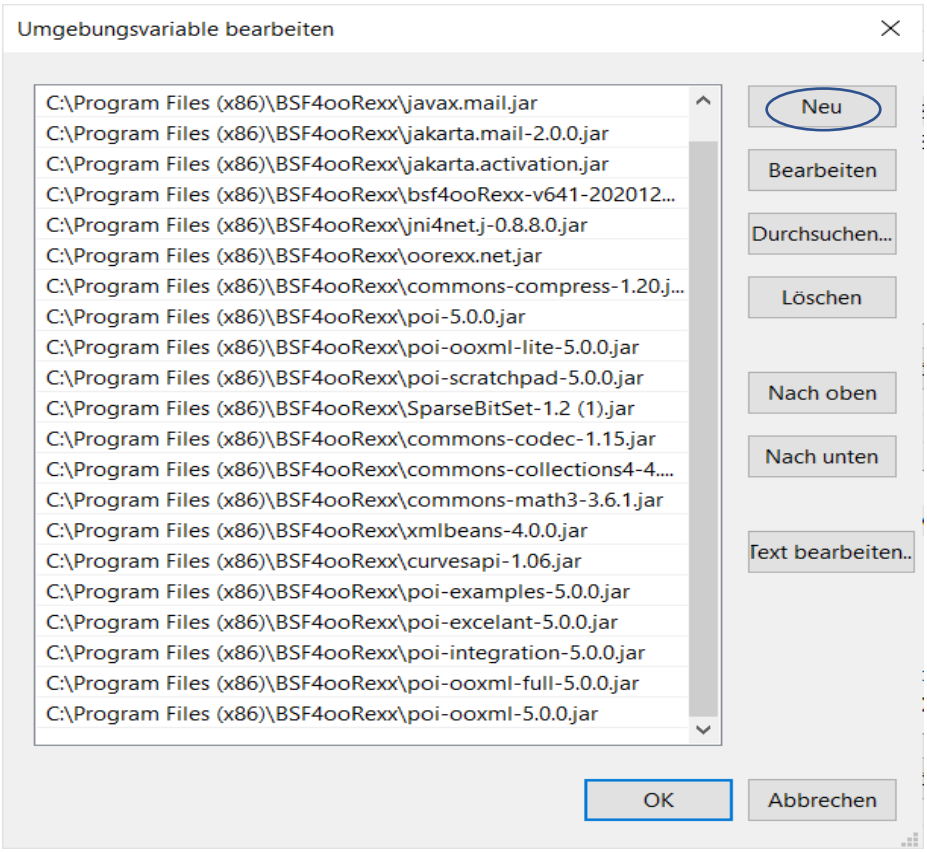

Figure 3: Directory and Name of the .jar files

4. Microsoft Excel

Microsoft Excel is the most common spreadsheet application on the market. As the other two applications featured in this paper it is part of the Microsoft Office family. The use cases for Excel are manifold. Amongst other things one can use calculation tools, visualize data and use it as a base to create extensive pivot tables (Wikipedia.org, 2021). Apache POI provides two different Java implementations for Excel, one for the current file format .xlsx, called POI-XSSF/SXSSF, and one for the prior file format .xls namely POI-HSSF (The Apache Software Foundation, 2021). Overall, HSSF and XSSF are the most advanced modules of Apache POI (The Apache Software Foundation, 2021).

This chapter will focus on providing example code for creating and editing Excel files as well as explaining the relevant steps necessary. It should provide the reader with sufficient knowledge to create basic applications by themselves. To achieve this the first part consists out of an application that shows all the basics necessary to create and edit a spreadsheet. The second part will feature a more advanced application for creating a graph. In order to keep it simple booth applications will not includ difficult programming concepts but rather focus on the most important features of the Apache POI class libraries.

4.1 Example 1 – Date.xlsx

The goal of the first example is to create an Excel sheet that shows us the date. This requires filling a cell and switching the standard cell type to date. Since these examples should show the different possibilities of Apache POI, the cells will additionally be edited. The output of the application can be seen in Figure 4.

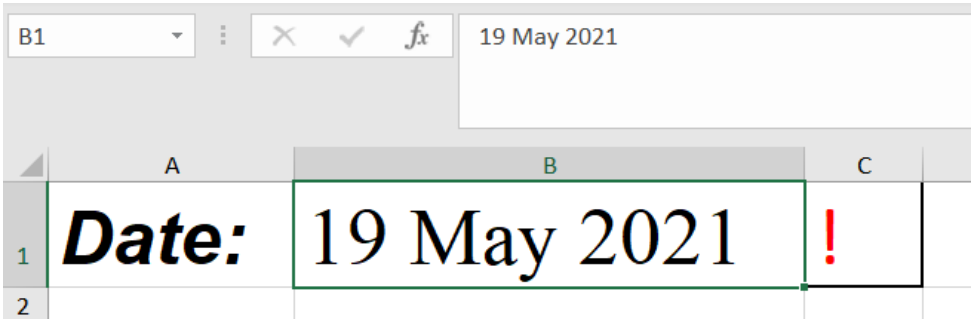

Figure 4: Output of ExcelExample1.rexx

4.1.1 Create Sheet and Cell

To get started we have to create a excel sheet, add a cell and subsequently add data to the cell. The first thing we have to do is creating an instance of the workbook class. This allows us to create and name our first sheet with the “createSheet” method. Now that we have a sheet, we can create our first cell. However, a cell cannot be created directly without first creating an associated row. It is important to mention that the parameter “rownum” of the “createRow” method as well as the parameter “column” of the “createCell” method start with zero. In this case we have successfully created the cell A1. The last step is to add the cell value which is done in line 8 below.

```
1 workbook=.bsf~new("org.apache.poi.xssf.usermodel.XSSFWorkbook")
2 sheet=workbook~createSheet("Date")
3
4
5 -- Create Cells and add Content
6 row=sheet~createRow(0)
7 cell=row~createCell(0)
8 cell~setCellValue("Date:")
```

4.1.2 Edit Cell and Font

Now that we have created our cell, we can edit it. There are two classes that provide the methods for editing a cell. The first one “XSSFFont” allows us to alter the content of the cell. The second one “XSSFCellStyle” is necessary to edit the cell itself and to return the stored references to the cell object. In our case we are only altering the font. This is done by creating an instance of the font class and assign the desired format via the given methods. It is noteworthy, that the Boolean parameter for “setBold” and “setItalic” are not “true” or “false” as in java but rather 0(false) or 1(true) for ooRexx. Afterwards an instance of the “XSSFCellStyle” class has to be created so we can store the format. This is necessary because the method “setCellStyle” requires a CellStyle object as parameter.

```
 1 --Edit Cell Style
 2 font=workbook~createFont()
 3 font~setFontHeightInPoints(30)
 4 font~setFontName("Arial")
 5 font~setBold(1)
 6 font~setItalic(1)
 7
 8 --Set Cell Style to Specific Cells
 9 style=workbook~createCellStyle()
10 style~setFont(font)
11 cell~setCellStyle(style)
```

4.1.3 Change Cell Type and Size

For the second cell we simply repeat all the prior steps for creating and editing it. The only difference is that we have to assign the correct data format to the cell. Line 8 and 9 in the picture below show us how this can be done. It is necessary to create a helper Object with the "getCreationHelper" method, which allows to instantiate the various instances of concrete classes for XSSF. With the helper object we can use the "createDataFormat" method, so we are able to get an instance of "XSSFDataFormat" and the associated methods. Then one can us the "getFormat" method, to choose the format with a text string. A list of all available formats and the respective strings is available in the Javadocs of the class "BuiltInFormats". This class also includes the method "getBuiltInFormat" with a string parameter, which returns the correct format. For example, "getBuiltInFormat("text") would yield the text format. The cell format is, as all other changes, passed to the "CellStyle" object. In our case we did not set the date manually but as one can see in line 2, we set the value of the cell with the built-in ooRexx function "Date", that always returns the correct date.

```
 1 cellx=row~createCell(1)
 2 cellx~setCellValue(Date())
 1 font2=workbook~createFont()
 2 font2~setFontHeightInPoints(33)
 3 font2~setFontName("Times New Roman")
 4
 5
 6 style2=workbook~createCellStyle
 7 style2~setFont(font2)
 8 helper=workbook~getCreationHelper()
 9 style2~setDataFormat(helper~createDataFormat()~getFormat("m/d/yy")) --
Set Cell Format
10 cellx~setCellStyle(style2)
```

