Rexx

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Rexx (Restructured Extended Executor) is designed to be a scripting language. Its goal is to make scripting as easy, fast, reliable, and error-free as possible. Many programming languages are designed for compatibility with older languages, and are written for specific audiences or platforms. Rexx ignores extraneous objectives. It was designed from day one to be powerful, yet easy to use. It is also very helpful for developing small programs that perform various text file transformations.

This is an introductory tutorial that covers the basics of Rexx and how to deal with its various components and sub-components.

Audience

This tutorial has been prepared mainly for those professionals who are within the IT industry, working as specialists in the field of Scripting and Macro Languages. It is very useful for those professionals who work on Data Processing, text and also for generating reports.

This tutorial is intended to make you comfortable in getting started with Restructured Extended Executor (Rexx) and its various functions.

Prerequisites

It is an elementary tutorial and you can easily understand the concepts explained here with a basic knowledge of how a company or an organization deals with its scripting languages and programs. However, it will help if you have some prior exposure on programming languages, data processing, and generating reports.

Copyright and Disclaimer

Table of Contents

1. Rexx – Overview

Rexx (Restructured Extended Executor) is designed to be a scripting language. Its goal is to make scripting as easy, fast, reliable, and error-free as possible. Many programming languages are designed for compatibility with older languages, and are written for specific audiences or platforms. Rexx ignores extraneous objectives. It was designed from day one to be powerful, yet easy to use.

Rexx was designed and first implemented, in assembly language, as an 'own-time' project between 20th March 1979 and the middle of 1982 by Mike Cowlishaw of IBM, originally as a scripting programming language to replace the languages **EXEC and EXEC 2**. It was designed to be a **macro or scripting language** for any system. As such, Rexx is considered a precursor to Tcl and Python. Rexx was also intended by its creator to be a simplified and easier to learn version of the PL/I programming language.

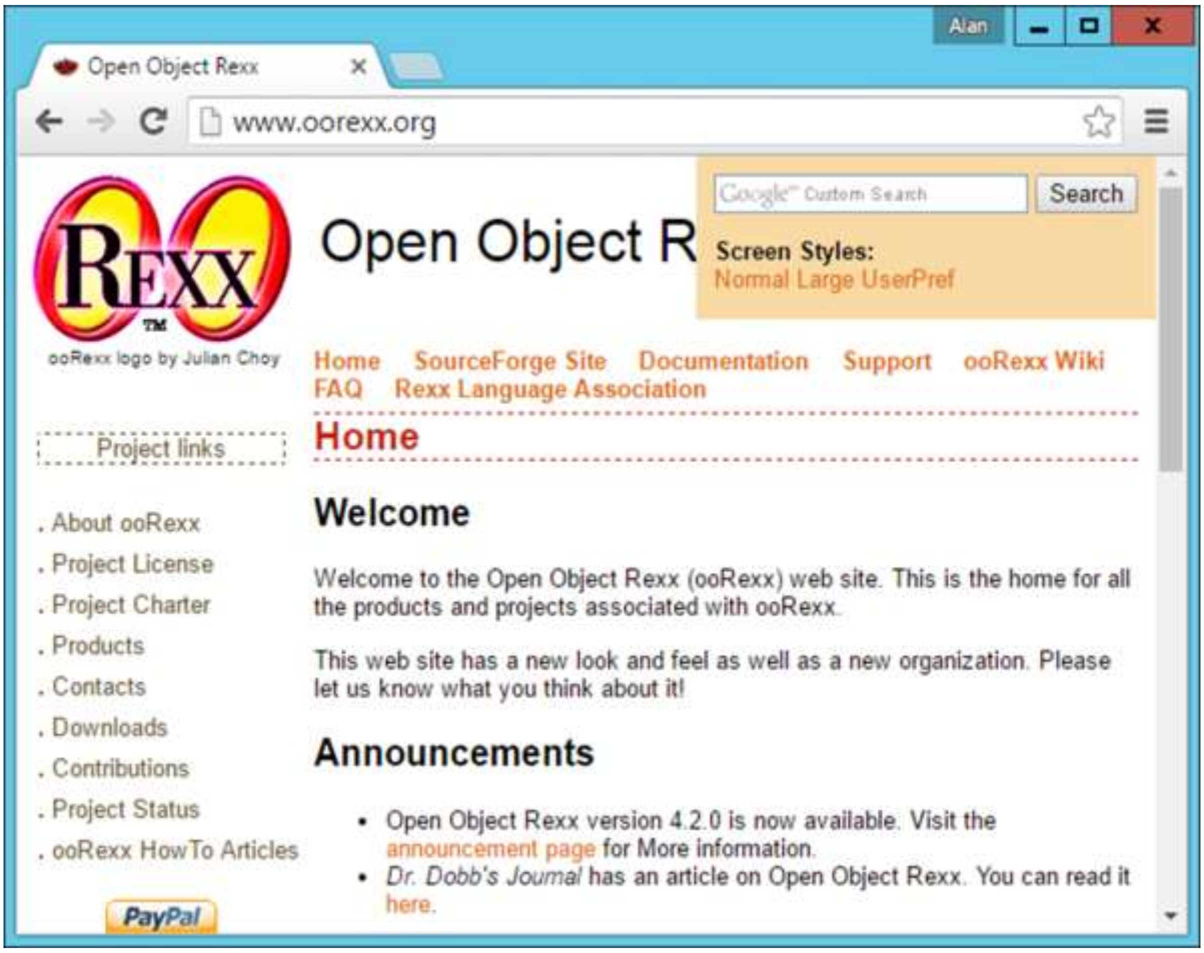
Features of Rexx

Rexx as a programming language has the following key features:

- Simple syntax
- The ability to route commands to multiple environments
- The ability to support functions, procedures and commands associated with a specific invoking environment.
- A built-in stack, with the ability to interoperate with the host stack if there is one.
- Small instruction set containing just two dozen instructions
- Freeform syntax
- Case-insensitive tokens, including variable names
- Character string basis
- Dynamic data typing, no declarations
- No reserved keywords, except in local context
- No include file facilities
- Arbitrary numerical precision
- Decimal arithmetic, floating-point
- A rich selection of built-in functions, especially string and word processing
- Automatic storage management

- Crash protection
- Content addressable data structures
- Associative arrays
- Straightforward access to system commands and facilities
- Simple error-handling, and built-in tracing and debugger
- Few artificial limitations
- Simplified I/O facilities

The official website for Rexx is http://www.oorexx.org/



Open Object R

Home

Welcome

Welcome to the Open Object Rexx (ooRexx) web site. This is the home for all the products and projects associated with ooRexx.

This web site has a new look and feel as well as a new organization. Please let us know what you think about it!

Announcements

- Open Object Rexx version 4.2.0 is now available. Visit the announcement page for More information.
- *Dr. Dobb's Journal* has an article on Open Object Rexx. You can read it here.

2. Rexx – Environment

Before you can start working on Rexx, you need to ensure that you have a fully functional version of Rexx running on your system. This chapter will explain the installation of Rexx and its subsequent configuration on a Windows machine to get started with Rexx.

Ensure the following System requirements are met before proceeding with the installation.

System Requirements

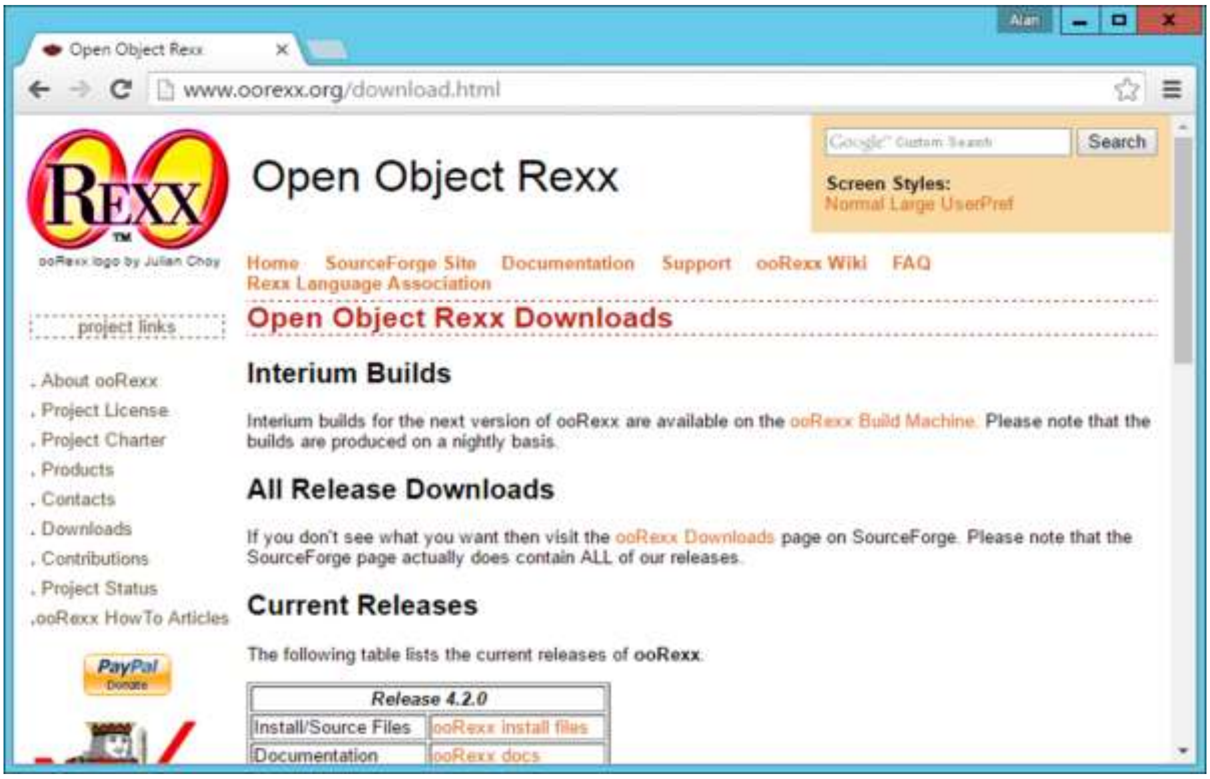
Memory	2 GB RAM (recommended)
Disk Space	No minimum requirement. Preferably to have enough storage to store the programs which will be created using Rexx.
Operating System Version	Rexx can be installed on Windows, Ubuntu/Debian, Mac OS X.

Downloading Rexx

To download Rexx, you should use the following URL –

http://www.oorexx.org/download.html

This page has a variety of downloads for various versions of Rexx as shown in the following screenshot.



Open Object Rexx

Screen Styles: Normal Large UserPref

Home SourceForge Site Documentation Support ooRexx Wiki FAQ Rexx Language Association

- About ooRexx
- Project License
- Project Charter
- Products
- Contacts
- Downloads
- Contributions
- Project Status
- ooRexx HowTo Articles

Open Object Rexx Downloads

Interium Builds

Interium builds for the next version of ooRexx are available on the ooRexx Build Machine. Please note that the builds are produced on a nightly basis.

All Release Downloads

If you don't see what you want then visit the ooRexx Downloads page on SourceForge. Please note that the SourceForge page actually does contain ALL of our releases.

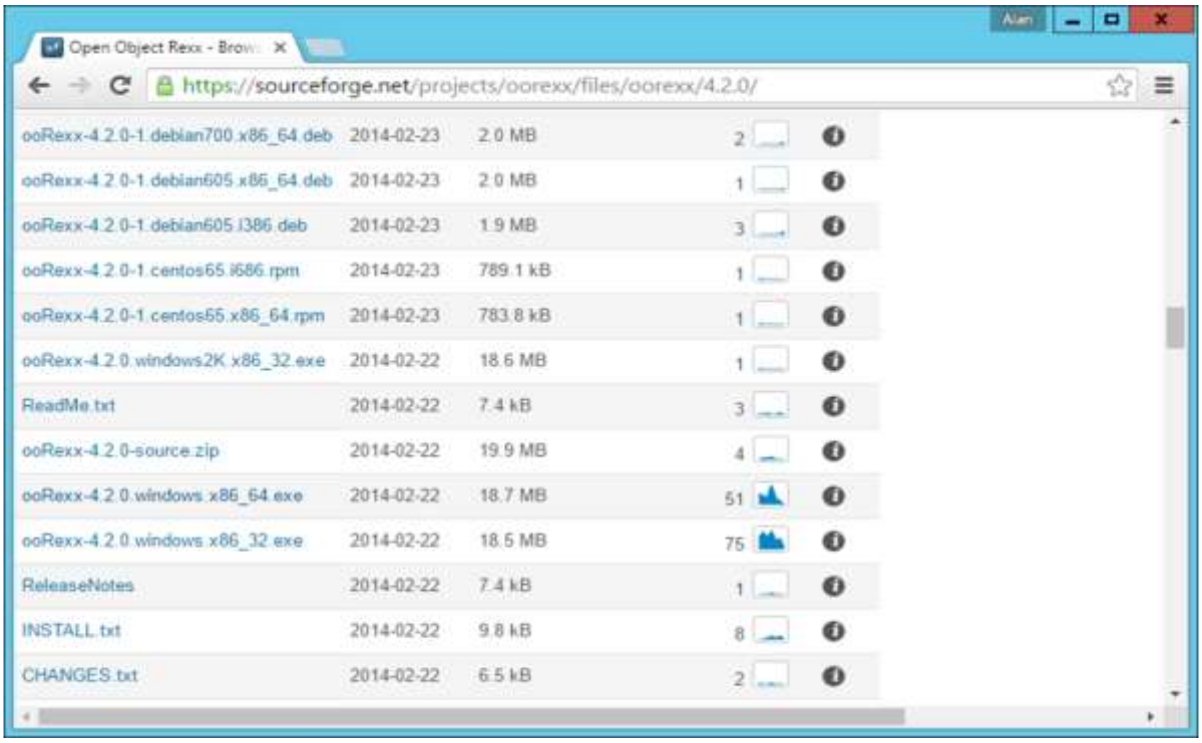
Current Releases

The following table lists the current releases of ooRexx.

Release 4.2.0	
Install/Source Files	ooRexx install files
Documentation	ooRexx docs

Click on the ‘ooRexx install files’ in the table with the header of Release 4.2.0.

After this, you will be re-directed to the following page.



