WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Seminararbeit

SBWL: Business Information Systems

im **WS 2022**

ooRexx: Nutshell Examples for MS Word

Oppermann Sabrina 11904962

PI-Leitung

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

Wien, 15.12.2022

Welthandelsplatz 1, A-1020, Wien

Telefon: +43/1/313 36/4622 ● Telefax: +43/1/313 36/717

Web: www.wu.ac.at/retail

EFMD EQUIS ACCREDITED

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Eigenständigkeitserklärung

für Seminararbeiten, Aufgaben, Reflexionen, Prüfung

Lehrveranstaltungsnummer: 0082 **Semester:** WS 2022/23

Lehrveranstaltung: SBWL BIS Kurs V: Seminar aus BIS (Skiseminar)

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

VerfasserIn: Sabrina Oppermann

Matrikelnummer: h11904962

Ich versichere / stimme zu:

1. dass ich die hochgeladenen Arbeiten, Aufgaben, Reflexionen sowie die Prüfung selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
2. dass ich dieses Thema bisher weder im In- noch im Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Mit der Unterschrift nehme ich zur Kenntnis, dass falsche Angaben studien- und strafrechtliche Konsequenzen haben können.

_____15.12.2022_____ _____Sabrina Oppermann_____

Datum Unterschrift

Table of Contents

1 Introduction 1

2 Overview 3

2.1 Rexx and ooRexx 3

2.2 BSF4ooRexx and BSF.CLS 5

3 Nutshell Examples 7

3.1 Basic Functionalities 7

3.1.1 How to Open MS Word and Create a Text 7

3.1.2 How to Create Headings 10

3.1.3 How to Save a Document 14

3.1.4 How to Reopen a Document 15

3.1.5 How to Get into Print Preview 17

3.2 Special Functionalities 19

3.2.1 How to Create a Table 19

3.2.2 How to Edit a Table 22

3.2.3 How to Insert an Image 24

3.2.4 How to Insert a Text from a Webpage 27

3.2.5 How to Encode an Inserted Text 29

4 Summary and Conclusion 32

5 Appendix 34

5.1 Table of Figures 34

5.1.1 Codes 34

5.1.2 Executed Codes 35

5.1.3 Figures 35

5.2 References 36

1 Introduction

This paper puts its focus on the programming language “ooRexx” in cooperation with the program “MS Word” which is included in the proprietary office package “Microsoft Office”.

After reading this seminar paper, the reader should have understood the basic usage of MS Word through the ooRexx interface. In order to get a better comprehension, nutshell examples are provided and explained adequately. Additionally, images not only of the programming code but also of its resulting execution enable a visual exemplification.

The structure of this paper consists of 5 main chapters. After the introduction, the second chapter containing a brief overview of the used programming language is provided. Additionally, the functional package “BSF4ooRexx” is introduced to the reader and the related Rexx-package “BSF.CLS” is shortly explained. This should allow the reader to completely understand the different nutshell examples which are part of the third chapter. In turn, mentioned chapter is divided into 2 subchapters - each one explaining five representative nutshell examples. The last content-related chapter summarizes this paper in short and, subsequently, provides a conclusion. Last but not least, the appendix containing the table of figures and literature register are closing this work.

If the reader would like to execute the nutshell examples next to reading this paper, it is recommended to download the used programming language. The language “ooRexx” can be downloaded from the following website providing the most current version 5.0.0 beta: https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0beta/. Furthermore, the adequate version of BSF4ooRexx is needed to execute one of the examples. In order to download this version, the subsequent link is to be opened and respective program downloaded: https://sourceforge.net/projects/bsf4oorexx/files/GA/BSF4ooRexx-641.20220131-GA/. It has to be taken into account to fulfil all the necessary requirements a computer needs in order to execute the program as well as download the suitable version for the

respective operating system.
In addition, the proprietary version “MS Word” is needed. Either the user is a student of the university “WU” and can download the “MS Office” package for free through the inherent student account. Or the reader needs to buy the “MS Word” program on the following website: https://www.microsoft.com/de-at/microsoft-365/word?activetab=tabs%3afaqheaderregion3 .

2 Overview

This chapter deals with the basic knowledge about the programming language ooRexx. It is thereby arranged into 2 subchapters. First, the reader is presented some aspects regarding the development of ooRexx starting with Rexx. Afterwards, a short insight about the external Rexx-function-package “BSF4ooRexx” and the ooRexx-package “BSF.CLS” is given.

2.1 Rexx and ooRexx

Open Object Rexx (short ooRexx) is an open-source program which enables users to profit from the free version of the programming language “Object Rexx”. The latter one was originally published by the IT and consulting enterprise “IBM” as a proprietary, follow-up version of “Rexx” (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 4).

Putting the focus on the programming language “Rexx” for a while, Rexx was first established in 1979 by Mike Cowlishaw who was working for IBM. His goal was ‘to create a “human-centric” language’ (Flatscher, An Introduction to Procedural and Object-oriented Programming (ooRexx), N.G., p. 5). Thereby this new language should be an easy-to-learn and simplified version of other PL/I programming languages. Therefore, Rexx is seen as an antecessor of TCI as well as Python. Originally, this scripting language was intended to replace the programming languages “EXEC” and “EXEC 2” (tutorialspoint, N.G.).

After facing some problems with patents, the original name “Rex” was turned into “Rexx” (ProTech, N.G.). Therefore, “Rexx” is an abbreviation for “REstructured eXtended eXecutor” (Flatscher, Procedural and Object-oriented Programming 1, 2022, p. 5). Although this language might not be as known as others such as “C”, it provides the user with a lot of useful features to learn and use Rexx in an easy way. Commands based on the English language, a smaller number of rules and functions which are already built-in are just a few examples of the various benefits of Rexx (Rexx Language Association, 2015). In Figure 1 the logo of named programming language is presented.

Figure 1: Logo of Rexx - URL: https://upload.wikimedia.org/wikipedia/en/f/f7/Rexx-img-lg.png

Years later, IBM published the subsequent version called “Object Rexx” (acronym “oRexx”). This version is to be object-oriented and as easy to handle as its predecessor. Speaking of the latter, “oRexx” was created in a way which enables users to still use the already existing Rexx program which was an important issue to a lot of users (Flatscher, Automatisierungssprache Open Object Rexx 5.0 vor der Tür - Menschenfreund, 2017). Moreover, another new feature is the possibility to command multiple environments right from the Rexx interface (Rexx Language Association, 2015).

In 2005, the first open-source version named “ooRexx” was made available to the public by the non-profit organization “Rexx Language Association” (acronym “RexxLa”) in order to publish and develop the source code further. Moreover, named scripting language is made available to the most important operating systems - Linux, Windows, and MacOSX - in not only 32-Bit but also 64-Bit version (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 309). In Figure 2, the reader is presented the typical logo of ooRexx.

Figure 2: Logo of ooRex - URL: https://avatars.githubusercontent.com/u/11989843?s=280&v=4

Putting the focus on RexxLa, it was thereby founded to spread the knowledge about Rexx. This international association has its headquarter situated in North Carolina and members all over the world (RexxLa, N.G.).

12 years later, the most current version of ooRexx called “ooRexx 5.0” was released. This new publication includes new features like an easier handling of arrays as well as the processing of meta data with the help of annotations and RESOURCE-directives (Förster, 2017).

2.2 BSF4ooRexx and BSF.CLS

Turning the focus to BSF4ooRexx, the end of the operating system IBM OS/2 led to the intention to build a bridge for Rexx programmers to the operating systems of Linux and Windows. The original purpose of BSF4ooRexx was to enable the access to the programming language Java from all such platforms to save costs of investment. After a successful proof-of-concept work in 2000, an external Rexx-function-package was developed. This was achieved by applying the open Java-class-library “Bean Scripting Framework of Apache Software Foundation” which was originally used by IBM employees. The external function-package named “BSF4ooRexx” was born. BSF4ooRexx is thereby an abbreviation for “Bean Scripting Framework for ooRexx”. With the help of this package, the user is able to use its functionalities in Rexx. Next to the ability of the “Augsburg” version for no need to interact with Java in a strictly typed way, the “Vienna” version of BSF4ooRexx supplies the user additionally with the feature of a support for Rexx for OpenOffice.org (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 312f).
In **Error! Reference source not found.** the new logo of this package is depicted. This logo connects graphically both ooRexx and BSF through the combination of the original logo of ooRexx and beans representing “BSF”.

Figure 3: Logo of BSF4ooRexx, URL: https://www.rexxla.org/images/BSF4ooRexx.png

In connection with the BSF4ooRexx package, the Rexx-package “BSF.CLS” enables the user to interact with the programming language Java as easy as possible. Thereby the user does not need to know the Java language itself. The whole Java-class-library and the communication with Java-objects is presented as if a communication just within ooRexx is taking place. A lot of advantages are therefore established. For example, using BSF.CLS, all functionalities of Java are available to the ooRexx programmer (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 313). Since the Java Runtime Environment is already installed on most of the computers, the user just needs to download “BSF4ooRexx” additionally (Flatscher, "The 2009 Edition of BSF4Rexx", 2009, p. 2).