The next picture shows us how we can automatically adapt the cells to fit our high font size. We only need the method "autoSizeColumn" and add the columns we would like to edit. If one prefers to edit the size of the cells manually the "XSSFSheet" class offers a method for setting the width, and the "XSSFRow" offer two methods for setting the height.

```
1 a=cell~getColumnIndex()
2 b=cellx~getColumnIndex()
3
4 do until a > b
5     sheet~autoSizeColumn(a)
```

```
6       a=a+1
7   end
```

4.1.3 Colour and Cell Border

To complete our worksheet, we add the last cell and format it accordingly. As one can see in the picture the method used for recolouring is “setColor”. There are two different set colour Methods. One uses the Enum constants of the class “IndexedColors as we see in line 8 below. The second possibility is to use an instance of the class “XSSFColour” and simply assign the RGB value to the constructor, which can be seen in line 5 of the second code snippet.

```
1  cell=row~createCell(2)
2  cell~setCellValue("!")
3
4  .bsf~bsf.importClass("org.apache.poi.ss.usermodel.IndexedColors",
"IndexedColors")
5
6  style3=workbook~createCellStyle
7  font3=workbook~createFont()
8  font3~setColor(.IndexedColors~Red~getIndex())
9  font3~setFontHeightInPoints(33)
```

```
1  .bsf~sf.importClass("org.apache.poi.xssf.usermodel.XSSFColor", "Color")
2  style3=workbook~createCellStyle
3  font3=workbook~createFont()
4  font3~setColor(.Color~new("#ff90d9"))
```

Booth methods have in common that they use a new Java class in their parameters. If one wants to translate this into ooRexx one has to use the BSF4ooRexx method “bsf.importClass”. This method makes it possible to use a Java class as if it were a ooRexx class. As one can see the same process is implemented for the method “setBorder…” were we need to determine the thickness of the border with the help of the “BorderStyle” class and its given Enum constants.

```
1 .bsf~bsf.importClass("org.apache.poi.ss.usermodel.BorderStyle", "BoStyle")
2
3  style3~setFont(font3)
4  style3~setBorderBottom(.BoStyle~Medium)
5  style3~setBorderTop(.BoStyle~Medium)
6  style3~setBorderRight(.BoStyle~Medium)
7  style3~setBorderLeft(.BoStyle~Medium)
```

4.1.4 Change Specific Cells and Save the File

The thoughtful reader might have already noticed that the third cell is called “cell” just like our first cell. Because the name is the same, one has to resort to two simple methods before we can set the style of the cell, that we just created, via the “setCellStyle” method. These methods are “getRow” and “getCell” in this order and can be found in line 1.

```
1 --Save the File
2 parse source . . a
3 call directory filespec('L', a)
4
5 filename = directory()"/Date.xlsx"
6 filename = qualify(filename)
7
8 output=.bsf~new("java.io.FileOutputStream", filename)
9 doc~write(output)
```

To finalize the first application, one needs the “fileOutputStream” class. The constructor allows us to the specify the path and the filename of our excel sheet. To make sure that our application can run on every operation system we need to add lines 2-6. This is due to the difference in how file paths are specified. For that reason, we need to automatically retrieve the file path and cannot manually add it to the constructor of “fileOutputstream” because a Unix System will not be able to save the file if a Windows directory is given. As one can see in line 2 and 3 the path of the file is manipulated. It is important to mention that this is only necessary if one wants to run the application in and integrated development environment, in this case IntelliJ. If the application is executed directly there is no need for manipulating the path. The reason being that the directory where the application runs is different from where it is saved. The parse source command allows us to find the directory where the program is saved. The dots are placeholder if one wants to get additional information which is not necessary, we only need the path which we can find in third place. Subsequently the next line calls the directory command which allows us to switch to the prior identified directory with the help of the “filespec” method, which returns the selected directory. This method has four different options for the first parameter, namely Location(L), Path(P), Drive(D), Name(N) and Extension(E). We add L for location and the correct directory with the variable “a”. Line 5 adds the directory, we just switched to, with the

help of the directory() method and the name of the file manually, to a variable. Now that we have the path and the name of the file stored inside a variable, we can use the “qualify” method to create a fully qualified path. It is necessary to store the new string in a variable otherwise the fully qualified name is passed as a command to the operating system and the file will be opened if it already exists.
At last, we use the method “write” to add the workbook to the instance of “FileOutputStream” we just created. To get access to all Java classes we have to use ::requires BSF.CLS, which is standard for BSF4ooRexx and will thus not be mentioned anymore.

To sum up the process, we have switched to the directory where the application is saved, added the file path of this directory and the name of our file to one variable. Subsequently we added this variable to the constructor of “FileOutputStream” and thus saved the Excel file in the same directory where the application is stored. This process will from now on be used for every instance of the “FileOutpuStream” or “FileInputStream” class in this paper.

4.2 Example 2 – Graph.xlsx

The second example shows how we can create a line graph, that shows the development of the Austrian population over the last 30 years. This example focuses solely on the creation of a graph, which means there will be no additional editing of certain cells. The output of the second application can be found below in Figure 5. As already mentioned in the first footnote Apache POI 5.0.0 can lead to a bug in this example. More precisely a java.lang.IndexOutOfBoundsException error can occur when creating the chart via the "plot" method Thus, it is strongly suggested to use a newer version where the bug is already resolved (5.0.1). However, it is also possible to use an older version like Apache POI 4.1.2. More information on the bug can be found in the official bug database entry here: https://bz.apache.org/bugzilla/show_bug.cgi?id=65016

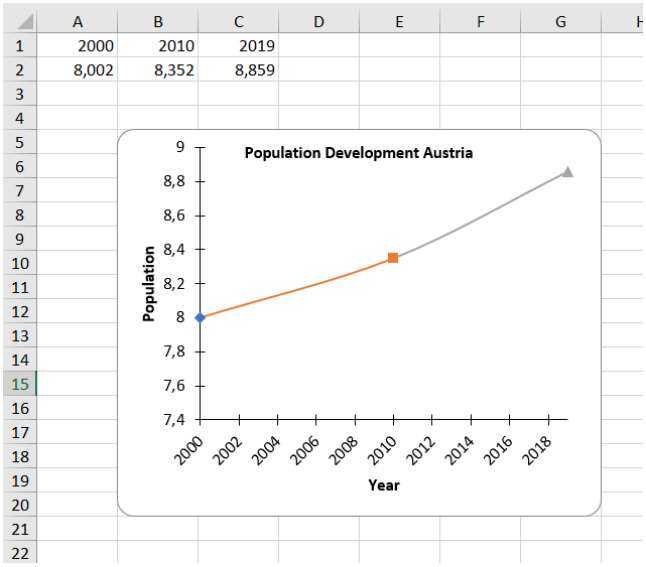

Figure 5: Output of ExcelExample2.rexx

4.2.1 Position the Graph

Before the graph can be created, we have to create the document, a sheet and fill in the data into the cells. This has been done the same way as in the first example and is this not featured here. As already mentioned, the full code is available in the *Appendix*. The second step is to create the graph and position it onto the sheet. For this we have to create a drawing patriarch. Then we position the empty drawing patriarch via the "createAnchor" method. Booth the size and the position are determined through the last four parameters of this method, which set the distance, in cells, from the edge of the excel sheet. Comparing Figure 5 with the values of the parameter helps to better understand the last four values of

this method. The last thing that has to be done is adding the graph with “createChart” and the prior set anchor as parameter and store it in a variable.

```
chart=.bsf~new("org.apache.poi.xssf.usermodel.XSSFChart")

drawing=sheet~createDrawingPatriarch()
anchor=drawing~createAnchor(0, 0, 0, 0, 1, 4, 7, 20)
chart=drawing~createChart(anchor)
```

4.2.2 Create Axes

Every graph has two axes. The next code sequences show how they are created with Apache POI. The “AxisPosition” class allows us to set the position of the axis, it is thus necessary to import it. Then we have three different choices of axes we can create, namely value, date, and category. In our case we need the date and the value axis and are, thus using the respective methods to add them to our chart variable, that stores the anchored empty graph. Lastly, we add the respective titles of the axes.

```
-- Axes
.bsf~bsf.importClass("org.apache.poi.xddf.usermodel.chart.AxisPosition","Position”)

bottomAxis=chart~createDateAxis(.Position~BOTTOM)
bottomAxis~setTitle("Year")
leftAxis = chart~createValueAxis(.Position~LEFT)
leftAxis~setTitle("Population")
```