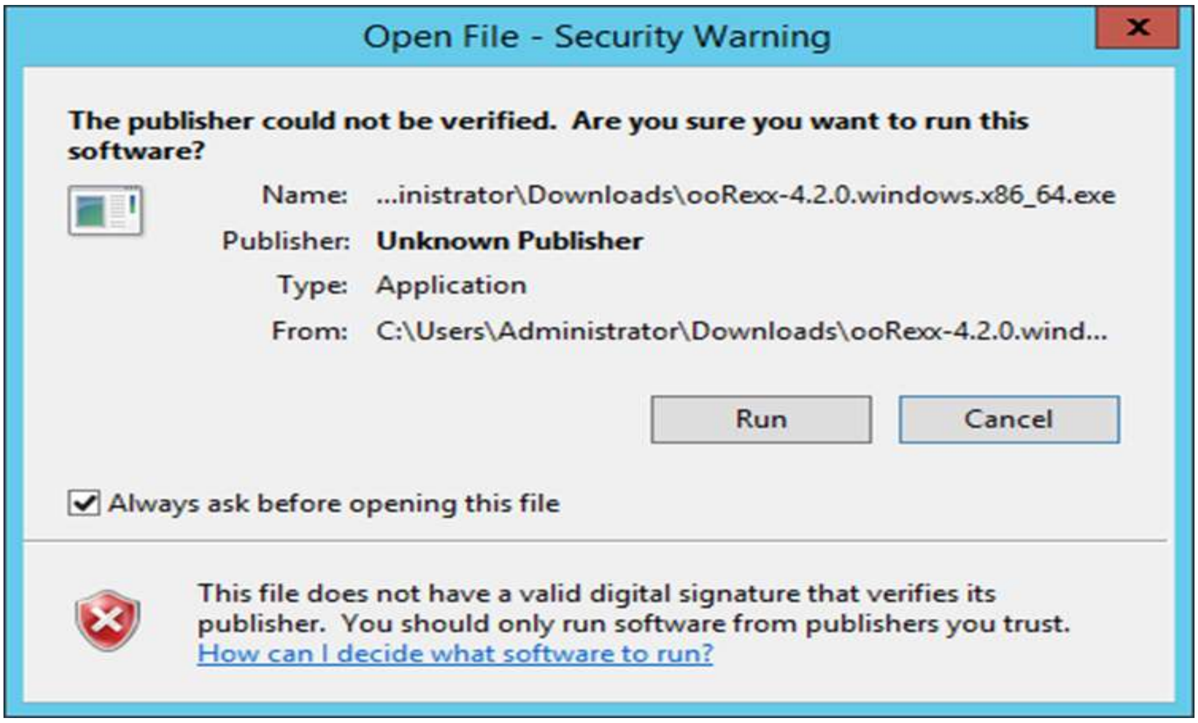
ooRexx-4.2.0-1.debian700.x86_64.deb	2014-02-23	2.0 MB	2
ooRexx-4.2.0-1.debian605.x86_64.deb	2014-02-23	2.0 MB	1
ooRexx-4.2.0-1.debian605.i386.deb	2014-02-23	1.9 MB	3
ooRexx-4.2.0-1.centos65.i686.rpm	2014-02-23	789.1 kB	1
ooRexx-4.2.0-1.centos65.x86_64.rpm	2014-02-23	783.8 kB	1
ooRexx-4.2.0.windows2K.x86_32.exe	2014-02-22	18.6 MB	1
ReadMe.txt	2014-02-22	7.4 kB	3
ooRexx-4.2.0-source.zip	2014-02-22	19.9 MB	4
ooRexx-4.2.0.windows.x86_64.exe	2014-02-22	18.7 MB	51
ooRexx-4.2.0.windows.x86_32.exe	2014-02-22	18.5 MB	75
ReleaseNotes	2014-02-22	7.4 kB	1
INSTALL.txt	2014-02-22	9.8 kB	8
CHANGES.txt	2014-02-22	6.5 kB	2

Click on the **ooRexx-4.2.0.windows.x86_64.exe** to download the **64-bit** version of the software. We will discuss regarding the installation of the software in the following chapter.

3. Rexx – Installation

The following steps will explain in detail how Rexx can be installed on a Windows system.

Step 1: Launch the Installer downloaded in the earlier section. After the installer starts, click on the Run button.



Open File - Security Warning

The publisher could not be verified. Are you sure you want to run this software?

Name: ...inistrator\Downloads\ooRexx-4.2.0.windows.x86_64.exe

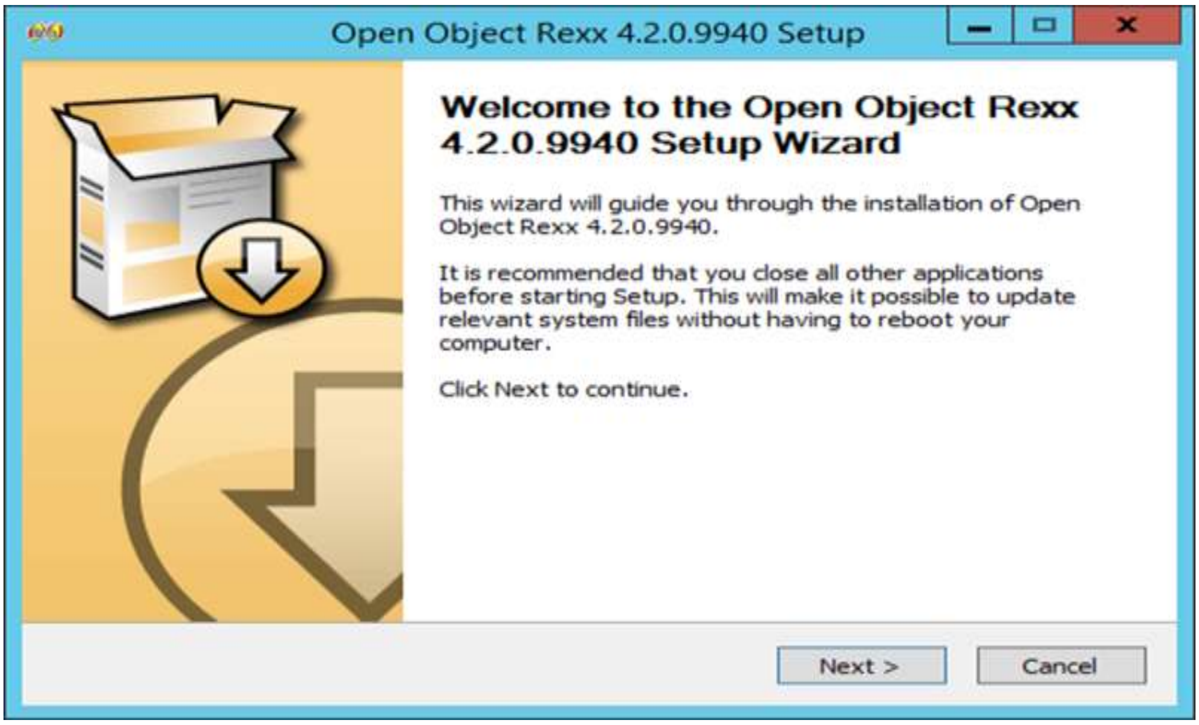
Publisher: Unknown Publisher

Type: Application

From: C:\Users\Administrator\Downloads\ooRexx-4.2.0.wind...

This file does not have a valid digital signature that verifies its publisher. You should only run software from publishers you trust.
How can I decide what software to run?

Step 2: Click the next button on the following screen to proceed with the installation.



Open Object Rexx 4.2.0.9940 Setup

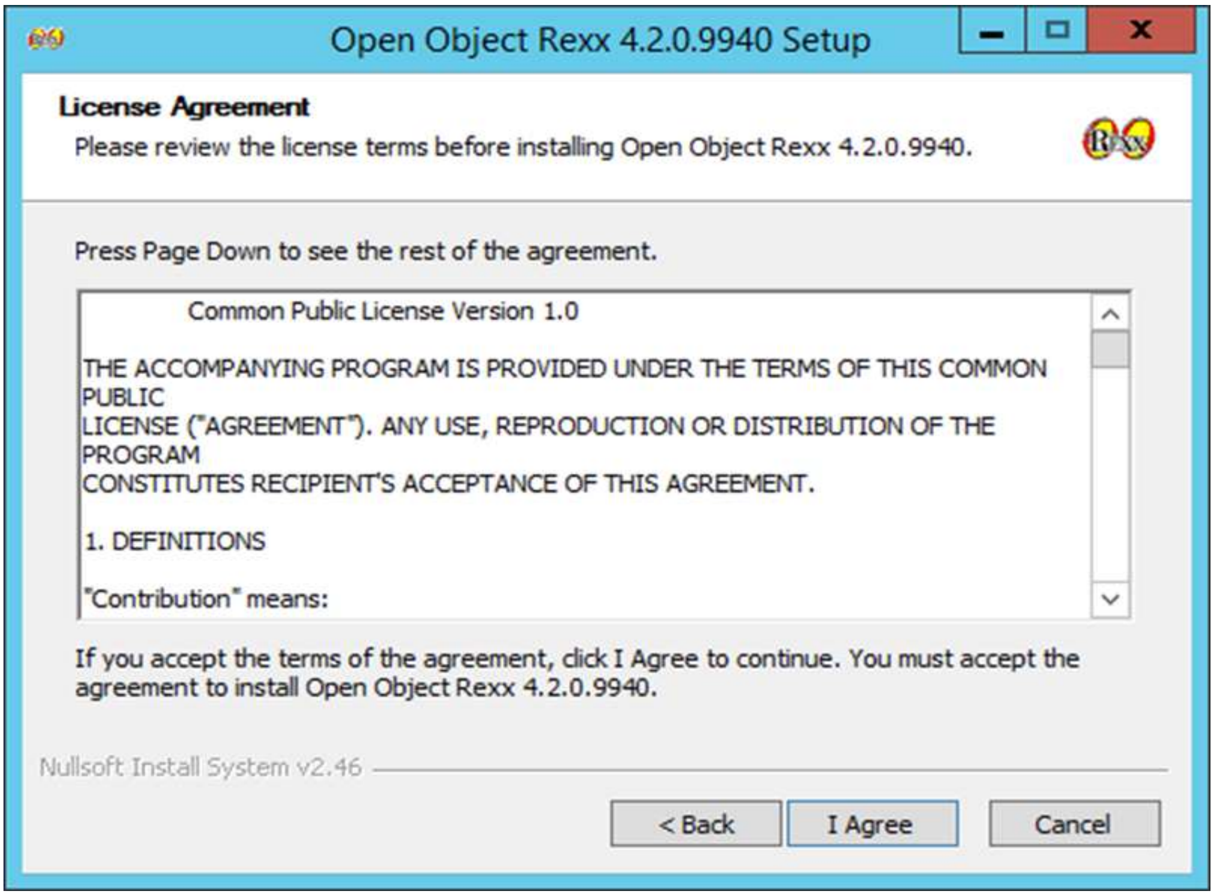
Welcome to the Open Object Rexx 4.2.0.9940 Setup Wizard

This wizard will guide you through the installation of Open Object Rexx 4.2.0.9940.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Step 3: Click on the **I Agree** button to proceed.



License Agreement

Please review the license terms before installing Open Object Rexx 4.2.0.9940.

Press Page Down to see the rest of the agreement.

Common Public License Version 1.0

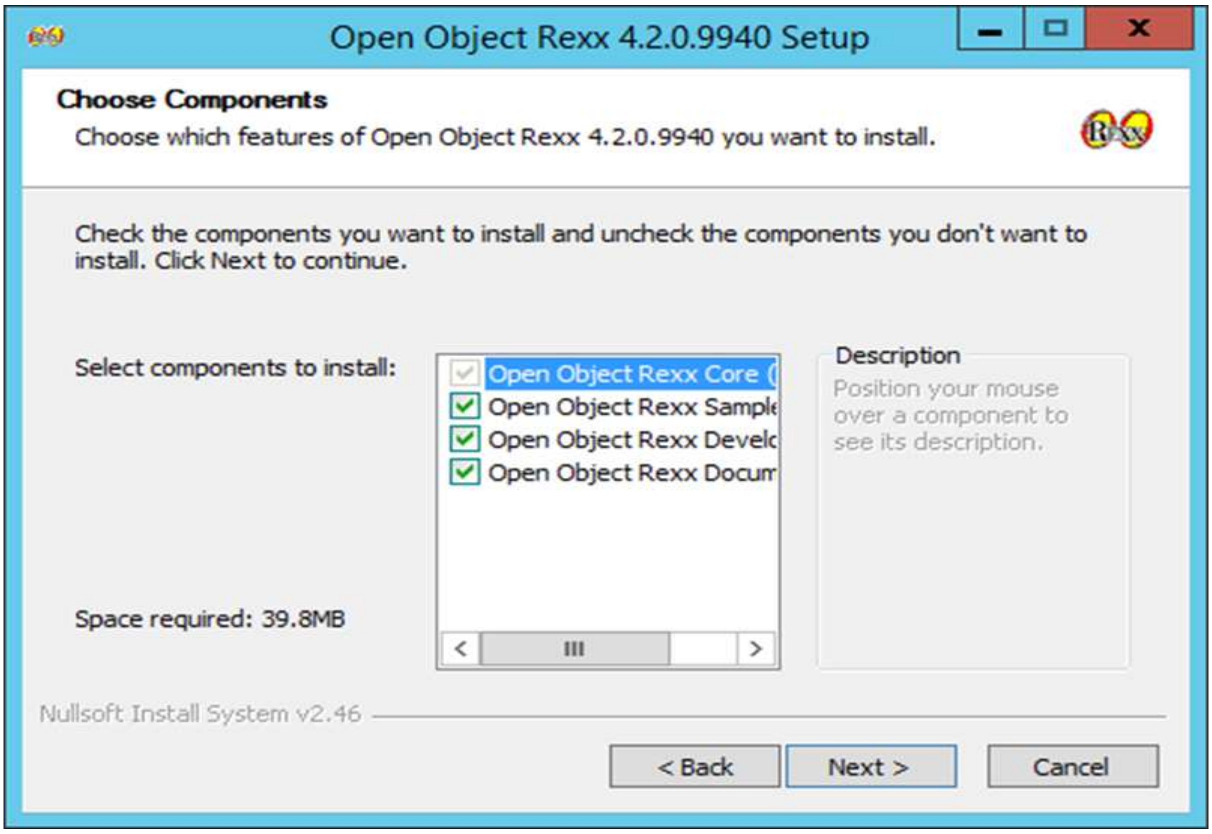
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC
LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM
CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Open Object Rexx 4.2.0.9940.

Step 4: Accept the **default components** and then click on the next button.