3 Nutshell Examples

This chapter consists of 10 nutshell examples controlling the Microsoft Office program “Word” completely automatically via the respective ooRexx code. Those examples form part of 2 chapters, namely “Basic Functionalities” and “Special Functionalities”. Within each subchapter, a short explanation will first state the purpose of the example. Afterwards, the respective code is provided to the reader and referred to continuously by explaining the individual code lines and its resulting process. Last but not least, the executed code is shown visually in pictures.

In the following subchapters different nutshell examples are explained to the reader. All of them are in relation to the Microsoft Office program “MS Word”. Since this paper does not comprise basic explanations of the Rexx language itself but only of those in relation to the use of MS Word, it is recommendable to get familiar with the basic functionalities of Rexx before continuing reading. In this case the following link is to be clicked at to get to the IBM website containing information to the usage of Rexx: https://www.ibm.com/docs/en/zos/2.1.0?topic=tsoe-zos-rexx-users-guide .

3.1 Basic Functionalities

In this chapter, the general basic functions of ooRexx in combination with MS Word are stated. The following 5 nutshell examples present the way to open MS Word, create a text in a document as well as headings, save it afterwards, reopen a saved document and, finally, get into the print preview. After reading this chapter, the programmer is able to use MS Word to, for instance, write essays or papers using stated features.

3.1.1 How to Open MS Word and Create a Text

Starting with opening MS Word and regulating the size of the appearing window, this nutshell example deals with the respecting code. Additionally, the way of creating a

text within the new document is presented. Furthermore, the user is instructed in how to edit a text in form and style.

```
1    /*open MS Word*/
2    word= .oleobject~new("Word.Application")
3    word~visible=.true
4    word~width=.true
5    word~height=.true
6
7    /*create new document*/
8    NewDocument= word~Documents~add
9
10   /*create text*/
11   SelectionObj=Word~Selection
12   SelectionObj~TypeText("This is an arbitrary text")
13
14   SelectionObj~TypeParagraph
15
16   /*edit text*/
17   EditText=SelectionObj~Font
18   EditText~Name="Papyrus"
19   EditText~size="20"
20   EditText~bold=.true
21   EditText~underline=.true
22   EditText~colorindex= 9
23   SelectionObj~TypeText("This text is edited")
```

Code 1: Nutshell Example 1 - How to Open MS Word and Create a Text

Looking at the second line in Code 1 above, the command *.oleobject* is used. OleObject is hereby a proxy-class which enables the user to command and interact with various Windows-COM/OLE-programs (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 311). In this case the application “MS Word” is needed, so the subsequent command *~new(“Word.Application”)* is given to open a new Word window. By creating a variable, the user must not repeat this first command to facilitate

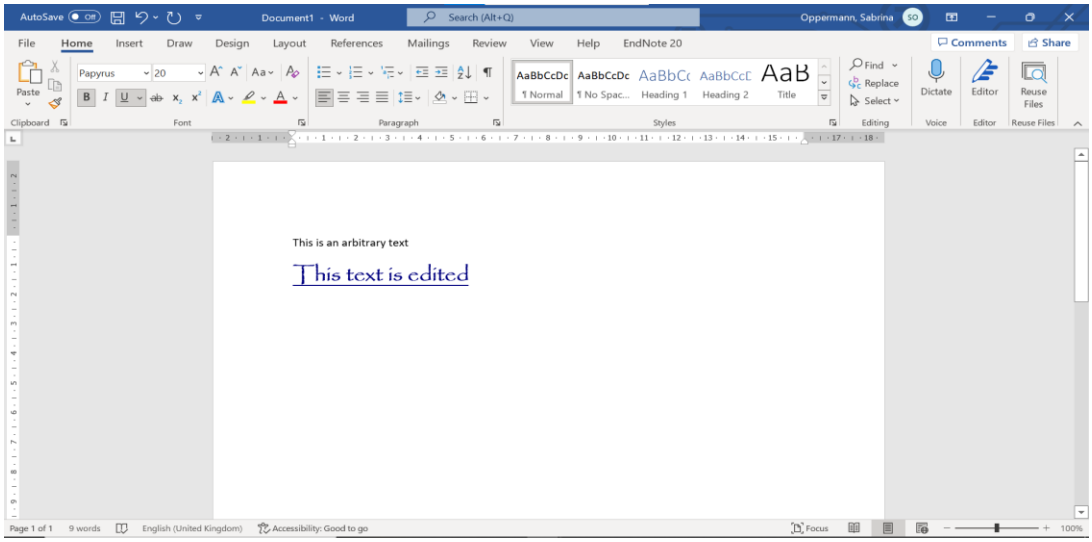
its usage. In the next step, the newly created window is to be made visible by using the function *word~visible=.true*. Now, the window is displayed to the user and if required the size of respective can be adjusted manually. This can be achieved by using the commands *word~width=…* and *word~height=…* . In this example, the window should take up the size of the entire screen. To enable a full screen the commands regarding width and height must be set to *.true*. If a specific size is desired, the user can enter the respective dimensions at will. Up to this part the programming code is freely adjustable to any to-be-opened program needed by the user.

After opening an empty Word window, a new document has to be created. In order to do so, the command *word~Document~add* is used and assigned to the variable *NewDocument*. If the programming code is executed to this extend, the result equals the manual clicking on the Word button on your desktop and opening of a blank sheet: a new and completely empty Word document is created.

In the following step, content in form of a text is to be added to the document. By using the code *word~selection*, which is written in line 12 in Code 1, the user gains access to the document for any desired manipulation. In this example the corresponding variable is named “SelectionObj”. With the help of the command *~TypeText(“…”)* any text construction may be entered to the document. In this case, the text “This is an arbitrary text” is inserted in the document. When entering plain text without following commands regarding text style, it is depicted in the standard “normal” style format of MS Word. The first line in Executed Code 1 below shows the text after the performance of the programming code.

If desired, any text can be edited. First, for a better overview, a paragraph is to be entered after the first text. This is achieved by using the command *~TypeParagraph*. Now, continuing with editing, the command *~Font* is applied to the variable *SelectionObj* to gain access to the style features of the latter. This command is allocated to the variable named *EditText*. In this example, the text “This is an edited text” should be changed in style. First, the font style is to be altered to “Papyrus” and size “20” by using the command *~Name* and *~size* (see Code 1, line 20 and 21). Second, the commands *~bold* and *~underline* are set equal to *.true* in order to adapt

the text respectively. Last but not least, the color is to be changed to darkblue by using ~colorindex and the corresponding color-figure. The final result is seen in the second line of Executed Code 1.



Executed Code 1: Nutshell Example 1

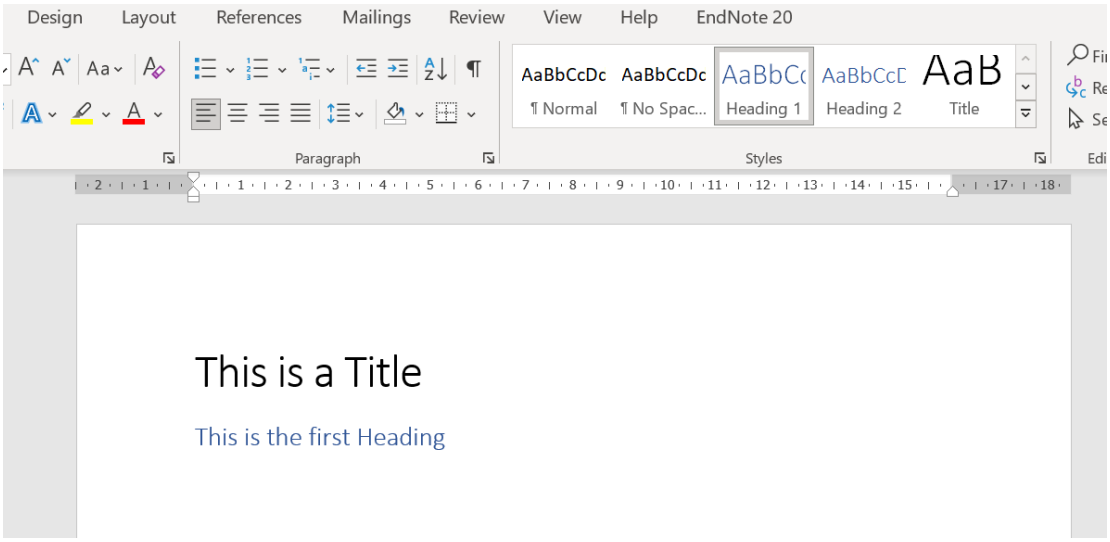
3.1.2 How to Create Headings

When writing an essay, the user needs to divide the text in chapters by using different headings. In this nutshell example, the way to use pre-set style formats like “Title” and “Heading” are explained. Furthermore, these styles are made adjustable to the user’s likings by changing the font-size, -color, etc.

```
word=.oleobject~New("Word.Application")
word~visible=.true

NewDocument=word~Documents~add

/*Get style of the title*/
SelectionObj=word~selection
SelectionObj~Style="Title"
SelectionObj~TypeText("This is a Title")
SelectionObj~TypeParagraph

/*Get style of the first heading*/
SelectionObj~Style="Heading 1"
SelectionObj~TypeText("This is the first Heading")
```

Code 2: Nutshell Example 2a – How to Create Headings

Continuing with the inspection of the programming code in Code 2, the first four lines to open a new document are already explained in the first subchapter. In the 7th line, the code to gain access to the document itself is also already familiar. In the next step, the style “Title” is selected by the command ~*Style*. Here, any desired style-name can be entered like it is stated in line 13 using “Heading 1”. After classifying the style, the corresponding text for the title and heading is to be put in. The result of this part of the code is presented in Executed Code 2 below. As it is depicted, the standard style format for the title and first heading are used.