4.2.3 Add Data to the Graph

Next, we have to add the data from the cells to the graph. For this we use the constructor of the “CellRangeAdresss” class, which allows us to enter the first row, last row, first column and last column in this order. In this case I used the “bsf.importClass” method but is also possible to simply to use “.bsf~new” as seen in the second image. The next step is to use the bsf.loadclass method in order to load the abstract class “XDDFDataSourcesFactory”. This method is necessary whenever an abstract class needs to be imported. However, if one tries the “bsf.importClass” method on a wrong Java class the occurring error message will explain the problem and suggest using “bsf.loadClass”. The factory supplies us with an instance of “ XDDFDataSource” which allows us to store our selected cells in the correct way for later adding it to our graph. This process is done with the help of the method “fromNumericCellRange” which allows us to specify the sheet and lets us add the cells we have just specified with the constructor of the “CellRangeAddress” class.

```
.bsf~bsf.importClass("org.apache.poi.ss.util.CellRangeAddress", "CellRangeAddress")
address=.CellRangeAddress~new(0, 0, 0, 2)
dataSource=bsf.loadClass("org.apache.poi.xddf.usermodel.chart.XDDFDataSourcesFactory")
```

```
address1=.bsf~new("org.apache.poi.ss.util.CellRangeAddress", 1, 1, 0, 2)
```

4.2.4 Combine the Parts

To finish the graph, we have to bring all the objects we have already created together. This means we have to first add the axes and the type of graph we want to the chart object. We are going to store all our prior created objects in the newly created data object. This is done with the help of the "createData" and the "addSeries" method. The first method is responsible for setting the kind of graph that is added to the empty chart. In addition, its stores the axes that were created earlier. The second method allows us to store the value of our cells. The last step we have to do now is to return the data object with all the information about the graph to the chart object with the help of the "plot" method.

```
--Add data to the graph
data=.bsf~new("org.apache.poi.xddf.usermodel.chart.XDDFLineChartData")
.bsf~bsf.importClass("org.apache.poi.xddf.usermodel.chart.ChartTypes", "ChartTypes")

data=chart~createData(.ChartTypes~LINE, bottomAxis, leftAxis)
data~addSeries(years, population)
chart~plot(data)
```

To save and name the file the same procedure as in *4.1 Example 1* – Date.xlsx is used.

5. Microsoft Word

Microsoft Word is the word processor of the Microsoft office family. It was first published in 1983. There are countless of features available for Word, which include adding picture and tables, spell-checking, and countless possibilities for editing text (Wikipedia.org, 2021). Apache POI offer support for Microsoft Word 97 and onwards. All versions of word until 2007 are using the .doc format and thus the HWPF port, while newer versions with the .docx format rely on the XWPF implementation (The Apache Software Foundation, 2021).

This Chapter will provide two examples that allow the reader to get an overview over the functionality of the Apache POI API in combination with bsf4ooRexx. As in the last chapter the focus will be to show as much of the possibilities as possible without adding difficult concepts. The goal is that the shown classes and methods can be readily used for one's own applications.

5.1 Example 1 – Table.docx

This example shows how a word document can be created and filled with text. The created text will subsequently be edited. In addition, as the name suggests, a table will be created and filled with text. The output of the first example can be found in Figure 6

Figure 6: Output of WordExample1.rexx

Create Document and Text Run

The first lines of code show the basics for creating a word document. At first, we create an object of the "XWPFDocument" class which is our high-level class for working with .docx files. In order to start adding text to our document we have to create a paragraph and a run. This can be done via the methods found in line four and eight. In addition, one can position the paragraph via the "setAlignement" method.

This will require the use of the Enum constants found in the class “ParagraphAlignement” as a parameter.

```
1 doc=.bsf~new("org.apache.poi.xwpf.usermodel.XWPFDocument")
2
3 .bsf~bsf.importClass("org.apache.poi.xwpf.usermodel.ParagraphAlignment", "ParagraphAlignment")
4 p1=doc~createParagraph()
5 p1~setAlignment(.ParagraphAlignment~CENTER)
6
7 --Create simple Text and edit it
8 t1=p1~createRun()
```

5.1.1 Edit the Text Run

The next lines of code are very straightforward. As already explained in *4.1 Example 1* – Date.xlsx*t* the Boolean values for “setBold” and “setItalic” are 0 and 1. Interestingly there are major differences between the methods of the classes “XWPFRun”, which is used for this example, and “XSSFFont”, which provides methods for editing text in Excel. The first difference being that the “setColor” method of “XWPFRun” cain directly use a hex value without the need to instantiate an extra class (line 8). Moreover, the method used to determine the font is called “setFontFamily” instead of “setFontName” for Excel.

```
1 t1~setText("This is the first run")
2 t1~addBreak()
3 t1~setText("We can edit each run as we like!")
4 t1~setFontSize(19)
5 t1~setBold(1)
6 t1~setItalic(1)
7 t1~setColor("0320fc")
8 t1~setFontFamily("Times New Roman")
```

5.1.2 Create Table and Fill Cells

Moving on the goal is to create a table. This can be done via one of the “createTable” methods of the “XWPFDocument” class we created right in the beginning. The two methods have different parameters. The first one does not require any input for the parameter and only creates one row with one cell, while the second method already creates rows and columns based on the given values off the parameter. We have used the second “createTable” method to create our first two rows and two columns. Subsequently we added text to each cell by getting the row and the specific cell we want to edit as seen in line 5-6 and 9-10. There is one main difference between the “createTable” method and the getter methods. The

“createTable” method starts counting with one in contrast to the getter methods which start with zero as common for java. It should also be kept in mind that every cell that has been created with “createTable” has to be filled, otherwise the Word document cannot process the table correctly and if one tries to open it an error message will occur.

```
--Create Table
tab=doc~createTable(2,2) -- All rows and coloumns have to be filled

--Fill all Creates Tables
row1=tab~getRow(0)
row1~getCell(0)~setText("Row 1/1")
row1~getCell(1)~setText("Row 1/2")

row2=tab~getRow(1)
row2~getCell(0)~setText("Row 2/1")
row2~getCell(1)~setText("Row 2/2")
```

Figure 6 shows that we have added additional cells beside the ones created with the “createTable” method. The first thing we can do is create an additional row with the “createRow” method. This method will subsequently create cells with as many columns as are defined in that moment. In this case two columns have been defined thus two cells will be added. The next method we can use is called “addNewTableCell” and which allows us to add a cell to a row. Furthermore, the cell was edited using “setColor”.

```
row3=tab~createRow()
row3~getCell(0)~setText("Row 3/1")
row3~getCell(1)~setText("Row 3/2")

--Add additional Cells
row1~addNewTableCell()~setText("I have been added separately")

--Get Cell to edit it
row1~getCell(2)~setColor("3ca832") -- Cell Colour
```

5.1.3 Edit Cell Text and Save the File

What was not done so far is editing the text inside the cell. This can simply be done by creating a text run for the cell one wants to edit. If the text run is created all the methods of the class “XWPFRun” are available for text editing. The code below shows that even though a paragraph is created when the cell is created, we have to add another one to create a run. To remove the unnecessary empty paragraph the

method “removeParagraph” is used. To separate booth tables, another paragraph has been created in the lines 1-2.

```
1  p2=doc~createParagraph()
2  p2~setAlignment(.ParagraphAlignment~CENTER)
3
4  tab2=doc~createTable()
5  p=tab2~getRow(0)~getCell(0)~addParagraph()
6  r=p~createRun()
7  r~setFontSize(20)
8  r~setColor("FF0000")
9  r~setText("Red")
10 tab2~getRow(0)~getCell(0)~removeParagraph(0)
```

To finish the first word application the same exact steps as for Excel applications are necessary. This implies that, one needs the “fileOutputStream” class. The constructor allows us to the specify the path and the filename of our word document. Since the application should run on every operating system this the directory where the file should be saved is retrieved automatically. At last, we use the method “write” to add the document to the instance of “FileOutputStream” we just created.

```
1 --Save the file
2 parse source . . a
3 call directory filespec('L', a)
4
5 filename = directory()"/Table.docx"
6 filename = qualify(filename)
7
8 output=.bsf~new("java.io.FileOutputStream",filename)
9 doc~write(output)
```

5.2 Example 2 – ImageAndReplace.docx

For students and working persons alike writing reports is a common task. Reports have to convey complex information; thus, pictures are a helpful way to avoid confusions for readers. This example will consequently focus on adding images to a Word document. The second problem when writing reports is to correct mistakes, therefore the second part of this examples shows how to replace certain words inside a text run. The output of this example can be seen in Figure 7 below:

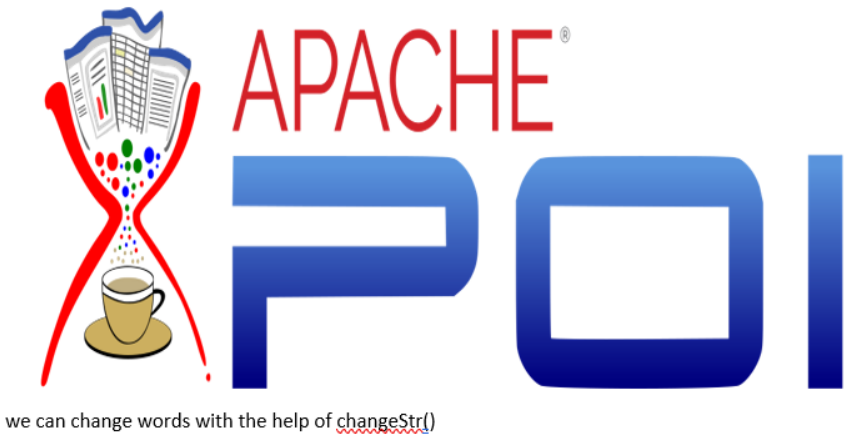

Figure 7: Output of Image+Replace.rexx

5.2.1 Add an Image

To get started one has to setup a Word document. This means creating the document, adding a paragraph and adding a run. The exact methods and classes that have been used for this procedure have already been shown in the first example.

To add an image, one has to resort to the method "addPicture" and add it to the run where the image should be featured. This method features five different parameters: "addPicture(java.io.InputStream pictureData, int pictureType, java.lang.String filename, int width, int height)"

```
1  --Add Image
2  .bsf~bsf.loadClass("org.apache.poi.xwpf.usermodel.Document", "Document")
3  .bsf~bsf.importClass("org.apache.poi.util.Units", "Units")
4
5  parse source . . b
6  call directory filespec('L', b)
7
8  filename1 = directory()"/ApachePOI.png"
9  filename1 = qualify(filename1)
10
```

```
11 img=.bsf~new("java.io.FileInputStream", filename1)
12
13 r1~addPicture(img, .Document~Picture_Type_Png, "ApachePOI.png", .Units~toEmu(417), .Units~toEmu(200))
```

As one can see in the code lines above, one needs to import two different classes and one interface to fully integrate an image into a word document. The first one that we imported is the interface "Document". A interface only features constants or abstract methods and subsequently it can only be instantiated with the method "bsf.loadClass". In this case the interface allows us to determine the picture type. The second variable that is needed is the width and length of the image. They have to be added in points and converted to English Metric Units (EMU). To do this we have to use the method "toEmu" of the "Units" class we implemented. The conversion is approximately 28.329 points for one centimetre. In our case the width is 14.73cm long and the height is 7.06cm. The last class that is needed is the "FileInputStream" class which allows us to import our image from the directory where it is saved. The directory is passed via the variable "filename1" because its path is retrieved automatically just like the process, we are using to save our files. The full process is explained in the Chapter *4.1.4 Change Specific Cells and Save the File*. The return value of "FileInputStream" is stored in the variable "img". Now the only thing that is missing is to add the name of the image to the method and the image will be successfully displayed in the word document.

5.2.2 Replace a Word

The second part of the application will focus on removing a word out of a run and replacing it with another. In our case the word "replace" was replaced with "change". The original text of the Word document can be seen in line 5 and the new one in Figure 7. To remove a word, one has to extract the text from each run. In this case there was only run available so extracting it was simply done with the "getText" method. Subsequently, a filter was added with the help of an if statement and the "contain" method. In this particular case the filter is not a necessity, since there is only one run. Nevertheless, in a case with many different runs it is necessary to pick the correct runs that have to be altered. After picking the correct run the word is swapped with the Rexx method "ChangeSTR", because the Java String, which is returned by the "getText" method, is automatically converted to an ooRexx string. To finish the example, we set our newly changed text for the run with a different "setText" method than used

priorly. The second parameter of this method allows us to define the position of the newly change text in the array. If we would not add the position zero, a second string would be created and added to the altered text run.

```
--Replace Text
run=p1~createRun()
run~setText("we can replace words with the help of changeStr()")


text=run~getText(0)
if text~contains("replace") then
do
   text=text~changeStr("replace", "change")
end
run~setText(text, 0)
```

Now that we have finished, we can follow the steps given in *4.1 Example 1 – Date.xlsx* to save and name our file correctly.

6. Microsoft PowerPoint

Microsoft PowerPoint was originally developed by a company called Forethought, Inc. Only three months after its release it was acquired by Microsoft to complement Microsoft Office. PowerPoint is the world's most common presentation application (Wikipedia.org, 2021). Apache POI offers Java implementation for PowerPoints file format before 2007, called POI-HSLF and a implementation called POI-XSLF for versions that use the newer file format (The Apache Software Foundation, 2021).

The following examples will show the functionality of the most important java classes available for POI-XSLF. All code lines are, as in the prior chapters, kept simple to ensure the focus is on the functionality of Apache POI. The Nutshell examples can easily be extended and used for whatever purpose they are needed.

6.1 Example 1 – Basics.pptx

The first application will focus on the basics of creating a PowerPoint presentation. The first slide will show how to add text and subsequently edit it. The second slide will demonstrate how to create a list. Lastly the third slide will show how to change the individual layouts of the slides. The output of the first application can be found in Figure 8

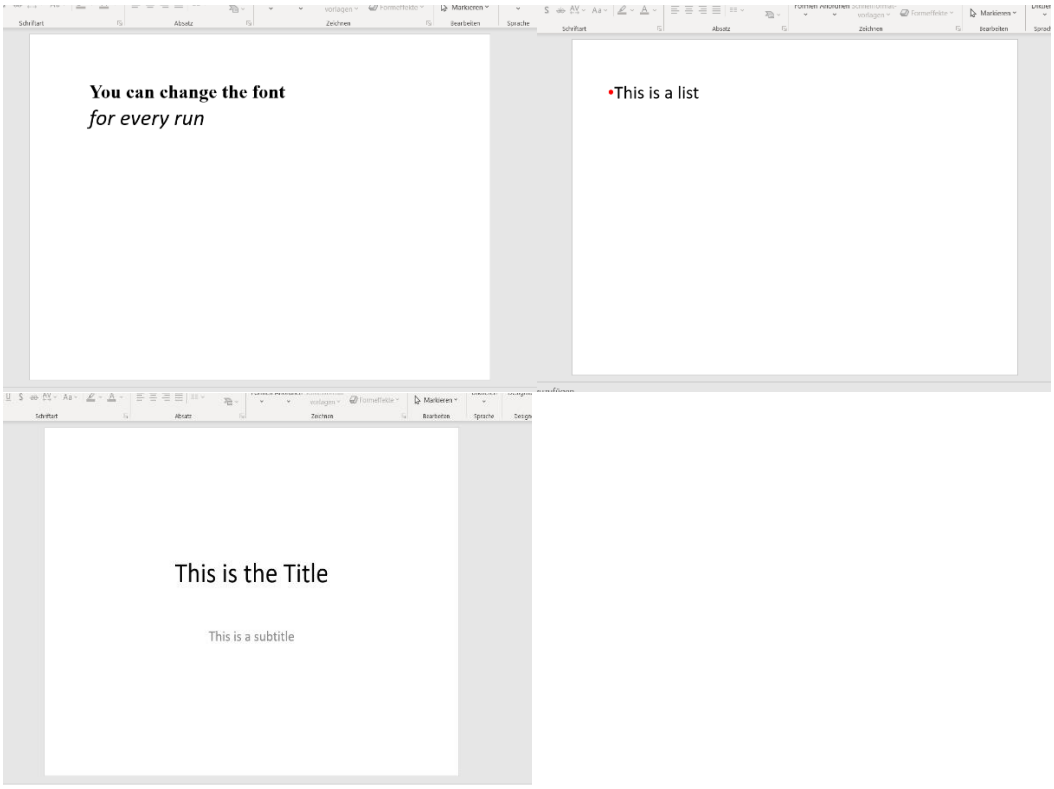

Figure 8: Output of PowerPointExample1.rexx (Slide 1 -3)

6.1.1 Create PowerPoint and Slides

The first step that is necessary for every one of the individual slides is creating the slide itself. To do this we need an instance of the "XMLSlideShow" class and the method "createSlide". This is very similar to the Excel and Word examples.

```
ppt=.bsf~new("org.apache.poi.xslf.usermodel.XMLSlideShow")

slide=ppt~createSlide()
```

6.1.2 Create a Text Box and Edit the Text

Adding text to PowerPoint slide is a bit more complex than adding it to a Word document. It basically works the same as if one would do it manually, which means one has to create and position a text box first.