Choose Components

Choose which features of Open Object Rexx 4.2.0.9940 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

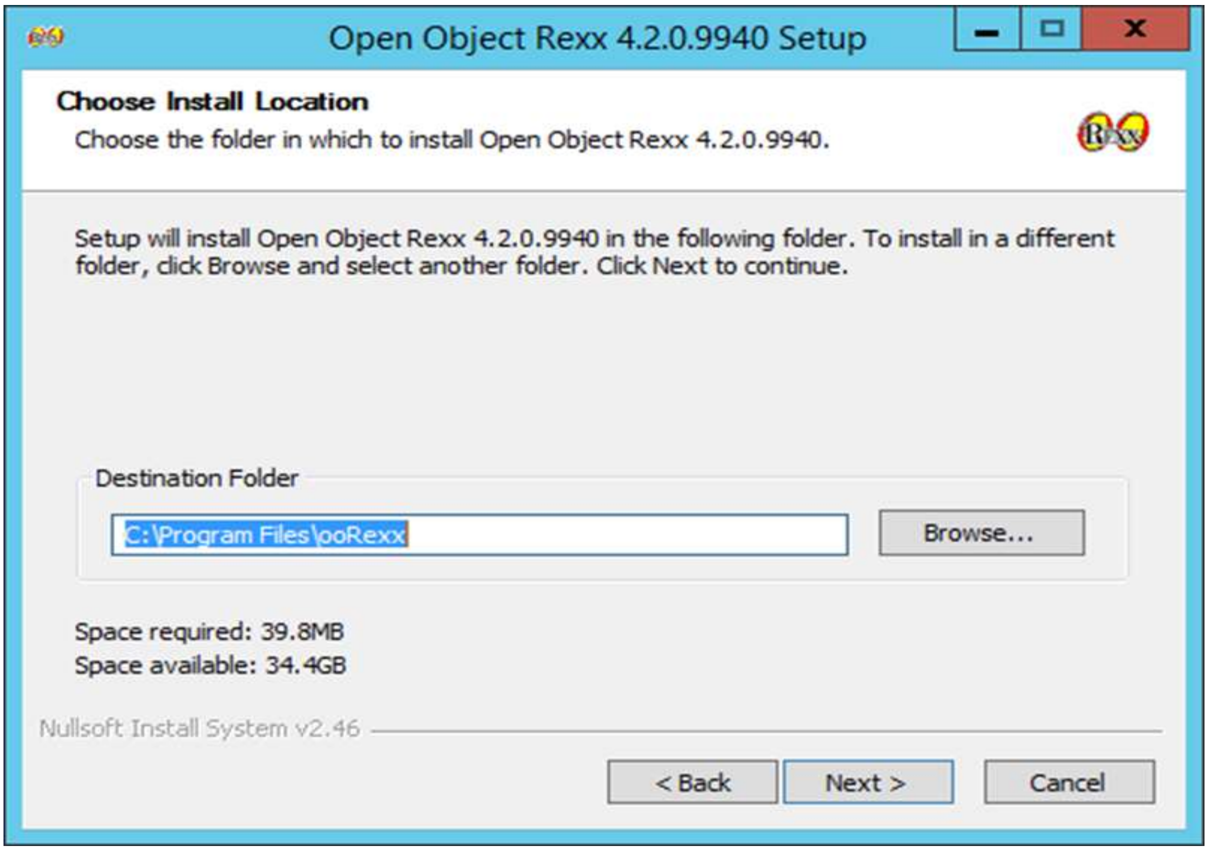
- [✓] Open Object Rexx Core (
- [✓] Open Object Rexx Sampl
- [✓] Open Object Rexx Develc
- [✓] Open Object Rexx Docum

Description

Position your mouse over a component to see its description.

Space required: 39.8MB

Step 5: Choose the installation location and click on the Next button.


Open Object Rexx 4.2.0.9940 Setup

Choose Install Location

Choose the folder in which to install Open Object Rexx 4.2.0.9940.

Setup will install Open Object Rexx 4.2.0.9940 in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

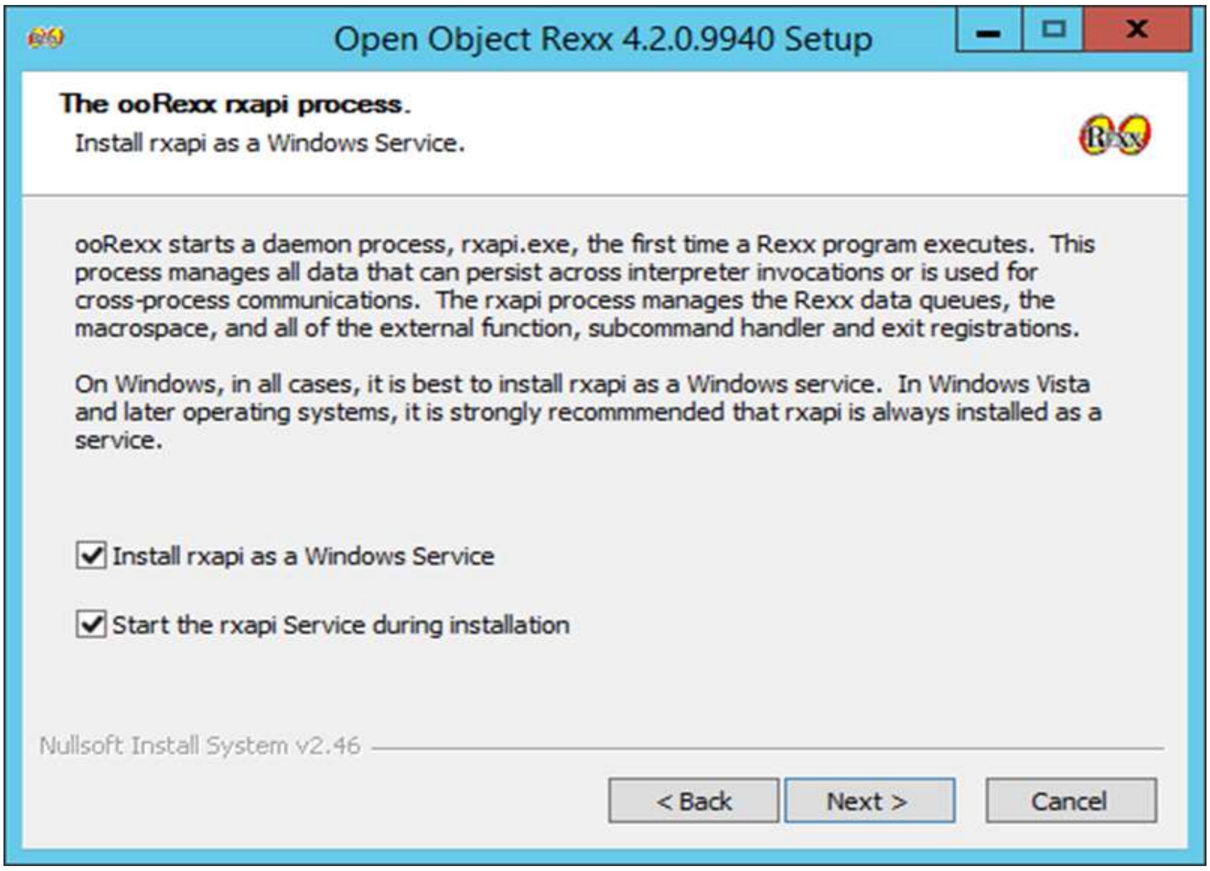
Destination Folder

C:\Program Files\ooRexx

Space required: 39.8MB
Space available: 34.4GB

Browse...
Nullsoft Install System v2.46
< Back
Next >
Cancel

Step 6: Accept the default processes which will be installed and click on the Next button.



The ooRexx rxapi process.

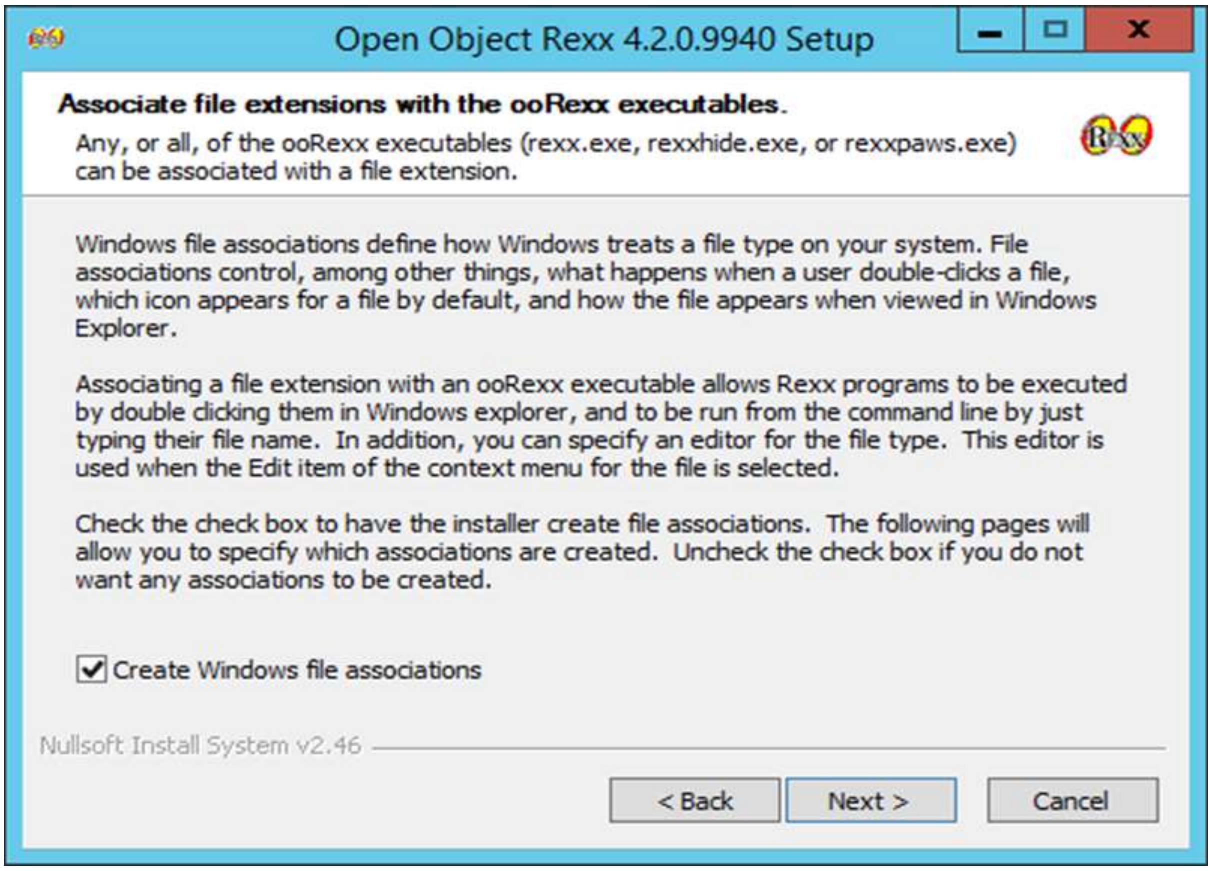
Install rxapi as a Windows Service.

ooRexx starts a daemon process, rxapi.exe, the first time a Rexx program executes. This process manages all data that can persist across interpreter invocations or is used for cross-process communications. The rxapi process manages the Rexx data queues, the macrospace, and all of the external function, subcommand handler and exit registrations.

On Windows, in all cases, it is best to install rxapi as a Windows service. In Windows Vista and later operating systems, it is strongly recommmended that rxapi is always installed as a service.

- [✓] Install rxapi as a Windows Service
- [✓] Start the rxapi Service during installation

Step 7: Choose the default file associations and click on the Next button.



Associate file extensions with the ooRexx executables.

Any, or all, of the ooRexx executables (rexx.exe, rexxhide.exe, or rexxpaws.exe) can be associated with a file extension.

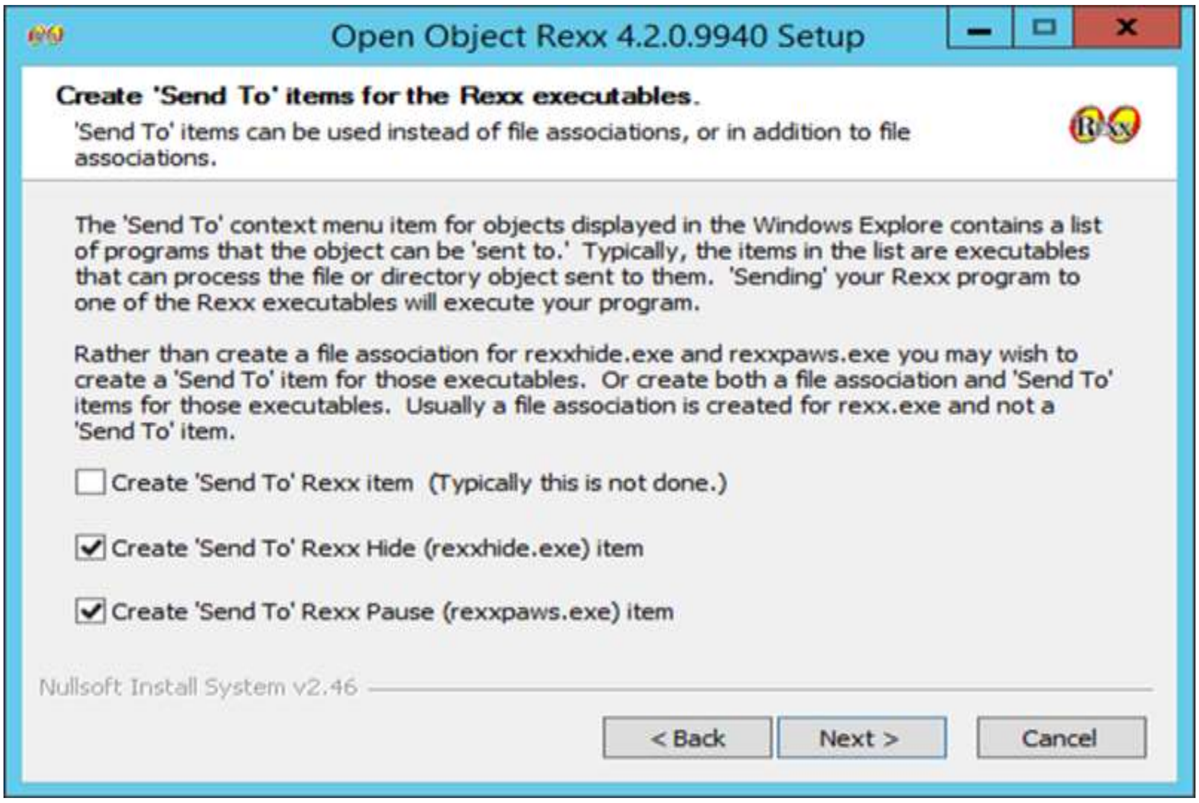
Windows file associations define how Windows treats a file type on your system. File associations control, among other things, what happens when a user double-clicks a file, which icon appears for a file by default, and how the file appears when viewed in Windows Explorer.