Executed Code 2: Nutshell Example 2a

In case the user is not satisfied with those pre-set style formats one can see in Executed Code 2, the following code allows to adapt the style as desired. Not only the font style but also the size, color and other characteristics can be edited at wish.

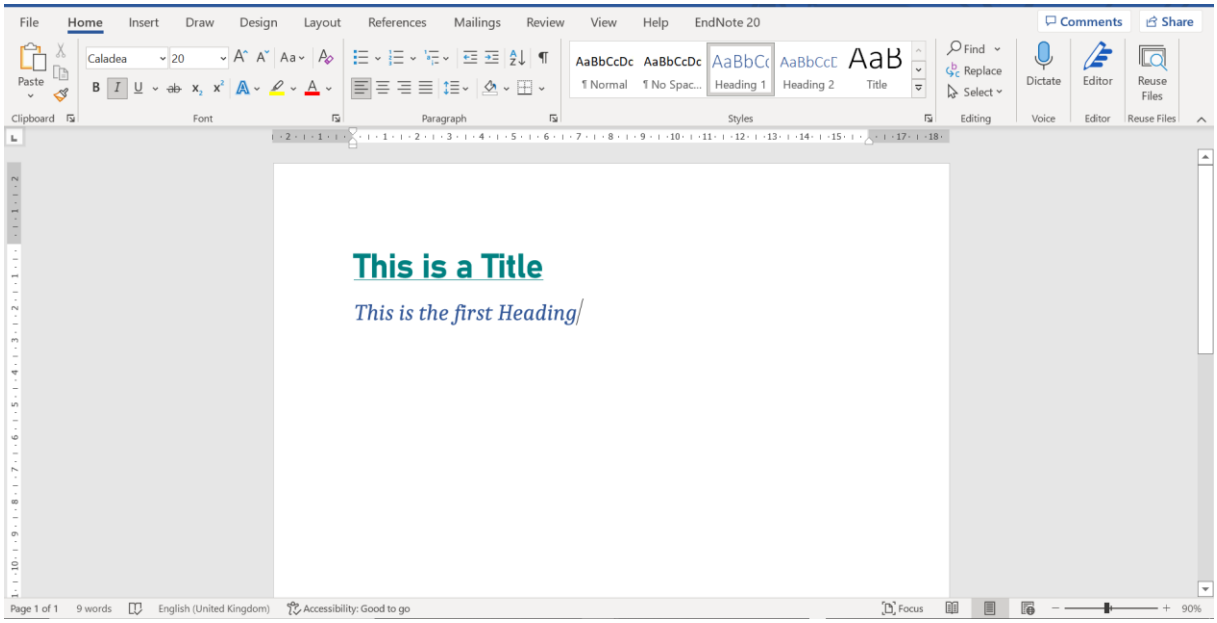
```
16   /*Adapt each style*/
17   /*count the sentences*/
18   CountSentences=newDocument~Sentences~Count
19   Font=SelectionObj~Font
20   do SentenceNumber= 1 to CountSentences
21
22   /*change each style*/
23   newDocument~Sentences(SentenceNumber)~Select
24   StyleName=SelectionObj~Style~NameLocal
25   Select case StyleName
26    when "Title" then do
27          Font~Name="Bahnschrift"
28          Font~Size="30"
29          Font~Colorindex= "10"
30          Font~bold=.true
31          Font~underline=.true
32    end
33    when "Heading 1" then do
34          Font~Name="caladea"
35          Font~Size="20"
36          Font~italic=.true
37    end
38    otherwise NOP
39    end
40   end
```

Code 3: Nutshell example 2b - How to Create Headings

First, ooRexx should select the sentences. To achieve this, the number of sentences is to be counted and saved in the variable *CountSentences* by using the command in line 18 of Code 3. Afterwards, access to the style features is gained and applied to the variable *Font*. In the following line, a do-command is used to go through each sentence. To pick the respective sentence in the document, the expression *~select* is executed. Now, each sentence is selected one after the other. In the next step, the program should identify whether a sentence is a title, a heading or neither. After selecting one sentence, the respective style name is to be recognized by using the code in line 24.

Now, the aim is to only format the sentence if the style is either “Title” or “Heading 1”. This can be achieved by using *select case StyleName* in combination with a when-then-do-condition. The latter specifies what to do when the selected sentence is a “Title”, a “Heading 1” or neither of both.

In the example given, the font style, the size, the color of the “Title” is changed and turned into bold and underlined. Whereas the style of “Heading 1” only differs from the original format in regard to the font style, size and italics. This function is particularly useful if headings in a long text are to be changed not individually but all together by identifying whether it is a text to be changed or not. In the following figure Executed Code 3, the output of Code 3 is depicted.



Executed Code 3: Nutshell Example 2b

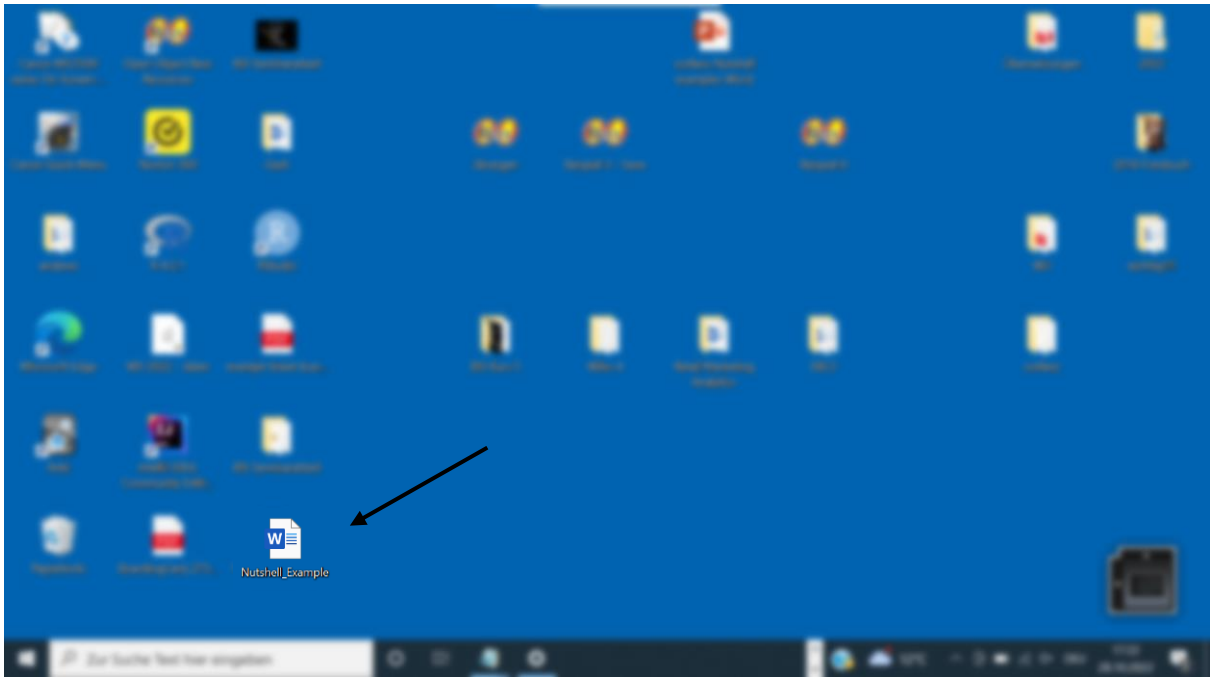
3.1.3 How to Save a Document

After being able to open Word, create a text and edit it, the user wants to save the document in most cases. This part of the paper deals with saving a document in a predetermined location on the user's computer.

```
1    /*Choose location to save document*/
2    Dir = Value("UserProfile",, ENVIRONMENT)
3    FileName= Dir || "\Desktop\" || "Nutshell_Example"
4
5    /*save document*/
6    NewDocument~SaveAs(FileName)
7    Word~Quit
8
```

Code 4: Nutshell Example 3 - How to Save a Document

The programming code in Code 4 can be added to any ooRexx code involving MS Word. First, the location where the document should be saved must be determined. In this regard, access to the respective environmental variable is needed. The user, in this case, wants to save the document on the desktop of the PC. Therefore, the environmental variable "UserProfile" has to be addressed by using the command *Value().* Now, the access to this part of the computer is granted. In the next command line 3, the path as well as the name of the document is specified. As already mentioned, the document is to be saved on the user's desktop. Hence, the path contains now-available "UserProfile", "Desktop" and the name of the document "Nutshell_Example". This determined path is accessed via the variable "FileName" which is eventually used to save the document with the command *~SaveAs()* in line 12 of the figure above. The saved document is now available on the desktop as it is depicted in Executed Code 4. Last but not least, the document closes itself after the saving process through the command *~Quit*.