To create a text box, use the "createTextBox" method. The position will be set by the method "setAnchor", which uses the constructor of the class "Rectangle". The first two variables inside the parameter define the height and width and the second pair defines the x and y coordinates. After creating the text box, we add an paragraph to the text box. Optionally one can define the indent level with the method "setIndentLevel".

```
--Create all prequesites for adding a Text run
shape=slide~createTextBox()
.bsf~bsf.importClass("java.awt.Rectangle", "Rectangle")
shape~setAnchor(.Rectangle~new(50, 50, 400, 200)) -- TextBox has to be
positioned
p=shape~addNewTextParagraph()
p~setIndentLevel(1) -- Set Indent Level for the Whole Paragraph
(optional)
```

Once the text box with the paragraph is created on can start adding and editing the text with the help of a text run. This basically works the same as for booth Word examples. As one can see below, after the creation of the run, the text is added, and all subsequent alterations are in turn added to the run with their respective methods. The code for the second run which contains "for every run" seen in Figure 8. Is not displayed but can be found in the *Appendix*.

```
--Format first Run
run=p~addNewTextRun()
run~setText("You can change the font")
run~setBold(1)
run~setFontFamily("Arial")
run~setFontSize(30)
```

6.1.3 Create Bullet Points

The second slide in this example shows how the reader can create a list of bullet points for the PowerPoint presentation. Since a list is just another form of text, we have to create another text box for the second slide. However, instead of adding a run directly to the paragraph it is necessary to first use the method "setBullet", which needs a Boolean values as parameter. There are two other "setBullet" methods, two of which use the class "ListAutoNumber" to create a numbered list. After labelling the paragraph as a bullet point, we can edit the size and colour of the it. In this example only the colour was edited with the "setBullteFontColor" that requires input in form of Enum constants of the "Color" class. The final step is to add the text run to the paragraph that is now set as a list of bullet points. The run can further be edited.

```
--Create all prequesites for adding a TextRun
form=slide2~createTextBox()
form~setAnchor(.Rectangle~new(50, 50, 400, 200))
p2=form~addNewTextParagraph()

--Add a list
p2~setBullet(1)
```

```
 8 .bsf~bsf.importClass("java.awt.Color", "Color")
 9 p2~setBulletFontColor(.Color~RED) --Bullet can be edited
10
11 --Add a TextRun to the list
12 run=p2~addNewTextRun()
13 run~setFontSize(30)
14 run~setText("This is a list")
```

6.1.4 Add a Layout and Save the File

The third slide in this application features a pre-set layout. To gain access to all the available layouts we need to access the slide master. One can do this with the "getSlideMaster" method. In addition, the "get" method is necessary to not only retrieve the slide master but to make the layouts accessible for our next method. As this is a high-level alteration of the document, we need the ppt object that was created at the start by the constructor of the class "XMLSlideshow". Once this is successfully done one can retrieve the actual layouts with one of the two "getLayout" methods. In this case we choose from the various Enum constants available in the class "SlideLayout". The line 1 of the second code snippet shows the second "getLayout" method that uses a string with name of the layout as parameter. The layout that was chosen is then saved with the variable "layout"

```
1 template=ppt~getSlideMasters()~get(0)  --retrives all master slides
2 .bsf~bsf.importClass("org.apache.poi.xslf.usermodel.SlideLayout", "SlideLayout")
3 layout=template~getLayout(.SlideLayout~TITLE) -- Set Layout
```

```
1 layout=template~getLayout("title slide")
```

This layout is then passed on via the "createSlide" method and the slide with the layout is successfully created. As one can see in the snippet below, the placeholder has to be accessed. This can be done with the "getPlaceholder" method and the corresponding index of the placeholder. In this example there are two placeholder that means the first one is accessed with zero and the second one with one. For the title box one can directly set the text that will be displayed with "setText". However, if the text is set directly without an instance of "XSLFTextRun" it cannot be further edited. The second placeholder that is accessed is the subtitle box. The subtitle box of the text displays a sample text, which has to be removed. Once we get the placeholder this is done with the method "cleartext". The next steps in line 9-12 are the same as for every other text run that has been created so far

```
--Edit Title
slide3=ppt~createSlide(layout) -- Create Slide with prior set layout
title=slide3~getPlaceholder(0) -- Get Placeholder Box for the Title
title~setText("This is the Title")

--Edit Content
body=slide3~getPlaceholder(1) --Get Placeholder Box for the Content
body~clearText() -- Necessary to clear to placeholder Text
p=body~addNewTextParagraph
f=p~addNewTextRun
f~setText("This is a subtitle")
f~setFontSize(23)
```

Now that all the slides are finished the only thing that is missing is to save the file. This is just a repetition the first four examples. Nevertheless, the code is displayed below, out of the sake of completeness.

```
--Save the file
parse source . . a
call directory filespec('L', a)

filename = directory()"/Basics.pptx"
filename = qualify(filename)

output=.bsf~new("java.io.FileOutputStream", filename)
ppt~write(output)

::requires BSF.CLS
```

6.2 Example 2 – Read.pptx

The second nutshell example will explain how to alter and existing PowerPoint presentation. In addition, a hyperlink will be created and an image will be added. The PowerPoint presentation that has been altered is the one, that was created in 6.1 Example 1 – Basics.pptx, thus it can be found in Figure 8. The Output of the new application is very similar, nonetheless it is displayed below in Figure 9

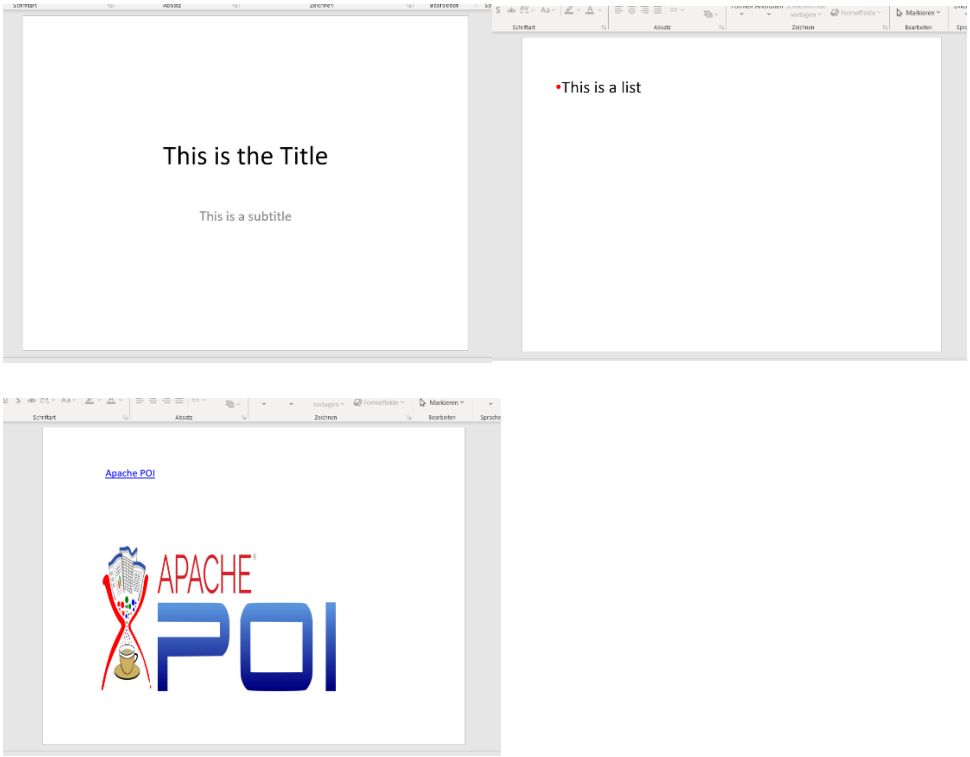

Figure 9: Output of PowerPointExample2.rexx (Slide 1 - 3)

6.2.1 Open the PowerPoint Presentation

As one can see the slides that were created in the first PowerPoint example were rearranged and one got deleted completely. In order to do so we have to create an “FileInputStream” object that we can pass on to the constructor of the “XMLSlideshow” class. This allows us to access existing PowerPoint presentations. We are once again using the standard procedure for automatically retrieving the path of the existing PowerPoint file.

```
parse source . . c
call directory filespec('L', c)


filename2 = directory()"/Basics.pptx"
filename2 = qualify(filename2)

pptold=.bsf~new("java.io.FileInputStream", filename2)
ppt=.bsf~new("org.apache.poi.xslf.usermodel.XMLSlideShow", pptold)
```