Associating a file extension with an ooRexx executable allows Rexx programs to be executed by double clicking them in Windows explorer, and to be run from the command line by just typing their file name. In addition, you can specify an editor for the file type. This editor is used when the Edit item of the context menu for the file is selected.

Check the check box to have the installer create file associations. The following pages will allow you to specify which associations are created. Uncheck the check box if you do not want any associations to be created.

☑ Create Windows file associations

Step 8: Click on the check boxes of send Rexx items to the executables and then click on the Next button as shown in the following screenshot.



Create 'Send To' items for the Rexx executables.

'Send To' items can be used instead of file associations, or in addition to file associations.

The 'Send To' context menu item for objects displayed in the Windows Explore contains a list of programs that the object can be 'sent to.' Typically, the items in the list are executables that can process the file or directory object sent to them. 'Sending' your Rexx program to one of the Rexx executables will execute your program.

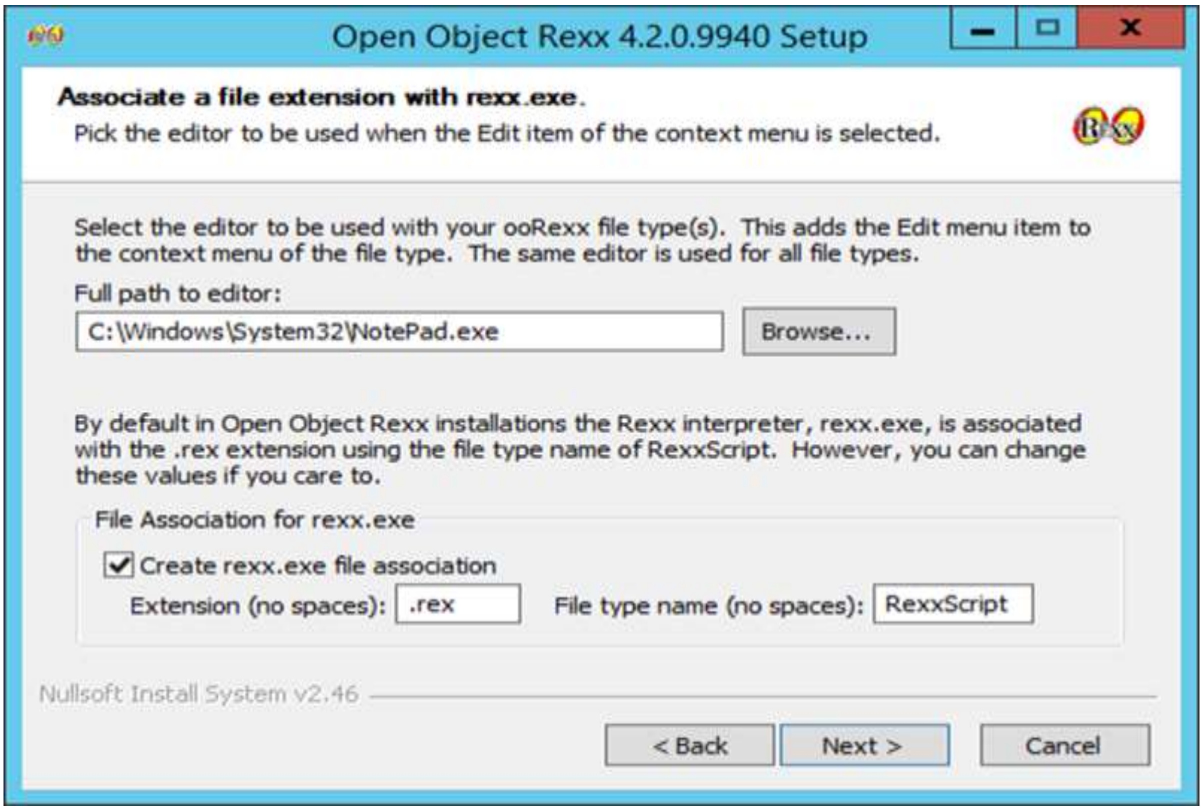
Rather than create a file association for rexxhide.exe and rexxpaws.exe you may wish to create a 'Send To' item for those executables. Or create both a file association and 'Send To' items for those executables. Usually a file association is created for rexx.exe and not a 'Send To' item.

☐ Create 'Send To' Rexx item (Typically this is not done.)

☑ Create 'Send To' Rexx Hide (rexxhide.exe) item

☑ Create 'Send To' Rexx Pause (rexxpaws.exe) item

Step 9: In the next screen, choose the editor for working with Rexx files. Keep the notepad as the default option. Also accept the default extension for each Rexx file.



Associate a file extension with rexx.exe.

Pick the editor to be used when the Edit item of the context menu is selected.

Select the editor to be used with your ooRexx file type(s). This adds the Edit menu item to the context menu of the file type. The same editor is used for all file types.

Full path to editor: C:\Windows\System32\NotePad.exe

By default in Open Object Rexx installations the Rexx interpreter, rexx.exe, is associated with the .rex extension using the file type name of RexxScript. However, you can change these values if you care to.

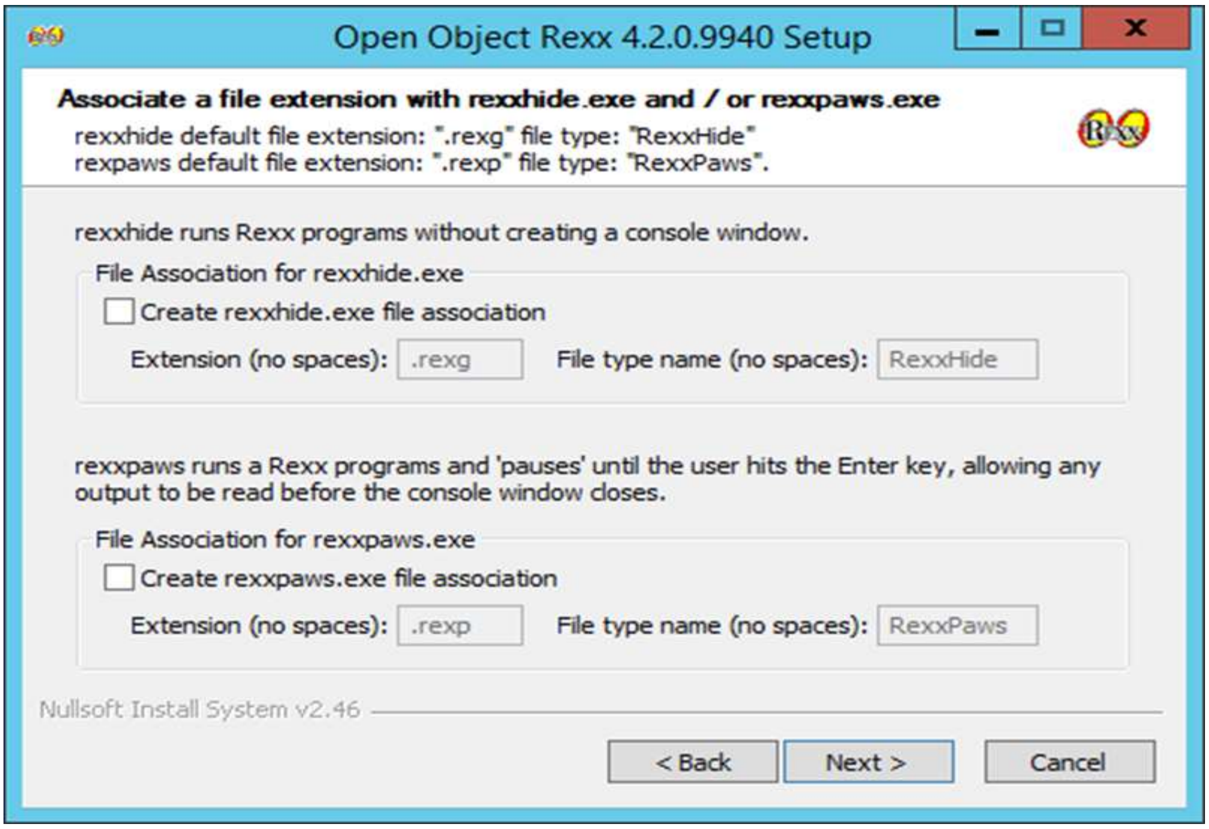
File Association for rexx.exe

[✓] Create rexx.exe file association

Extension (no spaces): .rex File type name (no spaces): RexxScript

Nullsoft Install System v2.46

Step 10: Accept the default settings on the following screen that comes up and click on the Next button to proceed further with the installation.



Associate a file extension with rexxhide.exe and / or rexxpaws.exe

rexxhide default file extension: ".rexg" file type: "RexxHide"
rexpaws default file extension: ".rexp" file type: "RexxPaws".

rexxhide runs Rexx programs without creating a console window.

File Association for rexxhide.exe

[ ] Create rexxhide.exe file association

Extension (no spaces): .rexg File type name (no spaces): RexxHide

rexxpaws runs a Rexx programs and 'pauses' until the user hits the Enter key, allowing any output to be read before the console window closes.

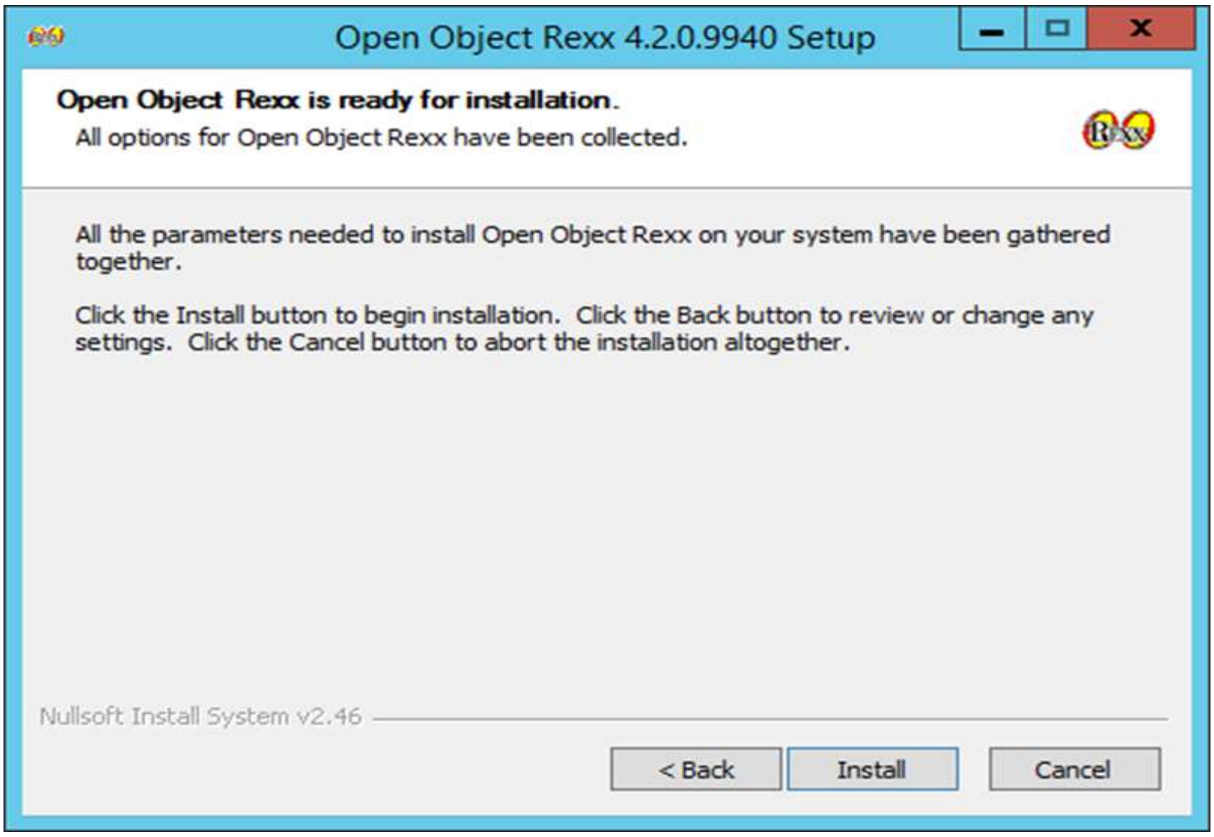
File Association for rexxpaws.exe

[ ] Create rexxpaws.exe file association

Extension (no spaces): .rexp File type name (no spaces): RexxPaws

Nullsoft Install System v2.46

Step 11: Finally click on the Install button to proceed with the installation.



Open Object Rexx 4.2.0.9940 Setup

Open Object Rexx is ready for installation.

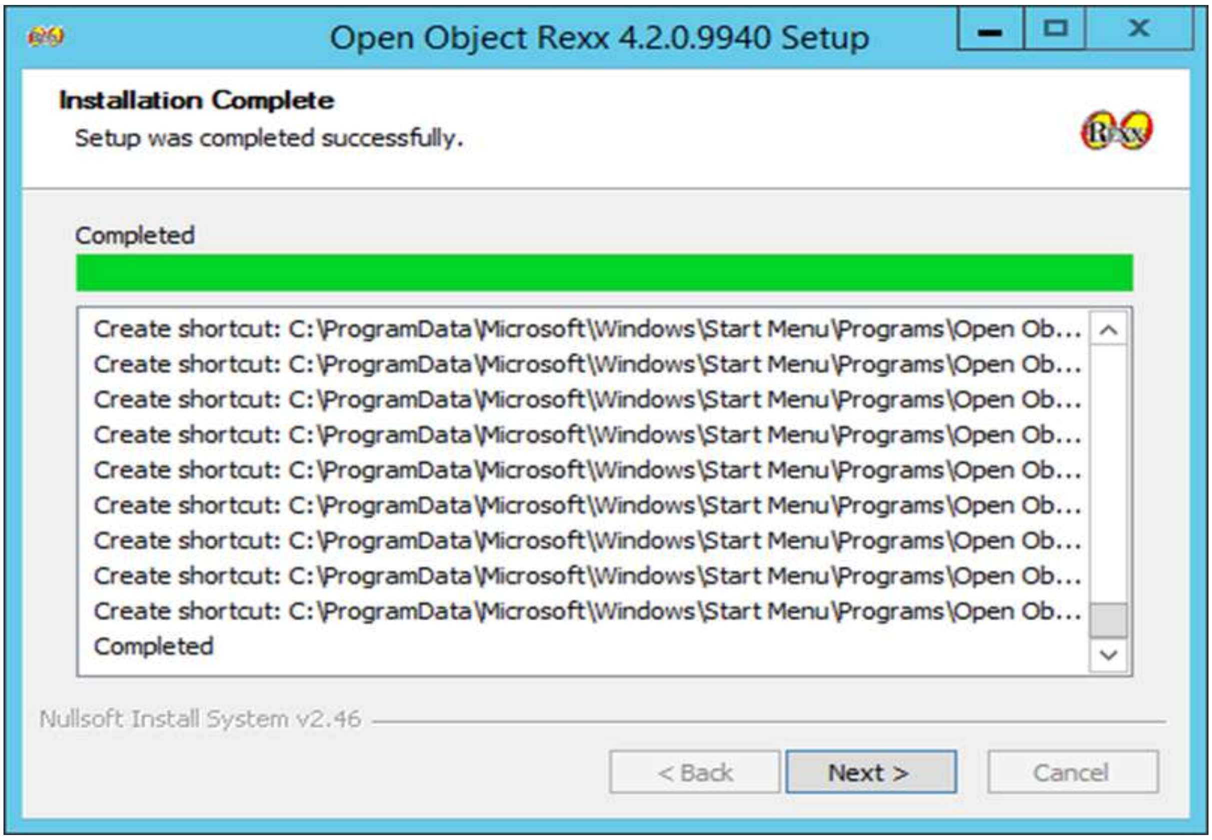
All options for Open Object Rexx have been collected.