Executed Code 4: Nutshell Example 3

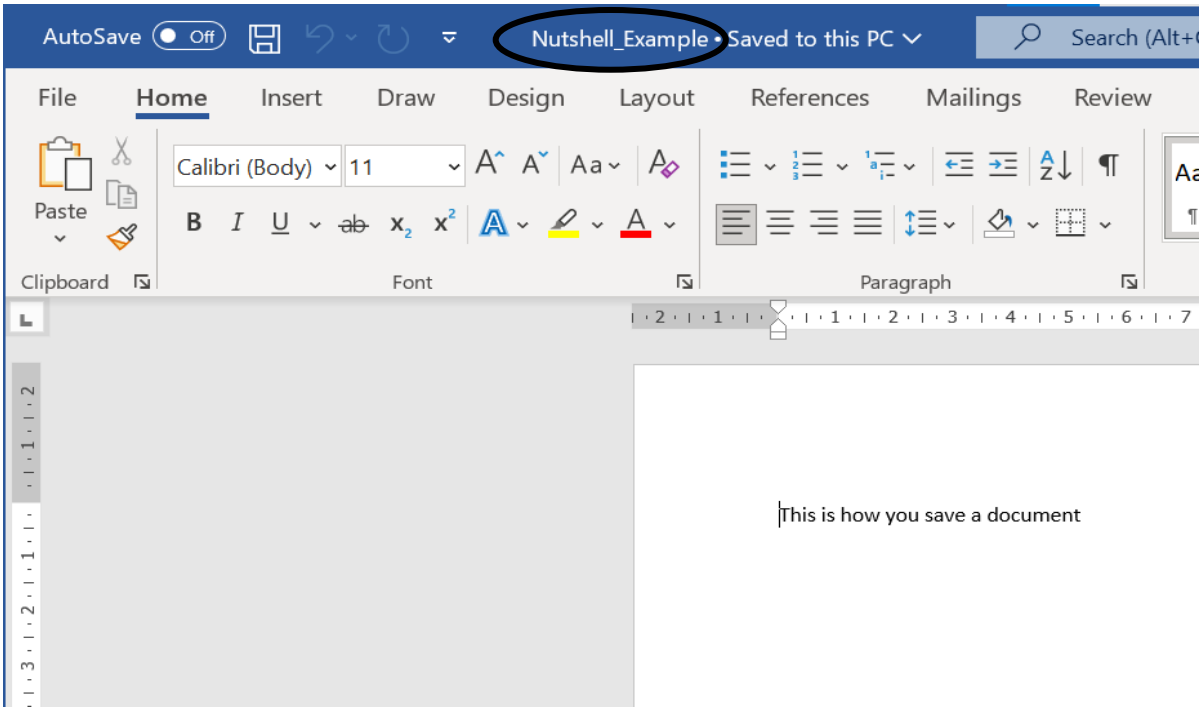
3.1.4 How to Reopen a Document

Next to saving a created document, the user might wish not only to open a completely new document but also an already created document which has been saved on the computer for further usage.

```
1     /*open MS Word*/
2     word=.oleobject~new("Word.Application")
3     word~visible=.true
4
5     /*reopen a document*/
6     Dir= Value("UserProfile",,ENVIRONMENT)
7     FileName= Dir || "\Desktop\" || "Nutshell_Example.docx"
8     OldDocument=word~Documents
9     OldDocument~Open(FileName)
10
```

Code 5: Nutshell Example 4 - How to Reopen a Document

In order to reopen a document, two major steps have to be gone through. First, MS Word needs to be opened by using the already familiar command *.oleobject* among other things (see line 2 and 3 in Code 5). Second, ooRexx needs the path of the desired document to be able to open it. In this example, the document saved in chapter “3.1.3 How to Save a Document” is to be reopened. Therefore, access to the UserProfile must be granted (line 6) and the complete path including the file name needs to be stated (line 7). However, it is to be noted that the file name must contain the respective ending of the file format. In this case, since the desired document is a Word document, the corresponding type is “.docx”. If the user is uncertain about the format, it is discovered easily by inspecting the properties of the document. After determining the path, ooRexx informs MS Word about a document to be opened. In this regard, an already known command is used in a slightly modified way: *word~Documents*. By omitting the *~add* command, ooRexx specifies to not open a new document but an existing one. Finally, the document is to be reopened by using *~open()* in combination with the path identified before. Executed Code 5 shows the result of this programming code – a reopened document.



Executed Code 5: Nutshell Example 4

3.1.5 How to Get into Print Preview

Last but not least, the user might want to print a created document after saving it. But before starting the printing process, the document must be checked if it is adequately formatted. The best way to perform it, is to activate the print preview in MS Word. This nutshell example intends to demonstrate this process.

```
7     NewDocument=word~Documents~add
8
9     ActiveDocument = Word~ActiveDocument
10    ActiveDocument~PrintPreview
11    ActiveWindow = Word~ActiveWindow
12    ActiveWindow~ActivePane~View~Zoom~Percentage = "100"
13    Call Syssleep 2
14    ActiveDocument~ClosePrintPreview
```

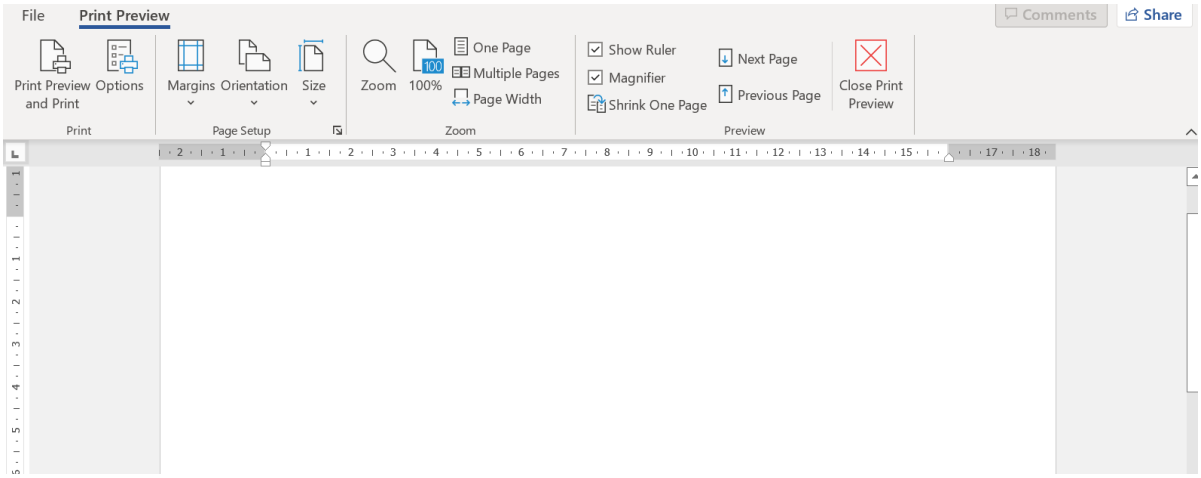
Code 6: Nutshell Example 5 - How to Get into Print Preview

In the first step, a document needs to be opened. Either the user can create a new one, as displayed in Code 6, or reopen an existing one. The instruction of the latter one is explained in chapter “3.1.4 How to Reopen a Document”. Afterwards, an active document is created to activate the Print Preview by using the command ~ActiveDocument (line 10 in Code 6). In order to understand the significance of an active document, Microsoft provides an explanation:

[…] active documents have complete control over their pages, and the full interface of the application (with all its underlying commands and tools) is available to the user to edit them (ghogen & et al, 2021).

In the continuous step the window is activated in the same way as the active document is done. The active window is then used to get access to the zoom option in the print preview of MS Word. Using the command *ActiveWindow~ActivePane~View~Zoom~Percentage=…* the degree of zoom can be specified. In this nutshell example, the zoom is to be at 100% for 2 seconds for

demonstration purposes. Latter is commanded by applying *call syssleep*. The result is depicted in Executed Code 6. Finally, the print preview is closed by using the command in line 14.



Executed Code 6: Nutshell Example 5

3.2 Special Functionalities

After getting to know the basic usage of ooRexx in MS Word, the nutshell examples will turn more specific. The following subchapters enable the advanced application of ooRexx to control MS Word. Thereby ways to create and edit tables, to insert local images and texts from webpages, as well as encode these texts are stated and explained to the reader.

3.2.1 How to Create a Table

This nutshell example is about how to insert a table in a Word document and enter text into the respective cells. Three different ways to achieve the latter are presented to the user. In this example, a table containing information about the number of credits gained each semester per year is created for visual illustration.

```
1    word=.oleobject~new("Word.Application")
2    word~visible=.true
3    NewDocument=Word~Documents~Add
4
5    SelectionObj=word~Selection
6    /*get an active document object */
7    ActiveDoc= word~ActiveDocument
8    /*create a table*/
9    ActiveDoc~Tables~Add(SelectionObj~Range,4,3)
10
```

Code 7: Nutshell Example 5a - How to Create a Table

In the first step, a new document is created in the already familiar way presented in chapter “3.1.1 How to Open MS Word and Create a Text” above. Continuing with line 5 of Code 7, access to the document is enabled by creating a selection object. After

an active document is generated and assigned to the variable “ActiveDoc”, a table is to be inserted. Thereby the commands *~Tables~Add(SelectionObj~Range,4,3)* are referred to the just created variable. The numbers in the parentheses determine the number of rows and columns. In this example the table will consist of 4 rows and 3 columns.