6.2.2 Change Slide Order and Remove Slides

After the PowerPoint is stored into the ppt variable one needs to access the single slides. The method "getSlides" lets us get all slides at once but with the "get" method one can subsequently choose the slide that has to be edited". As in almost every other example we start counting with zero and we pick the third slide in line 3 with the parameter two. Now we can rearrange the title slide in the correct position with the method "getSlideOrder". One has to add the slide that will b repositioned and the new position as seen in line 3. To remove the second slide the method "removeSlide" was used with the parameter one.

```
--Change existing Slides
slides=ppt~getSlides()
lastslide=slides~get(2)
ppt~setSlideOrder(lastslide, 0)
ppt~removeSlide(1)
```

6.2.3 Add a Hyperlink

The next thing on the agenda was to add a Hyperlink to the PowerPoint presentation. To do so a new slide with a text box was created just like in the prior example. To create a Hyperlink the methods "createHyperlink" and "setAddress" were used. The second method sets the actual internet address that will be opened, while the first adds the Hyperlink to the text run. It is important to mention that the in order to write a text again, without and underlying Hyperlink, a new text run has to be created.

```
run=p~addNewTextRun()
run~setText("Apache POI")
link=run~createHyperlink()
link~setAddress("https://poi.apache.org")
```

6.2.4 Add an Image

Adding an Image to a PowerPoint presentation requires a little more afford than adding an image to a

word document. The reason being that the “addPicture” method, requires two specific parameters. At first the return value of the “FileInputStream” class needs to be converted into the byte format. We can do this with the method “toByteArray” of the “IOUtils” class.

The second obstacle is that the class “PicturyType” that contains the Enum constants for the second parameter of “addPicture” is the inner class of the “PictureData” class. Thus, it cannot be instantized as normal classes. In line 5 the correct way to instantiate an inner class can be found. The $ sign allows us to access the inner class. Once we added all the information to our variable, it is subsequently used for the parameter of the “createPicture” method. The last step is to position our image, which works exactly the same as for text boxes.

```
1  --Add an Image
2  .bsf~bsf.importClass("org.apache.poi.util.IOUtils", "IOUtils")
3
4  parse source . . b
5  call directory filespec('L', b)
6
7  filename1 = directory()"/ApachePOI.png"
8  filename1 = qualify(filename1)
9
10 img=.bsf~new("java.io.FileInputStream", filename1)
11 imgData=.IOUtils~toByteArray(img)
12
13 innerClz=bsf.importClass("org.apache.poi.sl.usermodel.PictureData$PictureType")
14
15 pd=ppt~addPicture(imgData, innerCLz~PNG)
16
17 image=slide~createPicture(pd)
18 image~setAnchor(.Rectangle~new(100, 200, 400, 250))
```

To finish the application simply resort to the steps shown in *4.1 Example 1 – Date.xlsx*

7. Conclusion

The goal of this work was to create Apache POI nutshell examples in BSF4ooRexx to show the possibilities of booth Apache POI and BSF4ooRexx. In total six examples, two for each API available, have been created. These examples are not relying on difficult programming concepts, and are thus easily divisible, to ensure that future readers can successfully study and implement the functions of Apache POI in their own applications. Overall, every module of Apache POI provides manifold functions. It provides powerful tools that can be very useful for all kind of business contexts. While writing this paper, it was very clear that the XSSF implementation is, as stated on the Apache Website (Apache Software Foundation, 2021), the most developed implementation.

Even though comparing the three different Apache POI class libraries has not been the main task of this work differences were still noticeable. In particular, for methods that differ significantly, despite providing the same function. A good example for this is the method that is used to set the font of a text. For SSFX it is called “setFontName” while for XWPF and XSLF it is called “setFontFamily”

References

Apache Software Foundation. (2021, January 20). *Apache POI - the Java API for Microsoft Documents*. Retrieved May 5, 2021, from https://poi.apache.org/index.html

Flatscher, R. G. (2012). Automatisierung mit ooRexx und BSF4ooRexx. *Gesellschaft für Informatik e.V*, pp. 307-318.

The Apache Software Foundation. (2021, April 10). *Apache POI - HWPF and XWPF - Java API to Handle Microsoft Word Files*. Retrieved May 23, 2021, from https://poi.apache.org/components/document/index.html

The Apache Software Foundation. (2021, April 10). *POI-HSLF and and POI-XLSF - Java API To Access Microsoft Powerpoint Format Files*. Retrieved May 18, 2021, from https://poi.apache.org/components/slideshow/index.html

The Apache Software Foundation. (2021, January 17). *POI-HSSF and POI-XSSF/SXSSF - Java API To Access Microsoft Excel Format Files*. Retrieved May 10, 2021, from https://poi.apache.org/components/spreadsheet/

Wikipedia.org. (2021, January 19). *Apache POI*. Retrieved May 5, 2021, from https://en.wikipedia.org/w/index.php?title=Apache_POI&oldid=1001473049

Wikipedia.org. (2021, April 24). *Java (software platform)*. Retrieved May 5, 2021, from https://en.wikipedia.org/w/index.php?title=Java_(software_platform)&oldid=1019658740

Wikipedia.org. (2021, May 2021). *Java virtual machine*. Retrieved May 26, 2021, from https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=1025042195

Wikipedia.org. (2021, May 15). *Microsoft Excel*. Retrieved May 19, 2021, from https://en.wikipedia.org/wiki/Microsoft_Excel

Wikipedia.org. (2021, May 15). *Microsoft PowerPoint*. Retrieved May 18, 2021, from https://en.wikipedia.org/w/index.php?title=Microsoft_PowerPoint&oldid=1023230343

Wikipedia.org. (2021, May 23). *Microsoft Word* . Retrieved May 23, 2021, from https://en.wikipedia.org/w/index.php?title=Microsoft_Word&oldid=1024667574

Appendix

ExcelExample1.rexx

This application creates an Excel sheet with three cells, that are edited in various ways (font,colour…). Most importantly the standard type of a cell is changed.

```
/*
    Purpose: Excel sheet that show the date. The cells are additionally edited (Colour, Font...).
    Date:    July 2021


    ------------------------ Apache Version 2.0 license ----------------------
       Copyright 2021 Fabian Anton Rudolf Fuchs

       Licensed under the Apache License, Version 2.0 (the "License");
       you may not use this file except in compliance with the License.
       You may obtain a copy of the License at

           http://www.apache.org/licenses/LICENSE-2.0

       Unless required by applicable law or agreed to in writing, software
       distributed under the License is distributed on an "AS IS" BASIS,
       WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
       See the License for the specific language governing permissions and
       limitations under the License.
    ---------------------------------------------------------------------------
*/

workbook=.bsf~new("org.apache.poi.xssf.usermodel.XSSFWorkbook")
sheet=workbook~createSheet("Date")


-- Create Cells and add Content
row=sheet~createRow(0)
cell=row~createCell(0)
cell~setCellValue("Date:")

cellx=row~createCell(1)
cellx~setCellValue(Date())

--Edit Cell Style
font=workbook~createFont()
font~setFontHeightInPoints(30)
font~setFontName("Arial")
font~setBold(1)
font~setItalic(1)

--Set Cell Style to Specific Cells
style=workbook~createCellStyle()
style~setFont(font)
```

```
 45 cell~setCellStyle(style)
 46
 47 font2=workbook~createFont()
 48 font2~setFontHeightInPoints(33)
 49 font2~setFontName("Times New Roman")
 50
 51
 52 style2=workbook~createCellStyle
 53 style2~setFont(font2)
 54 helper=workbook~getCreationHelper()
 55 style2~setDataFormat(helper~createDataFormat()~getFormat("m/d/yy")) --
Set Cell Format
 56 cellx~setCellStyle(style2)
 57
 58
 59 --Set Cell Size
 60 a=cell~getColumnIndex()
 61 b=cellx~getColumnIndex()
 62
 63 do until a > b
 64     sheet~autoSizeColumn(a)
 65     a=a+1
 66 end
 67
 68
 69 cell=row~createCell(2)
 70 cell~setCellValue("!")
 71
 72.bsf~bsf.importClass("org.apache.poi.ss.usermodel.IndexedColors","IndexeColors")
 73
 74 style3=workbook~createCellStyle
 75 font3=workbook~createFont()
 76 font3~setColor(.IndexeColors~Red~getIndex())
 77 font3~setFontHeightInPoints(33)
 78
 79.bsf~bsf.importClass("org.apache.poi.ss.usermodel.BorderStyle","BStyle")
 80
 81 style3~setFont(font3)
 82 style3~setBorderBottom(.BStyle~Medium)
 83 style3~setBorderTop(.BStyle~Medium)
 84 style3~setBorderRight(.BStyle~Medium)
 85 style3~setBorderLeft(.BStyle~Medium)
 86
 87
 88 sheet~getRow(0)~getCell(2)~setCellStyle(style3) -- Change Specific
Cells without naming them
 89
 90 --Save the file
 91 parse source . . a
 92 call directory filespec('L', a)
 93
 94 filename = directory()"/Date.xlsx"
 95 filename = qualify(filename)
 96
 97 output=.bsf~new("java.io.FileOutputStream" , filename)
 98 workbook~write(output)
 99
100 ::requires BSF.CLS
```