All the parameters needed to install Open Object Rexx on your system have been gathered together.

Click the Install button to begin installation. Click the Back button to review or change any settings. Click the Cancel button to abort the installation altogether.

Nullsoft Install System v2.46

Step 12: Once the installation is complete, you need to click on the Next button to proceed further.



Open Object Rexx 4.2.0.9940 Setup

Installation Complete

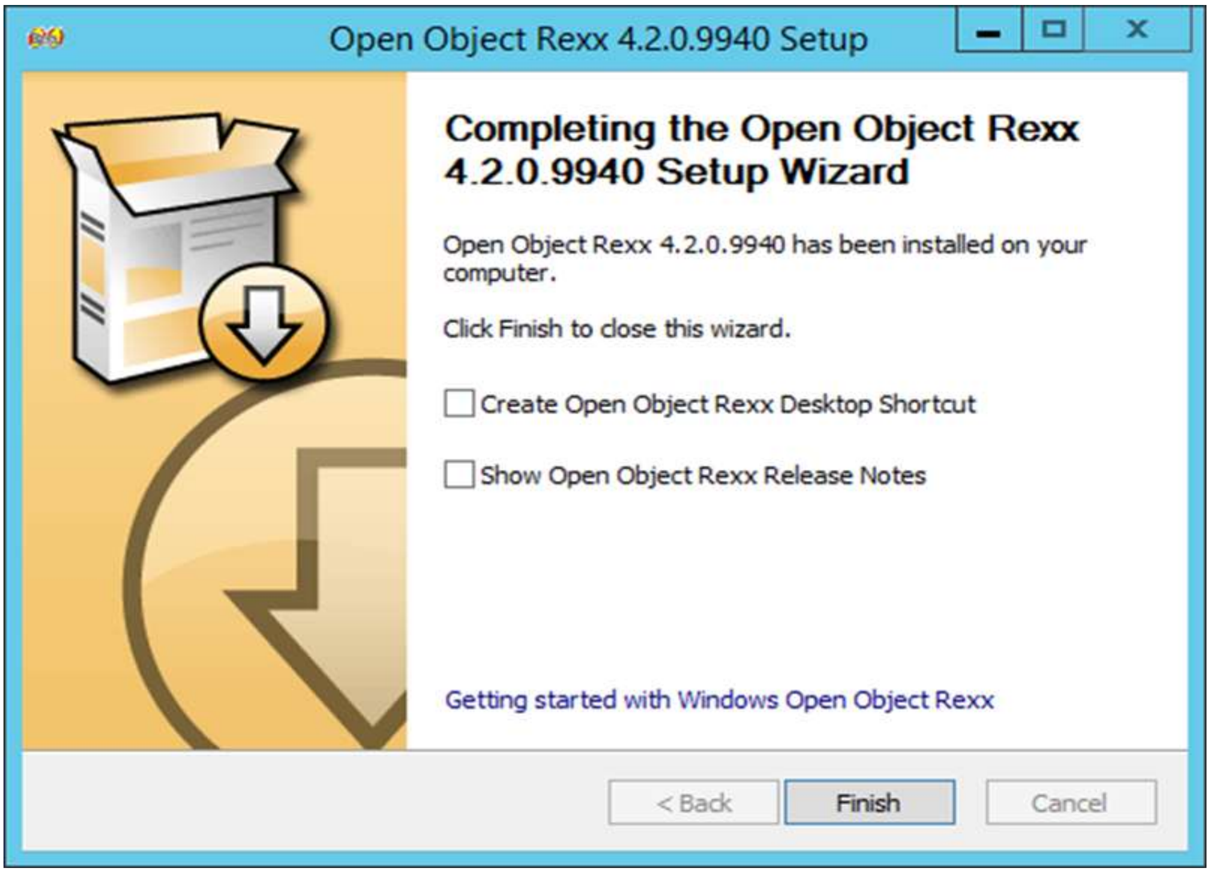
Setup was completed successfully.

Completed

```
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Open Ob...
Completed
```

Nullsoft Install System v2.46

Step 13: Click on the Finish button to complete the installation.


Open Object Rexx 4.2.0.9940 Setup

Completing the Open Object Rexx 4.2.0.9940 Setup Wizard

Open Object Rexx 4.2.0.9940 has been installed on your computer.

Click Finish to close this wizard.

☐ Create Open Object Rexx Desktop Shortcut

☐ Show Open Object Rexx Release Notes

Getting started with Windows Open Object Rexx

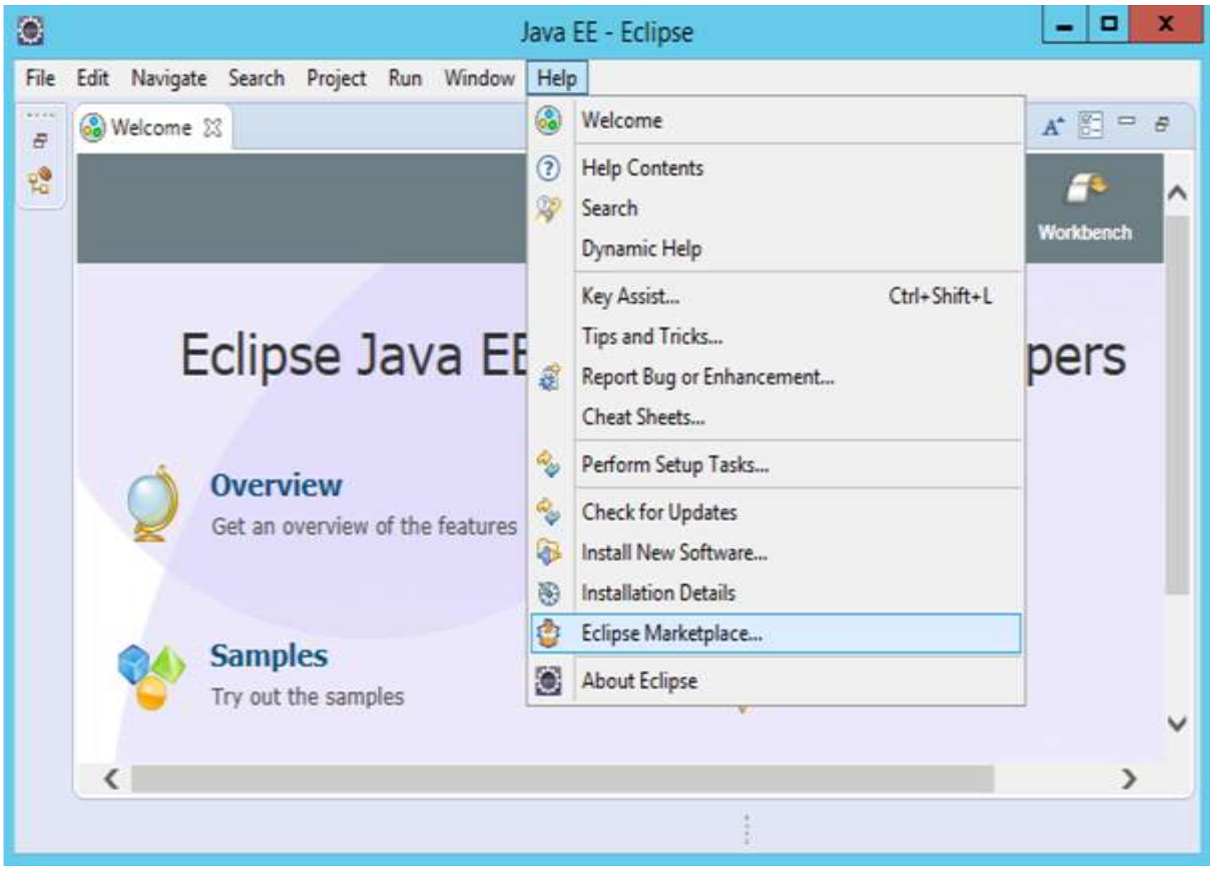
4. Rexx – Installation of Plugin-ins

In this chapter, we will discuss on how to install plug-ins on **popular IDE's (Integrated Development Environment)**. Rexx as a programming language is also available in popular IDE's such as **Eclipse**. Let's look at how we can get the required plugin's in these IDE's, so that you have more choices in working with Rexx.

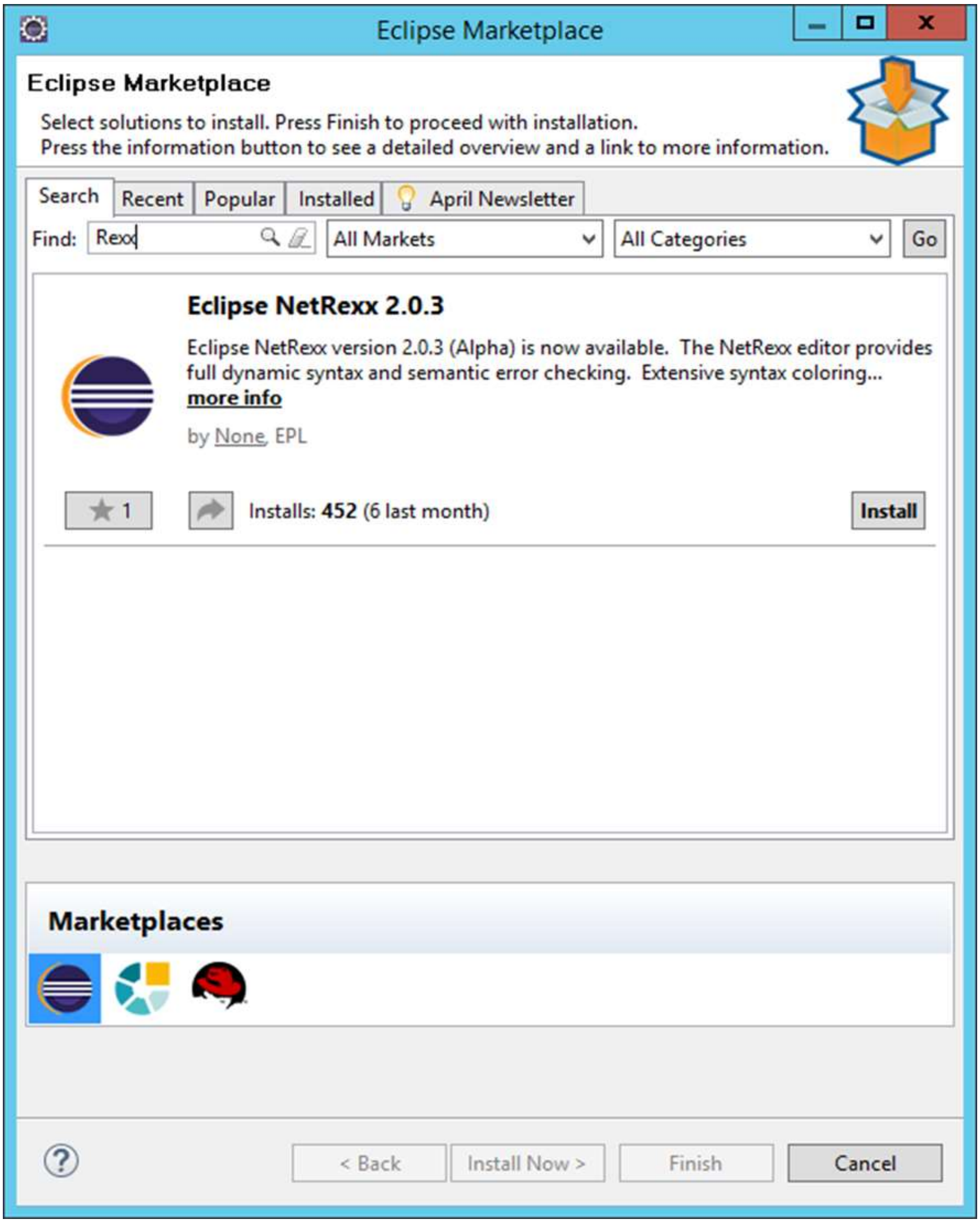
Installation in Eclipse

To make a trouble-free installation of Rexx in Eclipse, you will need to adhere to the following steps.

Step 1: Open Eclipse and click on the Menu item, **Help -> Eclipse Marketplace** as shown in the following screenshot.



Step 2: In the next dialog box, enter Rexx in the search criteria and click on the search button.


Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

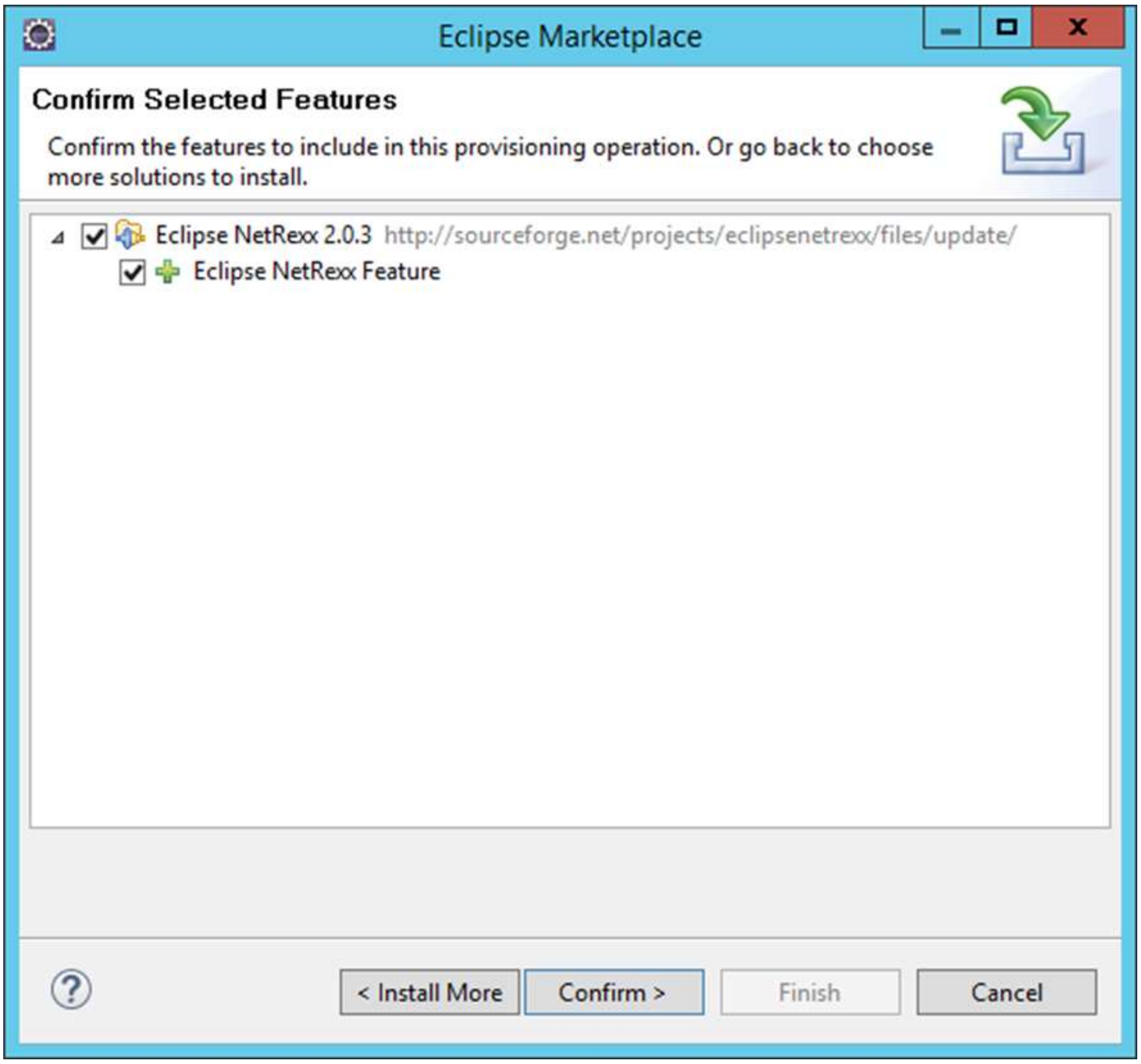
Eclipse NetRexx 2.0.3

Eclipse NetRexx version 2.0.3 (Alpha) is now available. The NetRexx editor provides full dynamic syntax and semantic error checking. Extensive syntax coloring...
more info

by None, EPL

Once done, click the Install button.

Step 3: Click on the Confirm button to further continue with the features installation.


Eclipse Marketplace

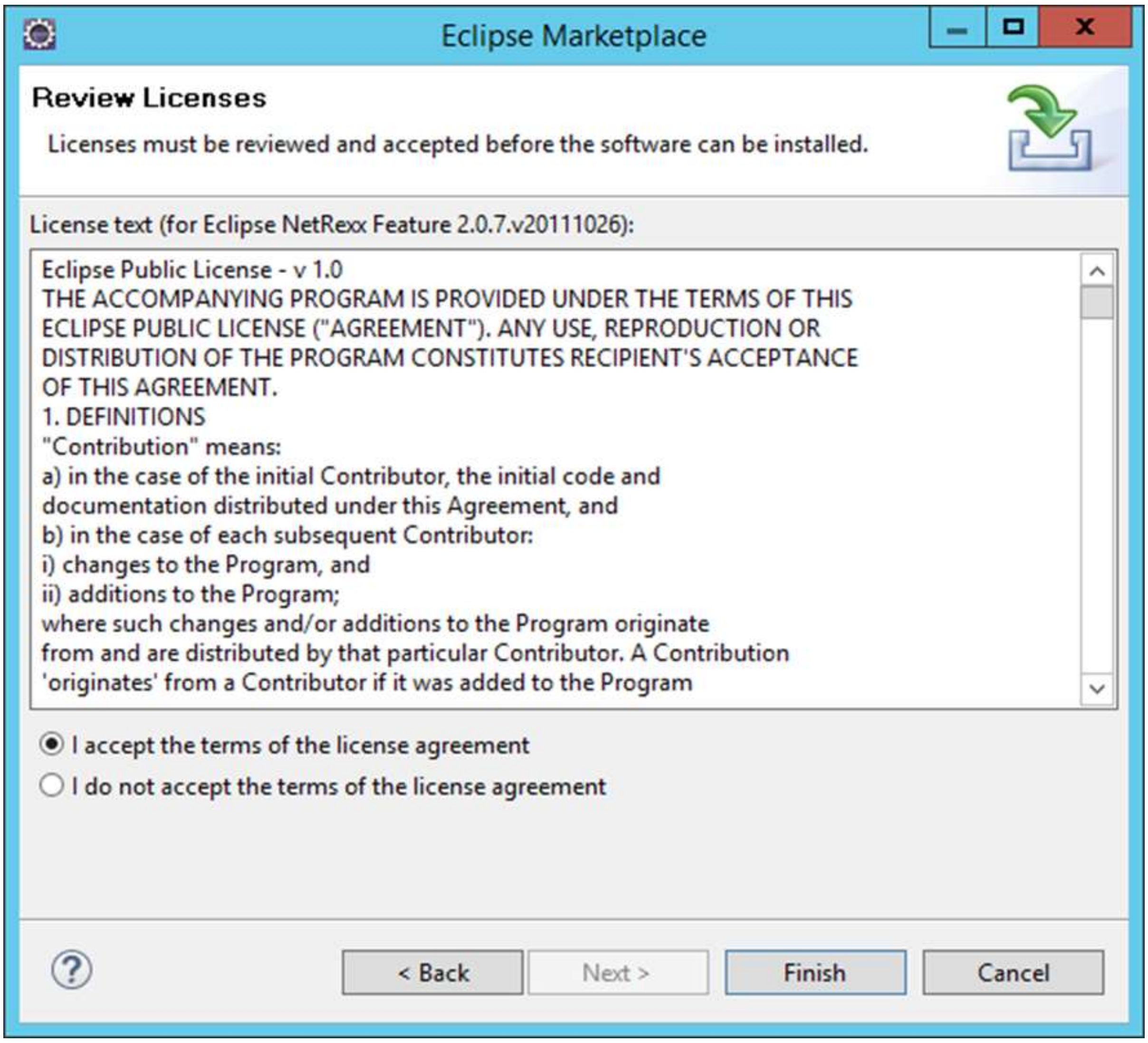
Confirm Selected Features

Confirm the features to include in this provisioning operation. Or go back to choose more solutions to install.

☑ Eclipse NetRexx 2.0.3 http://sourceforge.net/projects/eclipsenetrexx/files/update/

☑ Eclipse NetRexx Feature

Step 4: Eclipse will then download the necessary files to start off with the installation. Once done, Eclipse will ask for accepting the license agreements. Click on accepting the license agreements and then click on the Finish button as shown in the following screenshot.



Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

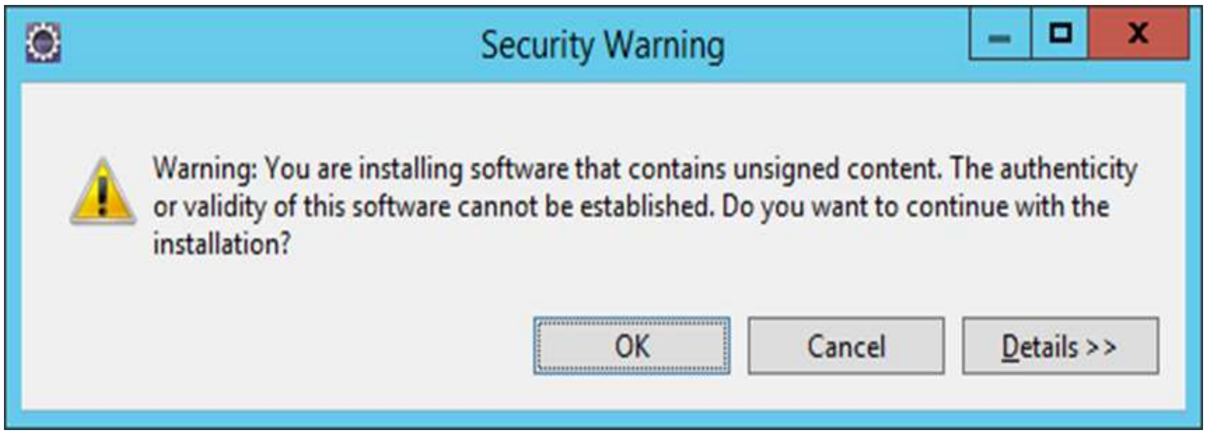
License text (for Eclipse NetRexx Feature 2.0.7.v20111026):

Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.
1. DEFINITIONS
"Contribution" means:
a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
b) in the case of each subsequent Contributor:
i) changes to the Program, and
ii) additions to the Program;
where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program

● I accept the terms of the license agreement
○ I do not accept the terms of the license agreement

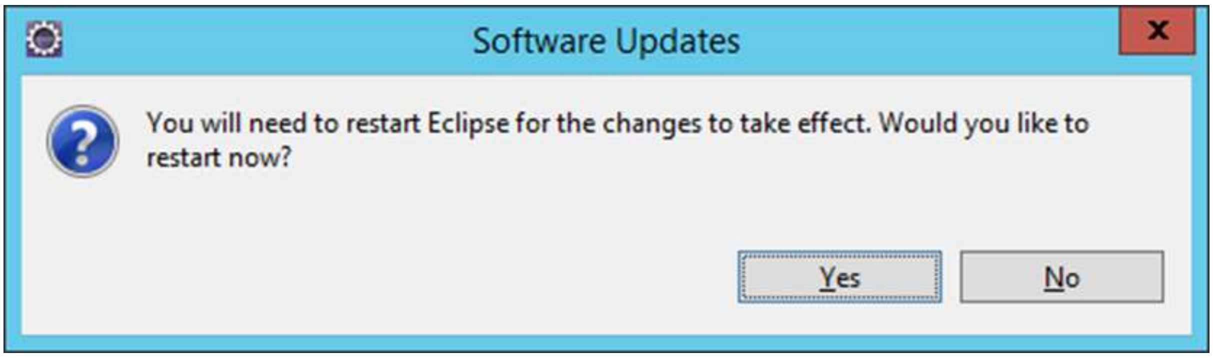
Eclipse will then start installing the software in the background.

Step 5: You will probably get a security warning (as shown in the following screenshot). Click on the OK button to proceed.



Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

Step 6: You will be prompted to restart Eclipse once the updates are installed. Click Yes to restart Eclipse.



You will need to restart Eclipse for the changes to take effect. Would you like to restart now?

5. Rexx – Basic Syntax

In order to understand the basic syntax of Rexx, let us first look at a simple Hello World program.

```
/* Main program */
say "Hello World"
```

One can see how simple the hello world program is. It is a simple script line which is used to execute the Hello World program.

The following things need to be noted about the above program:

- The **say command** is used to output a value to the console.
- The **/* */** is used for comments in Rexx.

The output of the above program will be:

```
Hello World
```

General Form of a Statement

In Rexx, let's see a general form of a program. Take a look at the following example.

```
/* Main program */
say add(5,6)
exit
add:
parse arg a,b
return a+b
```

Let's go through what we have understood from the above program:

- Add is a function defined to add 2 numbers.
- In the main program, the values of 5 and 6 is used as parameters to the add function.
- The exit keyword is used to exit from the main program. This is used to differentiate the main program from the add function.
- The add function is differentiated with the ':' symbol.
- The parse statement is used to parse the incoming arguments
- Finally, the return statement is used to return the sum of the numeric values.

Subroutines and Functions

In Rexx, the code is normally divided into subroutines and functions. Subroutines and functions are used to differentiate the code into different logical units. The key difference between subroutines and functions is that functions return a value whereas subroutines don't.

Below is a key difference example between a subroutine and a function for an addition implementation

Function Implementation

```
/* Main program */
say add(5,6)
exit
add:
parse arg a,b
return a+b
```

Subroutine Implementation

```
/* Main program */
add(5,6)
exit
add:
parse arg a,b
say a+b
```

The output of both the programs will be the value 11

Executing Commands

Rexx can be used as a control language for a variety of command-based systems. The way that Rexx executes commands in these systems is as follows. When Rexx encounters a program line which is neither an instruction nor an assignment, it treats that line as a string expression which is to be evaluated and then passed to the environment.

An example is as follows:

```
/* Main program */
parse arg command
command "file1"
command "file2"
command "file3"
exit
```

Each of the three similar lines in this program is a string expression which adds the name of a file (contained in the string constants) to the name of a command (given as a parameter). The resulting string is passed to the environment to be executed as a command. When the command has finished, the variable "rc" is set to the exit code of the command.

The output of the above program is as follows:

```
*-* command "file1"
       >>>   " file1"
       +++   "RC(127)"

     5 *-* command "file2"
       >>>   " file2"
       +++   "RC(127)"

     6 *-* command "file3"
       >>>   " file3"
       +++   "RC(127)"
```

Keywords in Rexx

The free syntax of REXX implies that some symbols are reserved for the language processor's use in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the **WHILE in a DO instruction**, and the **THEN** (which acts as a clause terminator in this case) following an **IF or WHEN clause**.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being taken to be keywords.

Comments in Rexx

Comments are used to document your code. Single line comments are identified by using the **/* */** at any position in the line.

An example is as follows:

```
/* Main program */
/* Call the add function */
add(5,6)
/* Exit the main program */
exit
add:
/* Parse the arguments passed to the add function */
parse arg a,b
/* Display the added numeric values */
say a+b
```

Comments can also be written in between a code line as shown in the following program:

```
/* Main program */
/* Call the add function */
add(5,6)
/* Exit the main program */
exit
add:
parse /* Parse the arguments passed to the add function */ arg a,b
/* Display the added numeric values */
say a+b
```

You can also have multiple lines in a comment as shown in the following program:

```
/* Main program
The below program is used to add numbers
Call the add function */
add(5,6)
exit
add:
parse arg a,b
say a+b
```

6. Rexx – DataTypes

In any programming language, you need to use various variables to store various types of information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory to store the value associated with that variable.

You may like to store information of various data types like String, Character, Wide Character, Integer, Floating Point, Boolean, etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Built-in Data Types

Rexx offers a wide variety of built-in data types. Following is a list of data types which are defined in Rexx.

- **Integer –** A string of numerics that does not contain a decimal point or exponent identifier. The first character can be **a plus (+) or minus (-) sign**. The number that is represented must be between -2147483648 and 2147483647, inclusive.