Up to now, the programming code inserts a completely empty table to the document without consisting of any borders or shading. When entering a text, the user can choose between 3 options. Each of those will be explained in the following.

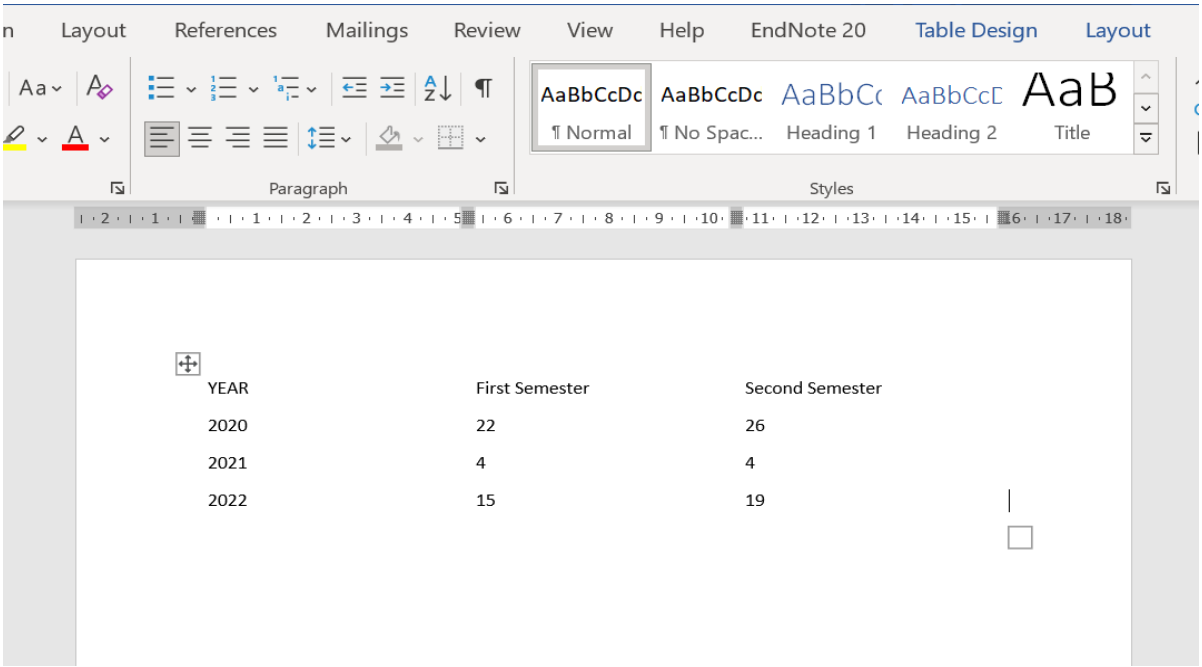
```
11   /* fill in table*/
12   /* first option*/
13   SelectionObj~Tables(1)
14   SelectionObj~TypeText("Year")
15   SelectionObj~MoveRight
16   SelectionObj~TypeText("First Semester")
17   SelectionObj~MoveRight
18   SelectionObj~TypeText("Second Semester")
19
20   SelectionObj~MoveRight
21
22   /*second option*/
23   do i=2020 to 2022
24     SelectionObj~TypeText(i)
25     SelectionObj~MoveRight
26     do 2
27       SelectionObj~TypeText(random(30))
28       SelectionObj~MoveRight
29     end
30   end
31
32   /* third option*/
33   ActiveTable=ActiveDoc~Tables(1)
34   ActiveTable~Cell(1,1)~Range~Text="YEAR"
35
```

Code 8: Nutshell Example 5b - How to Create a Table

In order to start with the first one, the right table must be selected by the command *~Tables().* Thereby the figure in parentheses determines which table is to be chosen. Now to reach the preferred cell, the command *SelectionObj~MoveRight* is used to pass one cell after the other from the left to the right side . When the desired cell is selected, text is entered by ~TypeText("…"). In the example given, line 14 to 18 of Code 8 enter text into the first row of the table.

A second option includes a loop. In case of using consecutive numbers, they can be easily entered with the command *do… to …* . The programming code in Code 8 consists of a double loop to fill out all remaining cells in one turn. In the first loop the year is entered. After one date, random numbers are entered in the next two cells by using a second loop (line 25 to 28) and so on.

Last but not least, text can be entered specifically by determining the desired cell. To do this, the table needs to be activated by commanding *ActiveDoc~Tables()*. At this point, any cell can be selected, and text entered. In the example given, the text in the cell of the first row and column is to be changed to capitals using *~Cell(1,1)~Range~Text="YEAR"*. Executed Code 7 presents the result of the programming code of Code 7 and Code 8.



Executed Code 7: Nutshell Example 5

3.2.2 How to Edit a Table

The topic of this chapter describes how to edit a table. Since the last nutshell example just enables to create a basic table without any borders or shading, this one is about adjusting the table to the user's preferences. However, not only will the table itself be edited but also the respective text in the cells. This example is a continuation of the former nutshell example (Code 7 and Code 8) in order to be able to distinguish the differences in a better way.

```
35   /*edit borders*/
36   ActiveTable=ActiveDoc~Tables(1)
37   ActiveTable~Borders~InsideLineStyle=.true
38   ActiveTable~Borders~OutsideLineStyle=.true
39
40   /*change background color*/
41   EditTable=SelectionObj~Tables(1)
42   EditTable~Shading~BackgroundPatternColorIndex="10"
43
44   /*change background color of cell*/
45   do i=1 to 4
46       EditTable~Cell(i,1)
            ~Shading~BackgroundPatternColorIndex="8"
47   end
48
49   /*adapt space between cells*/
50   ActiveTable~Spacing="5"
51
```

Code 9: Nutshell Example 6a - How to Edit a Table

Starting with line 36 to 38 of Code 9, the borders are to be added to the table. Since access to the table is needed, named must be activated in the first place. Afterwards, the user can adjust the borders at will using the command *~Borders~InsideLineStyle=.true* for the inner borders and

~Borders~OutsideLineStyle=.true for the outer borders. In this example, the user activated both types for better representation.

With reference to the coloring of the table, advantage of the SelectionObj is taken. In terms of changing the background color, *~Tables()~Shading* is added to determine the next step. Thereby the color can be chosen with the command *~BackgroundPatternColorIndex="…"* . But not only the background of the whole table can be changed but also the one of each cell individually. To achieve that, the correct cell has to be specified with *~Cell(row, column)*. Thereafter, the same command as used for the background color of the table is used. In the example given, the background of the table is defined as teal (line 42) and the one of the cells in the first column white (line 45 - 47)

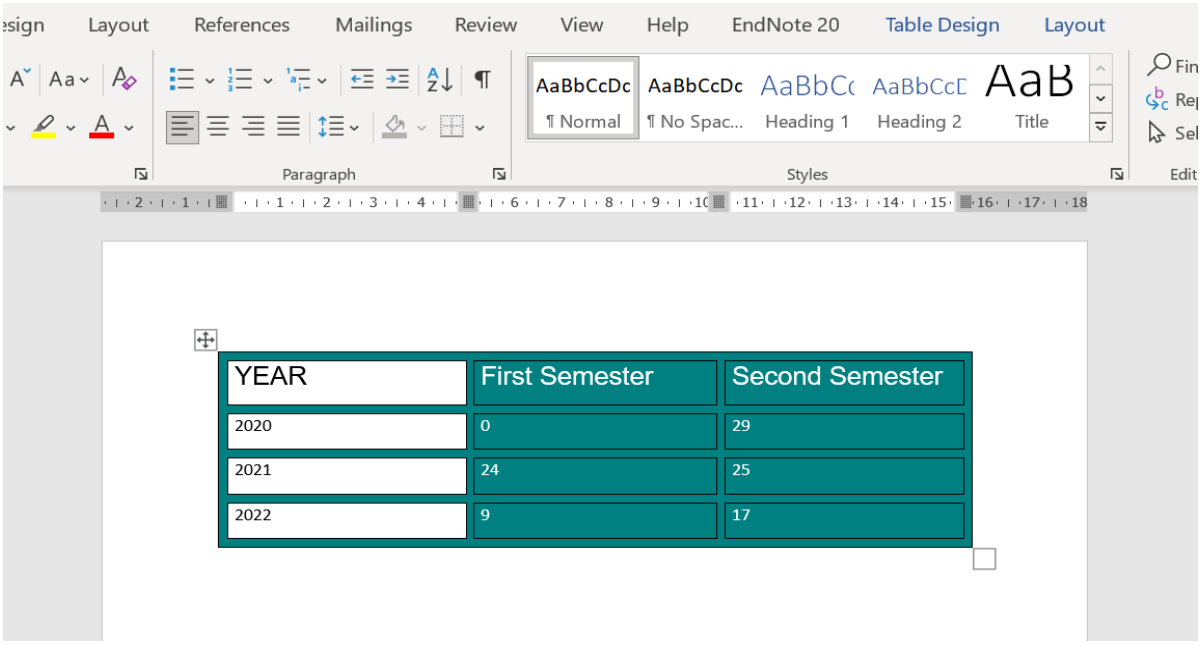
Continuing editing the cells, the space between each one can be adjusted using *ActiveTable~Spacing="…".* The number in quotation marks specifies the size of the distance - the greater the figure, the greater the space. As it is shown in Executed Code 8 at the end of this chapter, a gap parts two cells.

```
52   /*change font size and style*/
53   do i=1 to 3
54   ActiveTable~Cell(1,i)~Range~Font~Size="16"
55   ActiveTable~Cell(1,i)~Range~Font~Name="Arial Bold"
56   end
57
58   /*change font color*/
59   do i=2 to 3
60       do n=1 to 4
61           ActiveTable~Cell(n,i)~Range~Font~Colorindex="8"
62       end
63   end
```

Code 10: Nutshell Example 6b - How to Edit a Table

Up to this point, the user is able to edit the table at will. However, the text is still presented in standard format. To change the style of a text, the variable ActiveTable is needed as well as the command ~Cell(row, column) to define the desired cell and ~Range. Then, any command referring to the font style used in chapter “3.1.1 How to

Open MS Word and Create a Text" can be entered. For instance, in line 54 of Code 10 the size of the text is changed by adding ~Font~Size="…". In the same way the font type as well as the color are edited in this nutshell example (line 55 and 61).