ExcelExample2.rexx

This application creates an Excel sheet with the population development in Austria from 2000 until 2020. The data of the cells is subsequently used to create a Graph to better illustrate the development.

```
/*
    Purpose: Create an Excel sheet with a graph that shows the
development of the austrian Population from 2000 onwards.  .
    Date:    July 2021


    ------------------------ Apache Version 2.0 license ---------------
----------
        Copyright 2021 Fabian Anton Rudolf Fuchs

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

            http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing,
software
        distributed under the License is distributed on an "AS IS"
BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
        See the License for the specific language governing permissions
and
        limitations under the License.
    ------------------------------------------------------------------------
----------
*/

workbook=.bsf~new("org.apache.poi.xssf.usermodel.XSSFWorkbook")
sheet=.bsf~new("org.apache.poi.xssf.usermodel.XSSFSheet")
sheet=workbook~createSheet("Population")


--Row 1
row=sheet~createRow(0)

--Column 1
cell=row~CreateCell(0)
cell~setCellValue("2000")

--Column 2
cell=row~CreateCell(1)
cell~setCellValue("2010")

--Column 3
cell=row~CreateCell(2)
cell~setCellValue("2019")

--Row 2
row=sheet~createRow(1)

--Column 1
cell=row~CreateCell(0)
```

```
cell~setCellValue(8.002)

--Column 2
cell=row~CreateCell(1)
cell~setCellValue(8.352)

--Column 3
cell=row~CreateCell(2)
cell~setCellValue(8.859)


-- Create the Grapg
chart=.bsf~new("org.apache.poi.xssf.usermodel.XSSFChart")

drawing=sheet~createDrawingPatriarch()
anchor=drawing~createAnchor(0, 0, 0, 0, 1, 4, 7, 20)
chart=drawing~createChart(anchor)

-- Axes
.bsf~bsf.importClass("org.apache.poi.xddf.usermodel.chart.AxisPosition","Positio")

bottomAxis=chart~createDateAxis(.Positio~BOTTOM)
bottomAxis~setTitle("Year")
leftAxis = chart~createValueAxis(.Positio~LEFT)
leftAxis~setTitle("Population")


--Select Data
.bsf~bsf.importClass("org.apache.poi.ss.util.CellRangeAddress","CellRangeAddress")
address=.CellRangeAddress~new(0, 0, 0, 2)


dataSource=bsf.loadClass("org.apache.poi.xddf.usermodel.chart.XDDFDataSourcesFactory")

years=dataSource~fromNumericCellRange(sheet, address)

address1=.CellRangeAddress~new(1, 1, 0, 2)
population=dataSource~fromNumericCellRange(sheet, address1)


--Add data to the graph
data=.bsf~new("org.apache.poi.xddf.usermodel.chart.XDDFLineChartData")
.bsf~bsf.importClass("org.apache.poi.xddf.usermodel.chart.ChartTypes", "ChartTypes")

data=chart~createData(.ChartTypes~LINE, bottomAxis, leftAxis)
data~addSeries(years, population)
chart~plot(data)


--Save the file
parse source . . a
call directory filespec('L', a)
qualify(filename)

filename = directory()"/Graph.xlsx"
filename = qualify(filename)

output=.bsf~new("java.io.FileOutputStream" , filename)
workbook~write(output)

::requires BSF.CLS
```

WordExample1.rexx

This application creates a word document with text and a table. The text is well as the table is edited in multiple different ways (font, colour…).

```
/*
    Purpose: Creates a Word document filled with text, which is edited (Colour, font...).
             In addition a table will be created and filled with text.
    Date:    July 2021


    ------------------------ Apache Version 2.0 license ----------------------
        Copyright 2021 Fabian Anton Rudolf Fuchs

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

            http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
    ---------------------------------------------------------------------------
*/

doc=.bsf~new("org.apache.poi.xwpf.usermodel.XWPFDocument")

.bsf~bsf.importClass("org.apache.poi.xwpf.usermodel.ParagraphAlignment", "ParagraphAlignment")
p1=doc~createParagraph()
p1~setAlignment(.ParagraphAlignment~CENTER)

--Create simple Text and edit it
t1=p1~createRun()
t1~setText("This is the first run")
t1~addBreak()
t1~setText("We can edit each run as we like!")
t1~setFontSize(19)
t1~setBold(1)
t1~setItalic(1)
t1~setColor("0320fc")
t1~setFontFamily("Times New Roman")


--Create Table
tab=doc~createTable(2,2) -- All rows and coloumns have to be filled

--Fill all Creates Tables
row1=tab~getRow(0)
```

```
47 row1~getCell(0)~setText("Row 1/1")
48 row1~getCell(1)~setText("Row 1/2")
49
50 row2=tab~getRow(1)
51 row2~getCell(0)~setText("Row 2/1")
52 row2~getCell(1)~setText("Row 2/2")
53
54 row3=tab~createRow()
55 row3~getCell(0)~setText("Row 3/1")
56 row3~getCell(1)~setText("Row 3/2")
57
58 --Add additional Cells
59 row1~addNewTableCell()~setText("I have been added separately")
60
61 --Get Cell to edit it
62 row1~getCell(2)~setColor("3ca832") -- Cell Colour
63
64 p2=doc~createParagraph()
65 p2~setAlignment(.ParagraphAlignment~CENTER)
66
67 tab2=doc~createTable()
68 p=tab2~getRow(0)~getCell(0)~addParagraph()
69 r=p~createRun()
70 r~setFontSize(20)
71 r~setColor("FF0000")
72 r~setText("Red")
73 tab2~getRow(0)~getCell(0)~removeParagraph(0)
74
75
76 --Save the file
77 parse source . . a
78 call directory filespec('L', a)
79
80 filename = directory()"/Table.docx"
81 filename = qualify(filename)
82
83 output=.bsf~new("java.io.FileOutputStream",filename)
84 doc~write(output)
85
86 ::Requires BSF.CLS
```

WordExample2.rexx

The second Word application creates a word document with an image and text. The text has been altered, which cannot be seen in the output. The old text can be found in line 20.

```
1 /*
2     Purpose: Creates a Word document that shows an image and a line of
text.
3              The text has be altered, but the old text can only be found
in the code (line 42).
4     Date:    July 2021
5
6
7     ------------------------ Apache Version 2.0 license ----------------
---------
8        Copyright 2021 Fabian Anton Rudolf Fuchs
9
```

```
10          Licensed under the Apache License, Version 2.0 (the "License");
11          you may not use this file except in compliance with the License.
12          You may obtain a copy of the License at
13
14              http://www.apache.org/licenses/LICENSE-2.0
15
16          Unless required by applicable law or agreed to in writing,
software
17          distributed under the License is distributed on an "AS IS" BASIS,
18          WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
19          See the License for the specific language governing permissions
and
20          limitations under the License.
21      ----------------------------------------------------------------------
---------
22  */
23
24  doc=.bsf~new("org.apache.poi.xwpf.usermodel.XWPFDocument")
25
26  .bsf~bsf.importClass("org.apache.poi.xwpf.usermodel.ParagraphAlignment", "ParagraphAlignment")
27  p1=doc~createParagraph()
28  r1=p1~createRun()
29  p1~setAlignment(.ParagraphAlignment~CENTER)
30
31  --Add Image
32  .bsf~bsf.loadClass("org.apache.poi.xwpf.usermodel.Document", "Document")
33  .bsf~bsf.importClass("org.apache.poi.util.Units", "Units")
34
35  parse source . . b
36  call directory filespec('L', b)
37
38  filename1 = directory()"/ApachePOI.png"
39  filename1 = qualify(filename1)
40
41  img=.bsf~new("java.io.FileInputStream", filename1)
42
43  r1~addPicture(img, .Document~Picture_Type_Png, "ApachePOI.png", .Units~toEmu(417), .Units~toEmu(200))
44
45  p1=doc~createParagraph()
46  p1~setAlignment(.ParagraphAlignment~LEFT)
47
48  run=p1~createRun()
49  run~setText("we can replace words with the help of changeStr()")
50
51  --Replace Text
52  text=run~getText(0)
53  if text~contains("replace") then
54  do
55     text=text~changeStr("replace", "change")
56  end
57  run~setText(text, 0)
58
59  --Save the File
60  parse source . . a
61  call directory filespec('L', a)
62
63  filename = directory()"/PictureAndReplace.docx"
64  filename = qualify(filename)
65
66  output=.bsf~new("java.io.FileOutputStream", filename)
67  doc~write(output)
```

```
68
69 ::requires BSF.CLS
```