- **Big Integer –** A string of numbers that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -9223372036854775808 and -2147483648, inclusive, or between 2147483648 and 9223372036854775807.

- **Decimal –** It will be from one of the following formats:

    - A string of numerics that contains a decimal point but no exponent identifier. The **p** represents the precision and **s** represents the scale of the decimal number that the string represents. The first character can be a plus (+) or minus (-) sign.

    - A string of numerics that **does not contain a decimal point** or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented is less than -9223372036854775808 or greater than 9223372036854775807.

- **Float –** A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus (+) or minus (-) sign and a series of numerics). The string can begin with a plus (+) or minus (-) sign.

- **String –** A normal string of characters.

Following are some examples of how each data type can be used. Again each data type will be discussed in detail in the subsequent chapters. This is just to get you up to speed with a brief description of the above mentioned data types.

Integer

An example of how the number data type can be used is shown in the following program. This program shows the addition of 2 Integers.

```
/* Main program
The below program is used to add numbers
Call the add function */
add(5,6)

exit
add:

parse arg a,b
say a+b
```

The output of the above program will be:

```
11
```

Big Integer

The following program shows the capability of Rexx to handle big integers. This program shows how to add 2 big integers.

```
/* Main program
The below program is used to add numbers
Call the add function */
add(500000000000,6000000000000000000000)

exit
add:

parse arg a,b
say a+b
```

The output of the above program will be:

```
6.00000000E+21
```

Decimal

The following program shows the capability of Rexx to handle decimal numbers. This program shows how to add 2 decimal numbers.

```
/* Main program
The below program is used to add numbers
Call the add function */
add(5.5,6.6)

exit
add:

parse arg a,b
say a+b
```

The output of the above program will be:

```
12.1
```

Float

The following example show cases how a number can work as a float.

```
/* Main program
The below program is used to add numbers
Call the add function */
add(12E2,14E4)

exit
add:

parse arg a,b
say a+b
```

The output of the above program will be:

```
141200
```

String

An example of how the Tuple data type can be used is shown in the following program.

Here we are defining a **Tuple P** which has 3 terms. The **tuple_size** is an inbuilt function defined in Rexx which can be used to determine the size of the tuple.

```
/* Main program */
display("hello")

exit
display:

parse arg a
say a
```

The output of the above program will be:

```
hello
```

7. Rexx – Variables

In Rexx, all variables are bound with the **'='** statement. Variable names are sometimes referred to as symbols. They may be composed of Letters, Digits, and Characters such as **'. ! ? _'**. A variable name you create must not begin with a digit or a period. A simple variable name does not include a period. A variable name that includes a period is called a compound variable and represents an array or table.

The following are the basic types of variables in Rexx which were also explained in the previous chapter:

- **Integers –** This is used to represent an integer or a float. An example for this is 10.
- **Big integers –** This represents a large integer value.
- **Decimal –** A decimal value is a string of numerics that contains a decimal point but no exponent identifier.
- **Float –** A float value is a string that represents a number in the scientific notation.
- **String –** A series of characters defines a string in Rexx.

Different Types of Variable Functions

In this section, we will discuss regarding the various functions a variable can perform.

Variable Declarations

The general syntax of defining a variable is shown as follows:

```
var-name = var-value
```

where

- var-name – This is the name of the variable.
- var-value – This is the value bound to the variable.

The following program is an example of the variable declaration:

```
/* Main program */
X=40
Y=50
Result=X+Y
say Result
```

In the above example, we have 2 variables, one is **X** which is bound to the value **40** and the next is **Y** which is bound to the value of **50**. Another variable called Result is bound to the addition of **X and Y**.

The output of the above program will be as follows:

```
90
```

Naming Variables

Variable names are sometimes referred to as symbols. They may be composed of Letters, Digits, and Characters such as **'**. ! ? _**'** . A variable name you create must not begin with a digit or period.

If a variable has not yet been assigned a value, it is referred to as uninitialized. The value of an uninitialized variable is the name of the variable itself in uppercase letters.

An example of an unassigned variable is as follows:

```
/* Main program */
unassignedvalue
say unassignedvalue
```

If you run the above program you will get the following output:

```
UNASSIGNEDVALUE: command not found
     2 *-* unassignedvalue
       >>>   "UNASSIGNEDVALUE"
       +++   "RC(127)"
UNASSIGNEDVALUE
```

Variables can be assigned values more than once. The below program shows how the value of X can be assigned a value multiple times.

```
/* Main program */
X=40
X=50
say X
```

The output of the above program will be as follows:

```
50
```

Printing Variables

The values of variables are printed using the **say** command. Following is an example of printing a variety number of variables.

```
/* Main program */
X=40
/* Display an Integer */
say X
Y=50.5
/* Display a Float */
say Y
Z = "hello"
/* Display a string */
say Z
```

The output of the above program will be as follows:

```
40
50.5
hello
```

8. Rexx – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Rexx has various types of operators which are also explained in detail as follows:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators

Arithmetic Operators

The Rexx language supports the normal Arithmetic Operators as any the language. Following are the Arithmetic Operators available in Rexx.

Operator	Description	Example
+	Addition of two operands	1 + 2 will give 3
–	Subtracts second operand from the first	1 - 2 will give -1
*	Multiplication of both operands	2 * 2 will give 4
/	Division of numerator by denominator	2 / 2 will give 1
//	Remainder of dividing the first number by the second	3 // 2 will give 1
%	The div component will perform the division and return the integer component.	3 % 2 will give 1

The following program shows how the various operators can be used.

```
/* Main program*/
X = 40
Y = 50
Res1 = X + Y
Res2 = X - Y
Res3 = X * Y
Res4 = X / Y
Res5 = X % Y
Res6 = X // Y
say Res1
say Res2
say Res3
say Res4
say Res5
say Res6
```

The output of the above program will be:

```
90
-10
2000
0.8
0
40
```

Relational Operators

Relational Operators allow of the comparison of objects. Following are the relational operators available in Rexx. In Rexx the true value is denoted by 1 and the false value is denoted by 0.

Operator	Description	Example
==	Tests the equality between two objects	2 = 2 will give 1
<	Checks to see if the left object is less than the right operand.	2 < 3 will give 1

=<	Checks to see if the left object is less than or equal to the right operand.	2 =<3 will give 1
>	Checks to see if the left object is greater than the right operand.	3 >2 will give 1
>=	Checks to see if the left object is greater than or equal to the right operand.	3 > 2 will give 1

The following program shows how the various operators can be used.

```
/* Main program*/
X = 3
Y = 2
say X>Y
say X<Y
say X>=Y
say X<=Y
say X==Y
```

The output of the above program will be:

```
1
0
1
0
0
```

Logical Operators

Logical Operators are used to evaluate Boolean expressions. Following are the logical operators available in Rexx.

Operator	Description	Example
&	This is the logical "and" operator	1 or 1 will give 1
|	This is the logical "or" operator	1 and 0 will give 0

\	This is the logical "not" operator	\0 will give 1
&&	This is the logical exclusive "or" operator	1 && 0 will give 1

The following program shows how the various operators can be used.

```
/* Main program*/
say 1 & 0
say 1 | 0
say 1 && 0
say \1
```

The output of the above program will be:

```
0
1
1
0
```

Bitwise Operators

Groovy provides four bitwise operators. Below are the bitwise operators available in Groovy

Operator	Description
bitand	This is the bitwise "and" operator
bitor	This is the bitwise "or" operator
bitxor	This is the bitwise "xor" or Exclusive or operator

Following is the truth table showcasing these operators:

p	q	p bitand q	p bitor q	p bitxor q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

The following program shows how the various operators can be used.

```
/* Main program */
a=21
b=347
Say c2b(a)
Say c2b(b)
Say c2b(bitand(a,b))
Say c2b(bitor(a,b))
Say c2b(bitxor(a,b))
Exit
c2b: return x2b(c2x(arg(1)))
```

The output of the above program will be:

```
0011001000110001
001100110011010000110111
001100100011000000110111
001100110011010100110111
000000010000010100110111
```

Operator Precedence

The following table shows the Operator Precedence for the Rexx operators in order of descending priority of their precedence.

Operators	Precedence
Prefix operators	+ - \
Addition and subtraction	+ -
Comparison operators	= == > < >= <=
Logical AND	&
Logical OR	|
EXCLUSIVE OR	&&

9. Rexx – Arrays

Arrays in any programming language allow you to group a list of values of the same type. The use of arrays is that it allows you to build a list of similar type of values which are **sortable, searchable** and can be **easily manipulated**. Rexx also allows one to define arrays. These arrays can be one dimensional or multidimensional.

Rexx arrays may be sparse. That is, not every array position must have a value or even be initialized. There can be empty array positions, or slots, between those that do contain data elements. Or arrays can be dense, in which consecutive array slots all contain data elements.

In many programming languages, you must be concerned with what the subscript of the first entry in a table is. Is the first numeric subscript 0 or 1? In Rexx, the first subscript is whatever you use! So, input the first array element into position 0 or 1 as you prefer.

```
array_name.0 = ‘first element’
```

or

```
array_name.1 = ‘first element’
```

Let’s look at the different operations available for arrays.

Creating Arrays

Arrays are created with the same naming convention which is used for variables in Rexx.

The general syntax for creating arrays is as follows:

```
Arrayname.index=value
```

where

- **Arrayname –** This is the name provided to the array.
- **Index –** This is the index position in the array to refer to a specific element.
- **Value –** This is the value assigned to the index element in the array.

An example of an array declaration is as follows:

```
/* Main program */
list.1=0
list.2=0
list.3=0
```

The following points needs to be noted about the above program:

- The name of the array is given as list
- There are 3 elements of the array which are initialized to the value of 0.

Assigning Values to an Array Element

Values can be re-assigned to array elements in the same way as array elements are initialized.

The following program is an example of values which can be assigned to various index values of an existing array.

```
/* Main program */
list.1=0
list.2=0
list.3=0
/* Assigning new values to the array*/
list.1=10
list.3=30
```

Displaying Values of an Array

The values of an array can be displayed by referring to the index position of the array element. The following example shows to access various elements of the array.

```
/* Main program */
list.1=0
list.2=0
list.3=0
/* Assigning new values to the array*/
list.1=10
list.3=30
say list.1
say list.2
say list.3
```

The output of the above program will be as follows:

```
10
0
30
```

Copying Arrays

All of the elements of an array can be copied onto another array. The general syntax of this is as follows:

```
Newarray. = sourcearray.
```

where

- **Newarray –** This is the new array in which the elements need to be copied onto.
- **Sourcearray –** This is the source array from which the elements need to be copied.

An example on how the copy operations for arrays can be carried out is shown in the following program:

```
/* Main program */
list.1=0
list.2=0
list.3=0
/* Assigning new values to the array*/
list.1=10
list.3=30
listnew. = list.
say listnew.1
say listnew.2
say listnew.3
```

The output of the above program will be:

```
10
0
30
```

Iterating through array elements

Elements of an array can also be iterated by using the iterative statements available in Rexx. An example on how this can be done is as follows:

```
/* Main program */
list.1=10
list.2=20
list.3=30
```

```
number_of_elements=3
do j = 1 to number_of_elements
say list.j
end
```

The following pointers need to be noted about the above program:

- The **do loop** is used to iterate through the array elements.
- The variable **number_of_elements** is used to store the number of elements in the array.
- The **variable j** is used to iterate through each element of the array.

The output of the above program will be:

```
10
20
30
```

Two-dimensional Arrays

It was also mentioned that we can construct multi-dimensional arrays in Rexx. Let's look at an example of how we can implement a 2-dimensional array.

```
/* Main program */
list.1=10
list.1.1=11
list.1.2=12
say list.1
say list.1.1
say list.1.2
```

The output of the above program will be shown as follows:

```
10
11
12
```

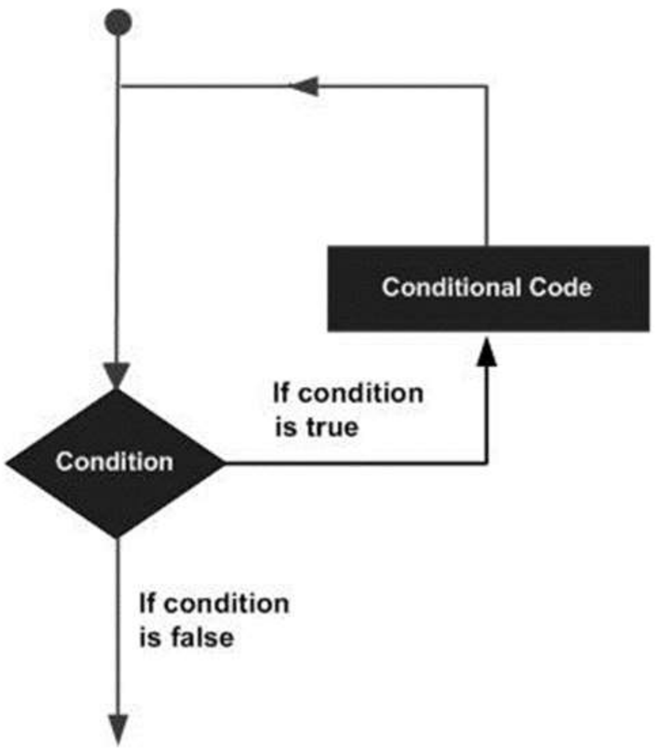
The following point needs to be noted about the above program –

- To create a multidimensional array, we can use another layer of indexing. So in our example, we used **list.1.1** to create another inner array for the index value 1 of the list array.

10. Rexx – Loops

So far we have seen statements which have been executed one after the other in a sequential manner. Additionally, statements are provided in Rexx to alter the flow of control in a program's logic. They are then classified into a flow of control statements which we will study in detail.

A loop statement allows us to execute a statement or group of statements multiple times. The following illustration is the general form of a loop statement in most of the programming languages.



Let's discuss the various loops supported by Rexx.

The do Loop

The **do loop** is used to execute a number of statements for a certain number of times. The number of times that the statement needs to be executed is determined by the value passed to the do loop.