Executed Code 8: Nutshell Example 6

3.2.3 How to Insert an Image

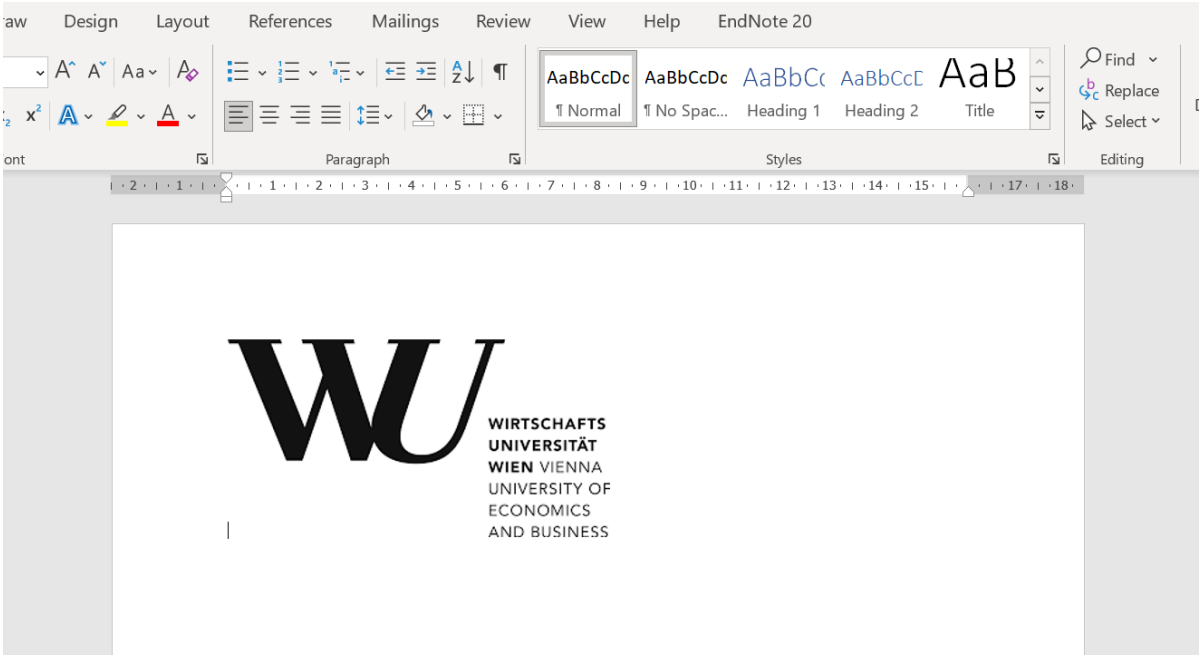
In many cases, images are needed to be positioned in a document to enable a better understanding of the reader or just put emphasize on a statement. The relevant code

```
1     word= .oleobject~new("Word.Application")
2     word~visible=.true
3
4     NewDocument= word~Documents~add
5
6
7     Dir= Value("UserProfile",,ENVIRONMENT)
8     FileName= Dir || "\Pictures\" || "WU_Logo.png"
9
10    /*Add picture*/
11    ActiveDoc= word~ActiveDocument
12    ActiveDoc~InlineShapes~AddPicture(FileName)
13
```

Code 11: Nutshell Example 7a - How to Insert an Image

to open saved pictures in a Word document is stated in the following chapter. Additionally, to round off this topic about pictures in MS Word, the way to crop them adequately is presented.

In order to add a picture, a new document must be created. Afterwards, access to the respective environment is to be enabled by using the same commands as in chapter “3.1.4 How to Reopen a Document” when a saved document is reopened. In this example, the relevant image is saved in the folder “Pictures” named “WU_Logo”. The respective image format must be stated as well, so, ooRexx knows which file type is to be opened (see line 8 in Code 11). In the next step, an active document is created and referred to with the commands ~InlineShape~AddPicture(…). The former created path of the saved picture is inserted in the empty brackets. Now, the program gets the information where to receive the image from and enters latter in the created document (Executed Code 9).

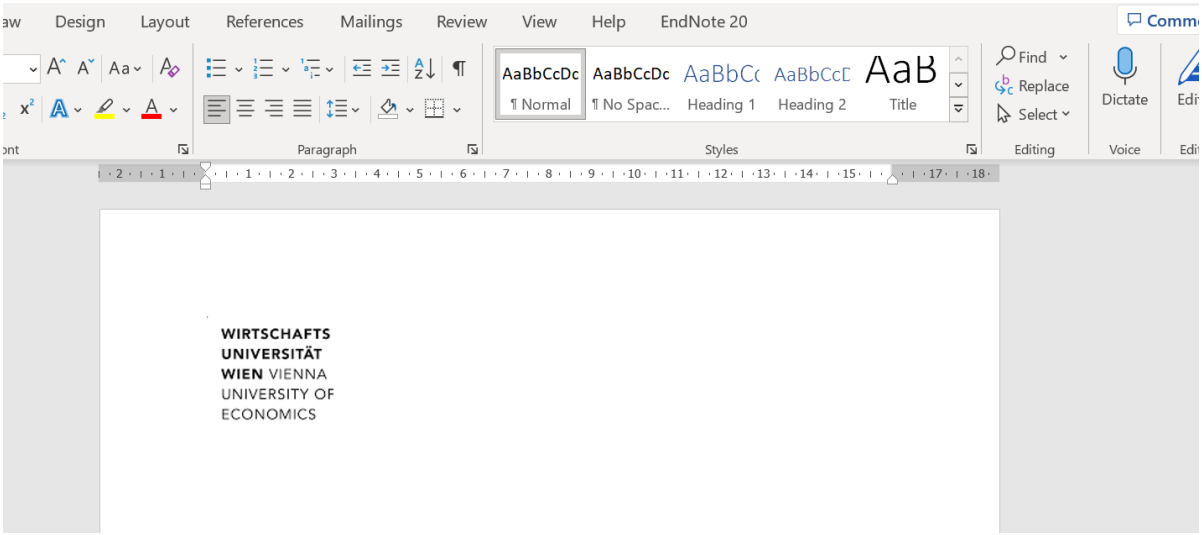


Executed Code 9: Nutshell Example 7a

```
14
15    /*Crop the picture*/
16    ActivePic=ActiveDoc~InlineShapes(1)~PictureFormat
17    ActivePic~CropBottom=10
18    ActivePic~CropLeft=150
19    ActivePic~CropRight=0.5
20    ActivePic~CropTop=40
```

Code 12: Nutshell Example 7b - How to Insert an Image

Sometimes, the inserted picture needs some adjustments made by the user. In order to do so, access to the format of the picture is enabled by using the command ~Inlineshapes(1)~PictureFormat on the activated document in line 16 of Code 12. In this nutshell example, the command is saved as the variable “ActivePic”. Subsequently, the image can be cropped at will applying ~CropBottom, ~CropLeft, ~CropRight and ~CropTop to the code. Thereby the measurements are in relation to the original size of the picture and in points. Executed Code 10 presents the picture after being cropped to only the written part of the logo.



Executed Code 10: Nutshell Example 7b

3.2.4 How to Insert a Text from a Webpage

After this chapter, the reader will be able to copy an arbitrary text into a MS Word document. However, it has to be mentioned that only specific websites are useable for this process. Either the webpage grants open access to any user, or the respective API is known. Providing a short excurse, API is an acronym for “Application Programming Interface”. It acts as an intermediary and allows 2 different programs to interact with each other (N.G., 2022). To get the API from a website, there are several lists on the internet, stating websites with public APIs. An example for such a website is provided in the following: https://mixedanalytics.com/blog/list-actually-free-open-no-auth-needed-apis/ .