PowerPointExample1.rexx

The application creates a PowerPoint file with three different slides. One with only text, one with a list and the last one features a pre-set layout.

```
  1 /*
  2     Purpose: Creates a PowerPoint presentation which includes:
  3              A Slide that shows edited text.
  4              A Slide that demonstrates how to create a list.
  5              A slide with a changed Layout.
  6     Date:    July 2021
  7
  8
  9     ------------------------ Apache Version 2.0 license ---------------
----------
 10        Copyright 2021 Fabian Anton Rudolf Fuchs
 11
 12        Licensed under the Apache License, Version 2.0 (the "License" ;
 13        you may not use this file except in compliance with the License.
 14        You may obtain a copy of the License at
 15
 16            http://www.apache.org/licenses/LICENSE-2.0
 17
 18        Unless required by applicable law or agreed to in writing,
software
 19        distributed under the License is distributed on an "AS IS"
BASIS,
 20        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 21        See the License for the specific language governing permissions
and
 22        limitations under the License.
 23     ------------------------------------------------------------------
----------
 24 */
 25
 26 ppt=.bsf~new("org.apache.poi.xslf.usermodel.XMLSlideShow")
 27
 28 --Slide 1 / Empty Slide with simple Text
 29 slide=ppt~createSlide() --Creates Empty Slide
 30
 31 --Create all prequesites for adding a Text run
 32 shape=slide~createTextBox() -- Creates TextBox
 33 .bsf~bsf.importClass("java.awt.Rectangle", "Rectangle")
 34 shape~setAnchor(.Rectangle~new(50, 50, 400, 200)) -- TextBox has to be
positioned
 35 p=shape~addNewTextParagraph() -- Paragraph has to be added
 36 p~setIndentLevel(1) -- Set Indent Level for the Whole Paragraph
(optional)
 37
 38 --Format first Run
 39 run=p~addNewTextRun()
 40 run~setText("You can change the font")
 41 run~setBold(1)
 42 run~setFontFamily("Arial")
 43 run~setFontSize(30)
```

```
44
45 --Format second Run
46 run2=p~addNewTextRun()
47 run2~setText(" for every run")
48 run2~setItalic(1)
49 run~setFontFamily("Times New Roman")
50 run2~setFontSize(35)
51
52
53 --Slide 2 / Empty Slide with a list
54 slide2=ppt~createSlide() --Creates Empty Slide 2
55
56 --Create all prequesites for adding a TextRun
57 form=slide2~createTextBox()
58 form~setAnchor(.Rectangle~new(50, 50, 400, 200))
59 p2=form~addNewTextParagraph()
60
61 --Add a list
62 p2~setBullet(1)
63 .bsf~bsf.importClass("java.awt.Color", "Color")
64 p2~setBulletFontColor(.Color~RED) --Bullet can be edited
65
66 --Add a TextRun to the list
67 run=p2~addNewTextRun()
68 run~setFontSize(30)
69 run~setText("This is a list")
70
71
72 --Slide 3 / Slide with Layout and Text
73 template=ppt~getSlideMasters()~get(0)  --retrives all master slides
74 .bsf~bsf.importClass("org.apache.poi.xslf.usermodel.SlideLayout", "SlideLayout")
75 layout=template~getLayout("title slide") -- Set Layout
76
77
78
79 --Edit Title
80 slide3=ppt~createSlide(layout) -- Create Slide with prior set layout
81 title=slide3~getPlaceholder(0) -- Get Placeholder Box for the Title
82 title~setText("This is the Title")
83
84 --Edit Content
85 body=slide3~getPlaceholder(1) --Get Placeholder Box for the Content
86 body~clearText() -- Necessary to clear to placeholder Text
87 p=body~addNewTextParagraph
88 f=p~addNewTextRun
89 f~setText("This is a subtitle")
90 f~setFontSize(23)
91
92
93 --Save the file
94 parse source . . a
95 call directory filespec('L', a)
96
97 filename = directory()"/Basics.pptx"
98 filename = qualify(filename)
99
100 output=.bsf~new("java.io.FileOutputStream", filename)
101 ppt~write(output)
102
103 ::requires BSF.CLS
```

PowerPointExample2.rexx

The second PowerPoint application makes use of the output of PowerPointExample1.rexx, namely the file Basics.pptx. In this application the slide order gets changed and one slide is removed. Lastly a new slide with an image is added.

```
/*
    Purpose: Creates a PowerPoint presentation that shows how to alter and existing PowerPoint presentation.
             In addition, a hyperlink will be created and an image will be added to the slides .  .
    Date:    July 2021


    ------------------------ Apache Version 2.0 license ----------------------
       Copyright 2021 Fabian Anton Rudolf Fuchs

       Licensed under the Apache License, Version 2.0 (the "License");
       you may not use this file except in compliance with the License.
       You may obtain a copy of the License at

           http://www.apache.org/licenses/LICENSE-2.0

       Unless required by applicable law or agreed to in writing, software
       distributed under the License is distributed on an "AS IS" BASIS,
       WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
       See the License for the specific language governing permissions and
       limitations under the License.
    ---------------------------------------------------------------------------
*/
--  !Requires PowerPointExample2.rexx to run first!
parse source . . c
call directory filespec('L', c)


filename2 = directory()"/Basics.pptx"
filename2 = qualify(filename2)

pptold=.bsf~new("java.io.FileInputStream", filename2)
ppt=.bsf~new("org.apache.poi.xslf.usermodel.XMLSlideShow", pptold)

--Change existing Slides
slides=ppt~getSlides()
lastslide=slides~get(2)
ppt~setSlideOrder(lastslide, 0)
ppt~removeSlide(1)

slide=ppt~createSlide() --Creates Empty Slide

--Create all prequesites for adding a Text run
shape=slide~createTextBox() -- Creates TextBox
.bsf~bsf.importClass("java.awt.Rectangle", "Rectangle")
shape~setAnchor(.Rectangle~new(100, 50, 400, 250)) -- TextBox has to be positioned
p=shape~addNewTextParagraph() -- Paragraph has to be added
```

```
47
48 run=p~addNewTextRun()
49 run~setText("Apache POI")
50 link=run~createHyperlink()
51 link~setAddress("https://poi.apache.org")
52
53 --Add an Image
54 .bsf~bsf.importClass("org.apache.poi.util.IOUtils", "IOUtils")
55
56 parse source . . b
57 call directory filespec('L', b)
58
59 filename1 = directory()"/ApachePOI.png"
60 filename1 = qualify(filename1)
61
62 img=.bsf~new("java.io.FileInputStream", filename1)
63 imgData=.IOUtils~toByteArray(img)
64
65 innerClz=bsf.importClass("org.apache.poi.sl.usermodel.PictureData$PictureType")
66
67 pd=ppt~addPicture(imgData, innerCLz~PNG)
68
69 image=slide~createPicture(pd)
70 image~setAnchor(.Rectangle~new(100, 200, 400, 250))
71
72 --Save the File
73 parse source . . a
74 call directory filespec('L', a)
75
76 filename = directory()"/Read.pptx"
77 filename = qualify(filename)
78
79 output=.bsf~new("java.io.FileOutputStream",filename)
80 ppt~write(output)
81
82 ::requires BSF.ClS
```

Jar Files

Jar files used:
poi-ooxml-5.0.0.jar
poi-ooxml-full-5.0.0.jar
poi-integration-5.0.0.jar
poi-excelant-5.0.0.jar
poi-examples-5.0.0.jar
curvesapi-1.06.jar
xmlbeans-4.0.0.jar
commons-math3-3.6.1.jar
commons-collections4-4.4.jar
commons-codec-1.15.jar
sparseBitSet-1.2.jar
poi-scratchpad-5.0.0.jar
poi-ooxml-lite-5.0.0.jar
poi-5.0.0.jar
commons-compress-1.20.jar