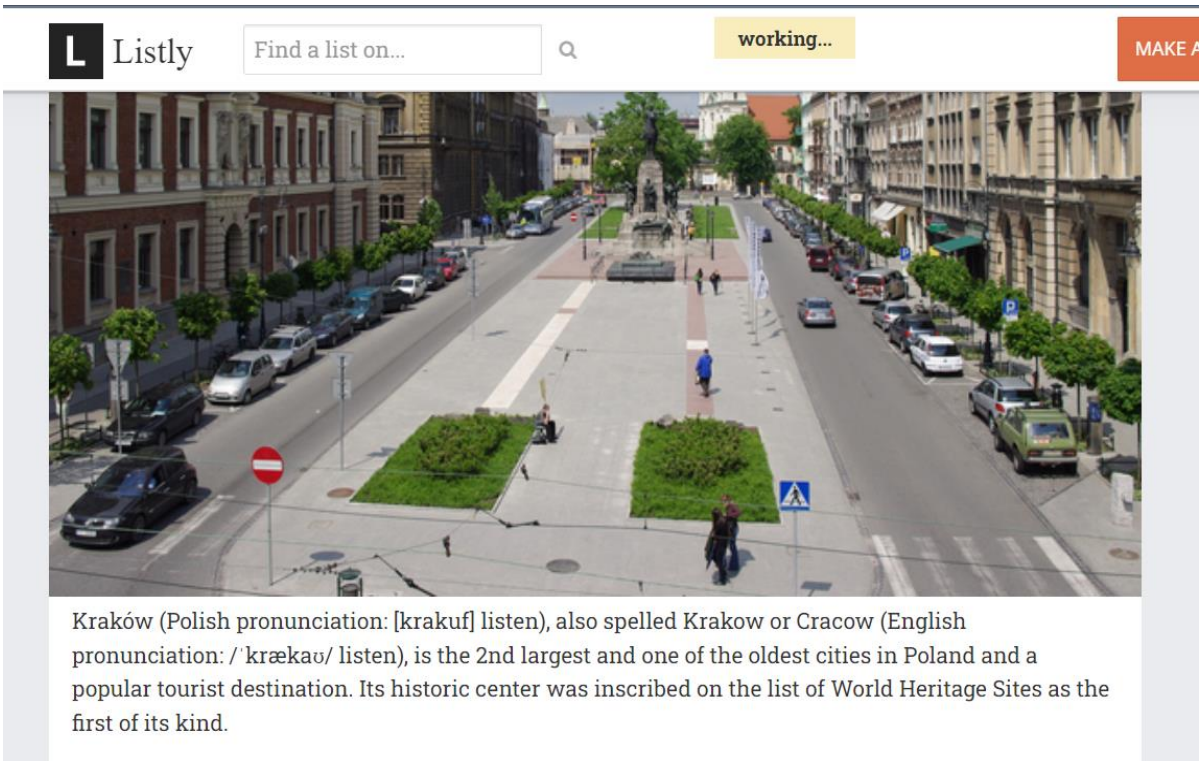


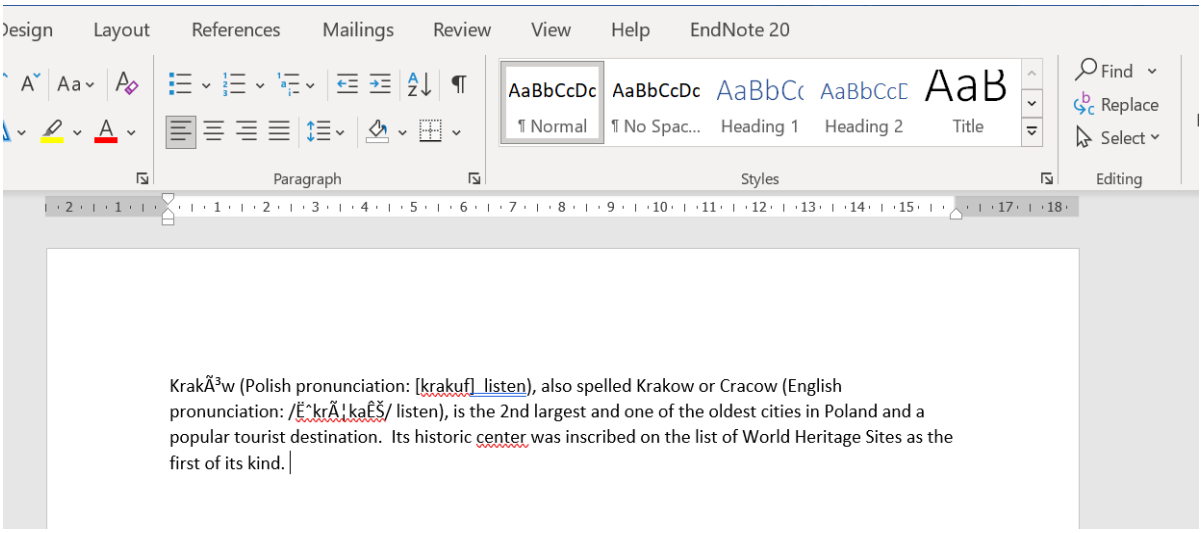
Figure 4: Nutshell Example 8 - Paragraph to Copy – URL: https://list.ly/list/m-best-european-cities-to-visit

First, a text on a website must be chosen. In this nutshell example, a short paragraph about the city Kraków from the website “Listly” was selected (see Figure 4).

```
1    /*using curl*/
2    url="https://list.ly/list/m-best-european-cities-to-
          visit/"
3    command= "curl" url
4    outArr=.array~new
5    errArr=.array~new
6    ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR
     USING (errArr)
7    html=outArr~makeString
8    /*get content*/
9    PARSE VAR html '<h2 class="ly-item-title"
      title="Krakow">' . '<div class="item_note wbreak
      ly-markdown"><p>' content '</p>'
10
11   /*open MS Word*/
12   word= .oleobject~new("Word.Application")
13   word~visible=.true
14   NewDocument=word~Documents~add
15   /*insert content in MS Word*/
16   SelectionObj=word~Selection
```

Code 13: Nutshell Example 8 - Inserting a Text from a Webpage

The respective link is to be saved and used in combination with the command *curl* in line 3 of Code 13. Curl is an acronym for “Client URL” and grants access to resources from the internet (Flatscher, Procedural and Object-oriented Programming 6 - Commands, 2022, p. 18).
In line 6 the webpage is addressed, and its standard-output and standard-error are redirected to the created arrays in the ooRexx program. Afterwards the output-array is converted into a plain string. The command *PARSE VAR* is used to receive the selected paragraph of the webpage. Due to inserting the respective html code, the position of the text, which is to be saved, is specified.
In order to display the paragraph in a Word document, the already known command to enter a text is used in combination with the variable of the saved text in line 17. As in Executed Code 11 presented, the desired paragraph is inserted in a document.



Executed Code 11: Nutshell Example 8

3.2.5 How to Encode an Inserted Text

In the previous nutshell example “3.2.43.2.43.2.4 How to Insert a Text from a Webpage”, the output of the code displays the selected paragraph of the website. However, as it is observable, special characters are not correctly inserted (see Executed Code 11). This is on account of the different codepages provided. To start with, ooRexx is able to work with UTF-8 (codepage 65001) used in different text sources (Flatscher, ooRexx and Unicode, 2010, p. 1). In contrast, MS Word only supports texts encoded in codepage 1252. Hence, MS Word might not completely process the entered data correctly. This nutshell example deals with this problem and presents a suitable solution.

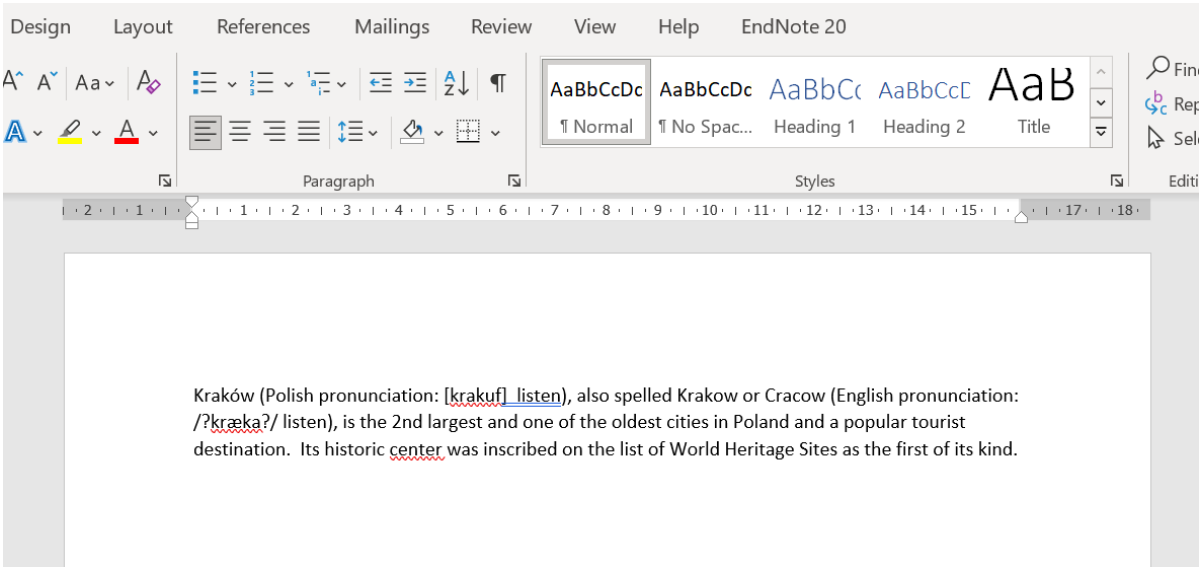
```
11   /*open MS Word*/
12   word= .oleobject~new("Word.Application")
13   word~visible=.true
14   NewDocument=word~Documents~add
15
16   say "textEncoding:" newDocument~textEncoding
17
18   /*insert content in MS Word*/
19   SelectionObj=word~Selection
20   newcontent=bsf.iconv(content, "utf-8","cp1252")
21   SelectionObj~TypeText(newcontent)
22
23   parse pull
24
25   ::requires "BSF.CLS"
```

Code 14: Programming Code - Nutshell Example 9

The programming code presented in Code 14 continues in line 11 provided in Code 13. First, the codepage supported by MS Word needs to be discovered by using the command ~textEncoding. In this example, the respective output corresponds with codepage 1252. In line 20, the text in UTF-8 is converted to the codepage 1252 to enable its correct presentation in a Word document.

In order to take advantage of the command bsf.iconv, the external function package BSF4ooRexx needs to be downloaded first. Afterwards the package BSF.CLS needs to be activated. As in chapter “2.2 BSF4ooRexx and BSF.CLS” stated, this package enables the programmer to use Java without knowing the respective codes in this programming language (Flatscher, Automatisierung mit ooRexx und BSF4ooRexx, 2012, p. 313).

As in Executed Code 12 displayed, the text is now correctly inserted except for the word in phonetic spelling. This can be explained by the amount of the character set of codepage 1252. Latter only provides 256 characters. Thereby it might be the case that a character is not included in this amount of provided characters. So, a similar presentation of the latter is intended by using available characters (Flatscher, E-Mail Exchange, 2022).



Executed Code 12: Nutshell Example 9

4 Summary and Conclusion

In the first part of this paper a short introduction of ooRexx is given. To sum up, the open-source program Open Object Rexx was published by the international, non-profit organization “RexxLa” in 2005. Thereby the program in question is an extension of the follow-up version of “Rexx” and was originally a proprietary program called oRexx. Rexx was developed in 1979 in order to create a language with the focus on the user. In other words, this programming language should be easy to learn. The commands are based on the English language and a small amount of rules and functions are just a few examples of how this was achieved.
ORexx, on the other hand, is object-oriented and used to command multiple environments.
With the development of BSF4ooRexx, a bridge to Linux and Windows was created with the help of “Bean Scripting Framework”. A special functionality is provided by the application of BSF.CLS. With the help of this extension, the user is able to use Java through ooRexx without knowing former language.

Continuing with the second part, the most important functionalities of ooRexx in combinational usage with MS Word are stated. Starting with the basic functionalities, 5 nutshell examples are demonstrated to the reader. The first two deal with the respective ways to open the program MS Word and insert an arbitrary text into a new document as well as create headings. Hereby,*.oleobject* and *Selection* are the key commands to make the codes work. In the following examples, the document is to be saved and reopened using commands to get access to the local computer. Last but not least, the print preview of a document should be opened by using *ActiveDocument.*
Afterwards, the special functionalities regarding ooRexx are presented in another 5 nutshell examples. Those are about creating and adjusting tables, inserting an image from the local computer as well as a text from a website. Finally, the inserted text is to be encoded in the last example. In those nutshell examples, commands like *ActiveDocument* and *Selection* dominate the majority of the examples given. Further

important commands include *curl* and *ADDRESS SYSTEM* to copy external texts and *bsf.iconv* to change the codepage.

In conclusion, ooRexx is a multi-functional program which is especially useful and easy to learn for beginners. The ability to combine it with other programs in order to use those, provides endless possibilities to the user. However, this programming language is unfortunately not as popular in the programming world as it should be regarding its various features. Nevertheless, it has great potential which might lead to extended usage if successfully disseminated.

5 Appendix

This chapter states not only the listing of the figures used in this paper but also the references regarding the citations.

5.1 Table of Figures

In the following subchapters, the used figures of codes, executed codes and other depicted images are listed with their names and the respective page.

5.1.1 Codes

Code 1: Nutshell Example 1 - How to Open MS Word and Create a Text 8

Code 2: Nutshell Example 2a – How to Create Headings 10

Code 3: Nutshell example 2b - How to Create Headings 12

Code 4: Nutshell Example 3 - How to Save a Document 14

Code 5: Nutshell Example 4 - How to Reopen a Document 15

Code 6: Nutshell Example 5 - How to Get into Print Preview 17

Code 7: Nutshell Example 5a - How to Create a Table 19

Code 8: Nutshell Example 5b - How to Create a Table 20

Code 9: Nutshell Example 6a - How to Edit a Table 22

Code 10: Nutshell Example 6b - How to Edit a Table 23

Code 11: Nutshell Example 7a - How to Insert an Image 24

Code 12: Nutshell Example 7b - How to Insert an Image 26

Code 13: Nutshell Example 8 - Inserting a Text from a Webpage 28

Code 14: Programming Code - Nutshell Example 9 30

5.1.2 Executed Codes

Executed Code 1: Nutshell Example 1 10

Executed Code 2: Nutshell Example 2a 11

Executed Code 3: Nutshell Example 2b 13

Executed Code 4: Nutshell Example 3 15

Executed Code 5: Nutshell Example 4 16

Executed Code 6: Nutshell Example 5 18

Executed Code 7: Nutshell Example 5 21

Executed Code 8: Nutshell Example 6 24

Executed Code 9: Nutshell Example 7a 25

Executed Code 10: Nutshell Example 7b 26

Executed Code 11: Nutshell Example 8 29

Executed Code 12: Nutshell Example 9 31

5.1.3 Figures

Figure 1: Logo of Rexx - URL: https://upload.wikimedia.org/wikipedia/en/f/f7/Rexx-img-lg.png..... 4

Figure 2: Logo of ooRex - URL: https://avatars.githubusercontent.com/u/11989843?s=280&v=4 4

Figure 3: Logo of BSF4ooRexx, URL: https://bach.wu.ac.at/d/media/cache/71/72/71725205a8f51921e894b891daae94b4.jpg .....**Error! Bookmark not defined.**

Figure 4: Nutshell Example 8 - Paragraph to Copy – URL: https://list.ly/list/m-best-european-cities-to-visit 27

5.2 References

Flatscher, R. G. (2009). "The 2009 Edition of BSF4Rexx". Vienna, Vienna, Austria. Retrieved November 23th, 2022, from https://wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_BSF4Rexx-20091031-article.pdf

Flatscher, R. G. (2010). ooRexx and Unicode. Retrieved November 9th, 2022, from https://wi.wu.ac.at/rgf/rexx/tmp/20110215-Unicode/_readme-20101115.pdf

Flatscher, R. G. (2012). Automatisierung mit ooRexx und BSF4ooRexx. In *Proceedings der GMDS 2012 / Informatik 2012* (pp. 1-12). Braunschweig.

Flatscher, R. G. (2017). *Automatisierungssprache Open Object Rexx 5.0 vor der Tür - Menschenfreund*. Retrieved November 20th, 2022, from Heise Magazin: https://www.heise.de/select/ix/2017/11/1509749512323037

Flatscher, R. G. (2022). E-Mail Exchange. (S. Oppermann, Interviewer) Retrieved November 8th, 2022

Flatscher, R. G. (2022). Procedural and Object-oriented Programming 1. Vienna, Austria. Retrieved November 24th, 2022

Flatscher, R. G. (2022). Procedural and Object-oriented Programming 6 - Commands. Vienna, Austria. Retrieved November 7th, 2022

Flatscher, R. G. (N.G.). An Introduction to Procedural and Object-oriented Programming (ooRexx). Vienna, Austria.

Förster, M. (2017). *Open Object Rexx 5.0: Mainframe-Klassiker für die Zukunft*. Retrieved November 20th, 2022, from heise online: https://www.heise.de/ix/meldung/Open-Object-Rexx-5-0-Mainframe-Klassiker-fuer-die-Zukunft-3888609.html

ghogen, & et al. (2021). *Active Document Containment*. Retrieved October 30th, 2022, from Microsoft: https://learn.microsoft.com/en-us/cpp/mfc/active-document-containment?view=msvc-170

N.G. (2022). *What is an API? (Application Programming Interface)*. Retrieved November 7th, 2022, from MuleSoft: https://www.mulesoft.com/resources/api/what-is-an-api

ProTech. (N.G.). *REXX Programming Language: What it is, Where it's Used, and Why You Should Care*. Retrieved December 10th, 2022, from ProTech: https://www.protechtraining.com/blog/post/rexx-programming#:~:text=REXX%20programming%20or%20Restructured%20Extended,is%20known%20for%20its%20efficiency.

Rexx Language Association. (2015). *About Open Object Rexx*. Retrieved November 29th, 2022, from ooRexx: https://www.oorexx.org/about.html

RexxLa. (N.G.). *The Rexx Language Association*. Retrieved December 10th, 2022, from Rexxla: https://www.rexxla.org/rexxla/about.srsp

tutorialspoint. (N.G.). *Rexx - Overview*. Retrieved December 10th, 2022, from tutorialspoint: https://www.tutorialspoint.com/rexx/rexx_overview.htm