Syntax

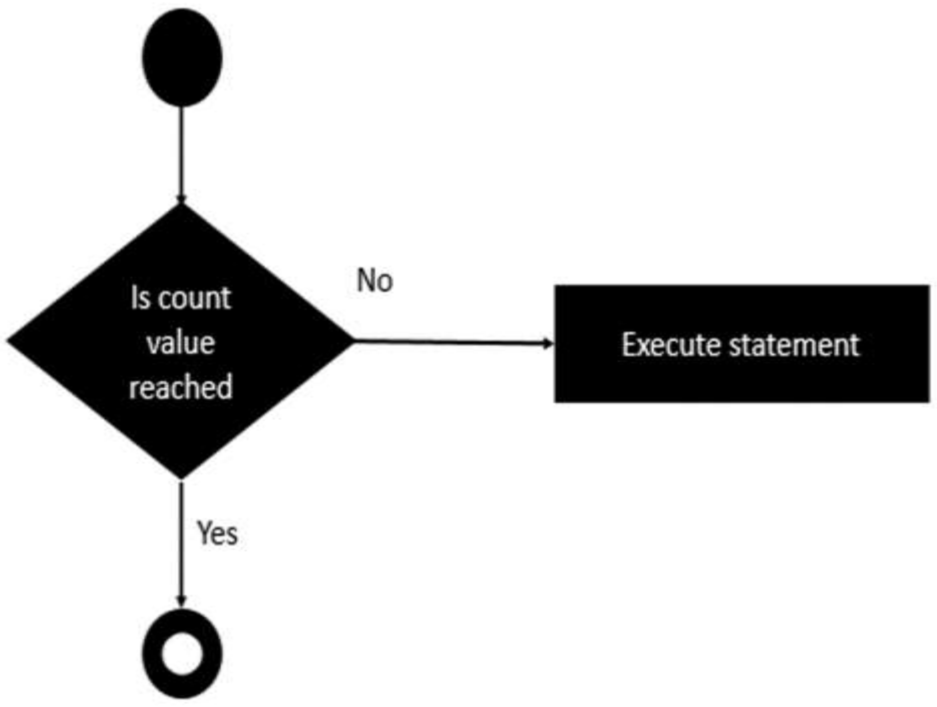
The syntax of the do loop statement is as follows

```
do count
statement #1
statement #2
...
End
```

Statement#1 and Statement#2 are a series of a block of statements that get executed in the do loop. The count variable symbolizes the number of times the do loop needs to be executed.

Flow Diagram

The following diagram shows the diagrammatic explanation of this loop.



The key point to note about this loop is that the **do** loop will execute based on the value of the count variable.

Example

The following program is an example of a do loop statement.

```
/* Main program */
do 5
say "hello"
end
```

The following key points need to be noted about the above program:

- The count in the above case is 5. So the do loop will be run for 5 times.
- In the do loop, the phrase hello is displayed to the console.

The output of the above program will be:

```
hello
hello
hello
hello
hello
```

The do-while Loop

The do-while statement is used to simulate the simple while loop which is present in other programming languages.

Syntax

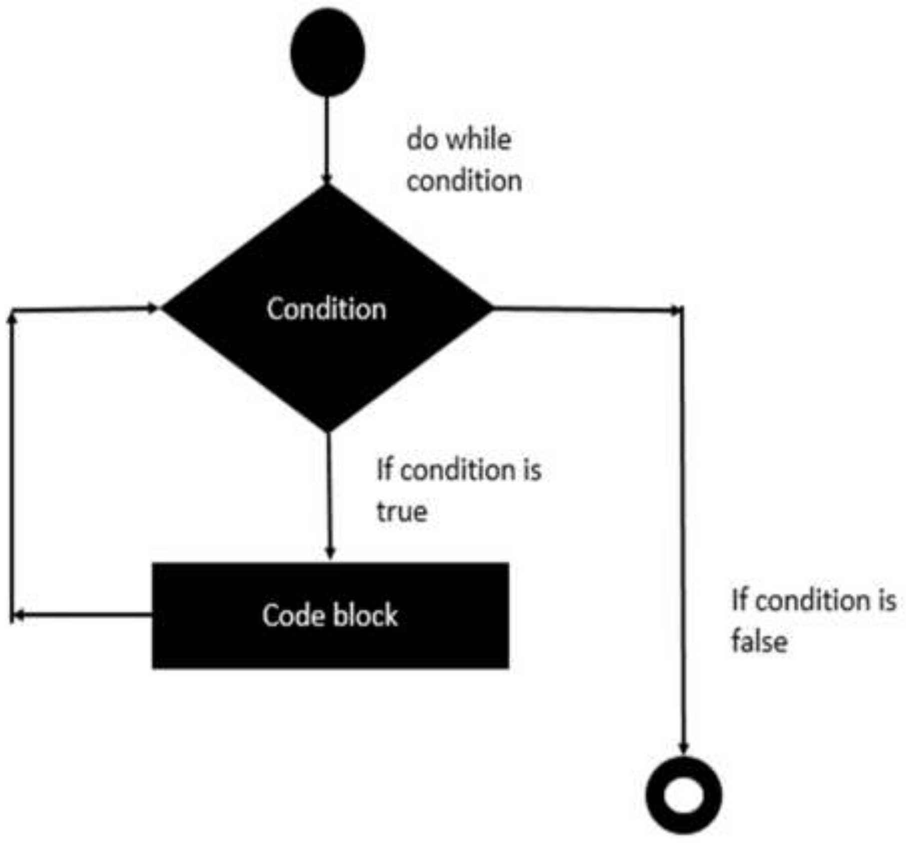
The syntax of the do-while statement is as follows:

```
do while (condition)
statement #1
statement #2
...
end
```

The while statement is executed by first evaluating the condition expression (a Boolean value), and if the result is true, then the statements in the while loop are executed. The process is repeated starting from the evaluation of the condition in the while statement. This loop continues until the condition **evaluates to false**. When the condition is false, the loop terminates. The program logic then continues with the statement immediately following the while statement.

Flow Diagram

The following diagram shows the diagrammatic explanation of this loop.



The key point to note is that the code block runs till the condition in the do loop **evaluates to true**. As soon as the condition evaluates to false, the do loop exits.

Example: The following program is an example of a do-while loop statement.

```
/* Main program */
j=1
do while(j <= 10)
say j
j=j+1
end
```

The following key points need to be noted about the above program.

- We are defining a recursive function called do while which would simulate the implementation of our while loop.
- We are initializing the variable j to a value of 1. This value will be incremented in our do-while loop.

- For each value of j, the do-while loop evaluates whether the value of j is less than or equal to 10. If so, it displays the value of j and increments the value of j accordingly.

The output of the above code will be:

```
1
2
3
4
5
6
7
8
9
10
```

The do-until Loop

The do-until loop is a slight variation of the do while loop. This loop varies in the fact that is exits when the condition being evaluated is false.

Syntax

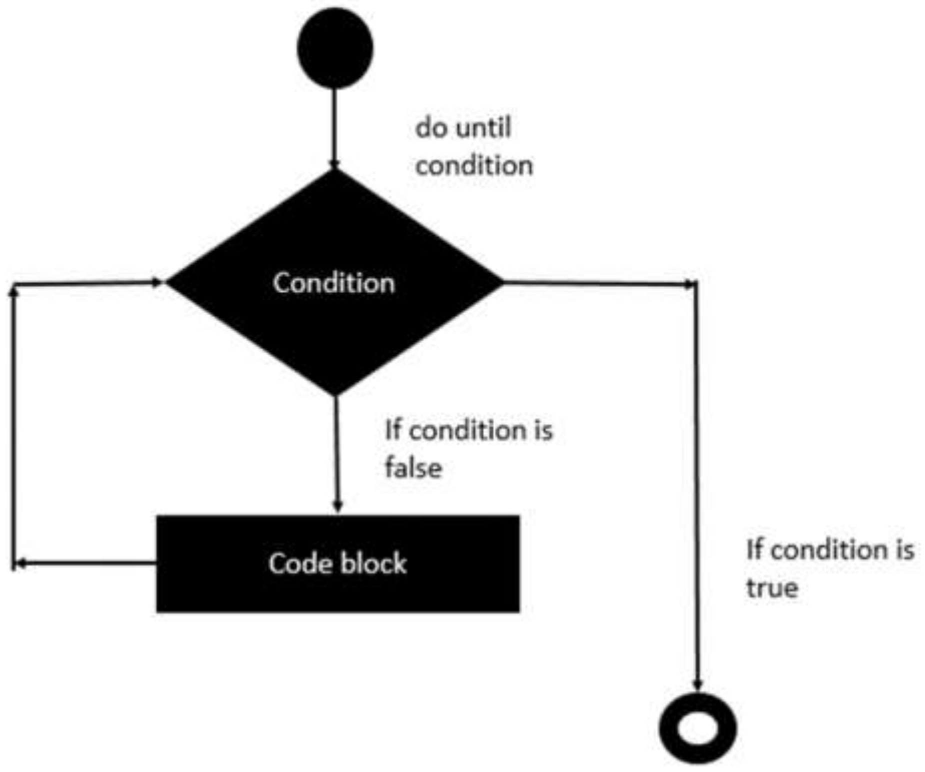
The syntax of the do-until statement is as follows.

```
do until (condition)
statement #1
statement #2
...
end
```

The do-until statement is different from the do-while statement in the fact, that it will only execute the statements until the condition evaluated is true. If the **condition is true,** then the loop is exited.

Flow Diagram

The following diagram shows the diagrammatic explanation of this loop.



The key thing to note is that the code block runs till the condition in the do-until **evaluates to false**. As soon as the condition evaluates to true, the do loop exits.

Example

The following program is an example of a do-until loop statement.

```
/* Main program */
j=1
do until (j <= 10)
say j
j=j+1
end
```

The output of the above code will be:

```
1
```

Controlled Repetition

The do loops can be catered to carry out a controlled repetition of statements.

Syntax

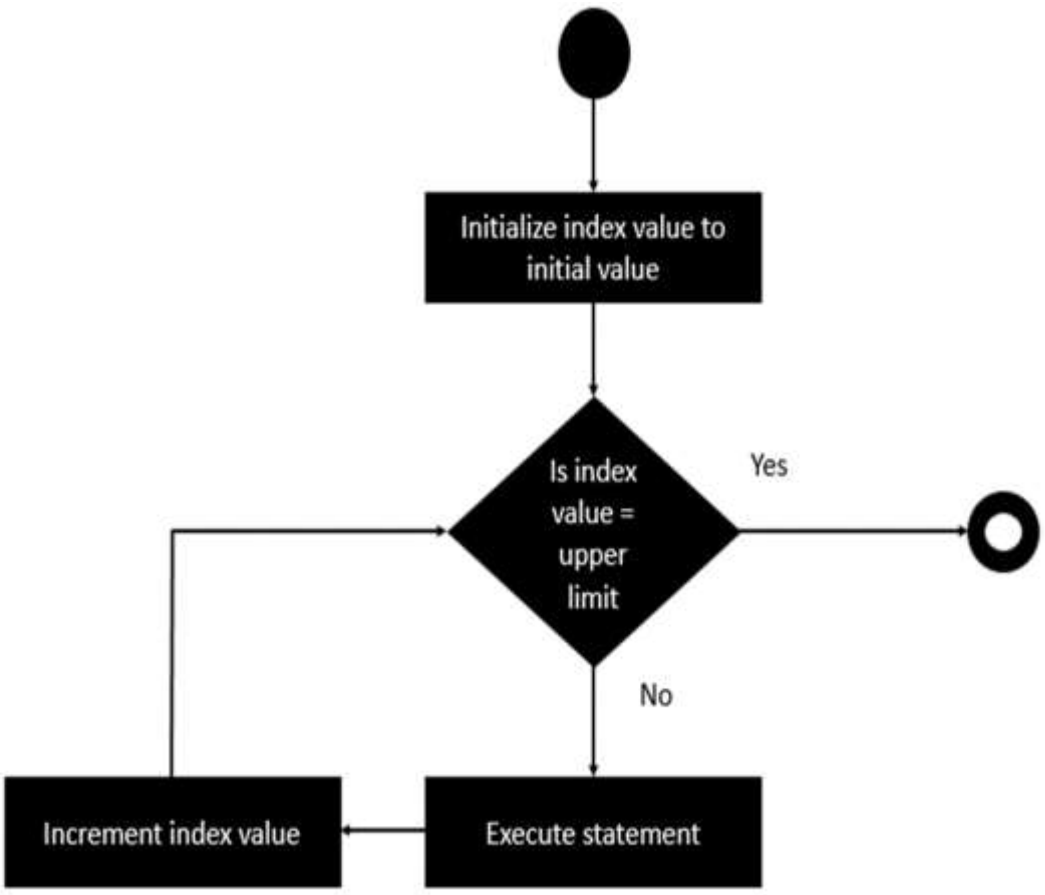
The general syntax of this sort of statement is as follows.

```
do index = start [to limit] [by increment] [for count]

statement #1

statement #2

end
```

The difference in this statement is that there is an index which is used to control the number of times the loop is executed. Secondly, there are parameters which state the value which the index should start with, where it should end and what is the increment value.

Flow Diagram

Let's check out the flow diagram of this loop:


Initialize index value to initial value
Is index value = upper limit
Yes
No
Execute statement
Increment index value

From the above diagram you can clearly see that the loop is executed based on the index value and how the index value is incremented.

Example

The following program is an example of the controlled repetition statement.

```
/* Main program */
do i = 0 to 5 by 2
say "hello"
end
```

In the above program, the value of the **count i** is set to 0 first. Then it is incremented in counts of 2 till the value is not greater than 5.

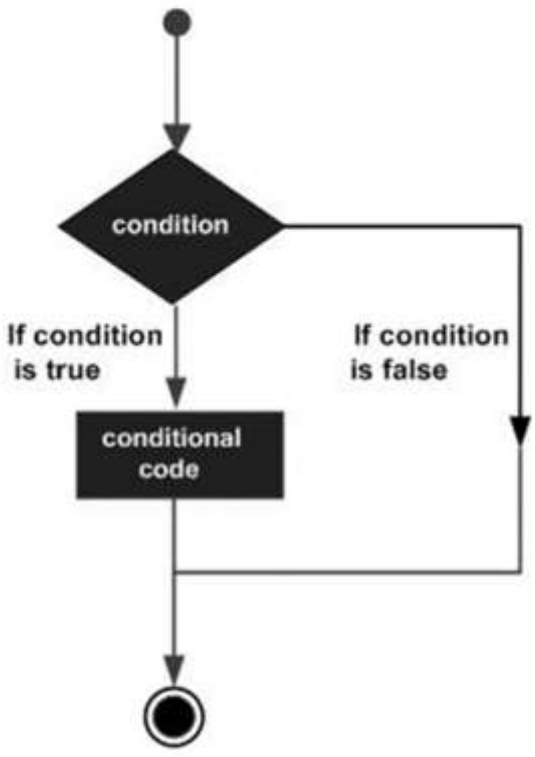
The output of the above code will be:

```
hello
hello
hello
```

11. Rexx – Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program.

The following diagram shows the general form of a typical decision-making structure found in most of the programming languages.



There is a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Let’s look at the various decision-making statements available in Rexx.

If statement

The first decision-making statement is the **if** statement. An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

The general form of this statement in Rexx is as follows:

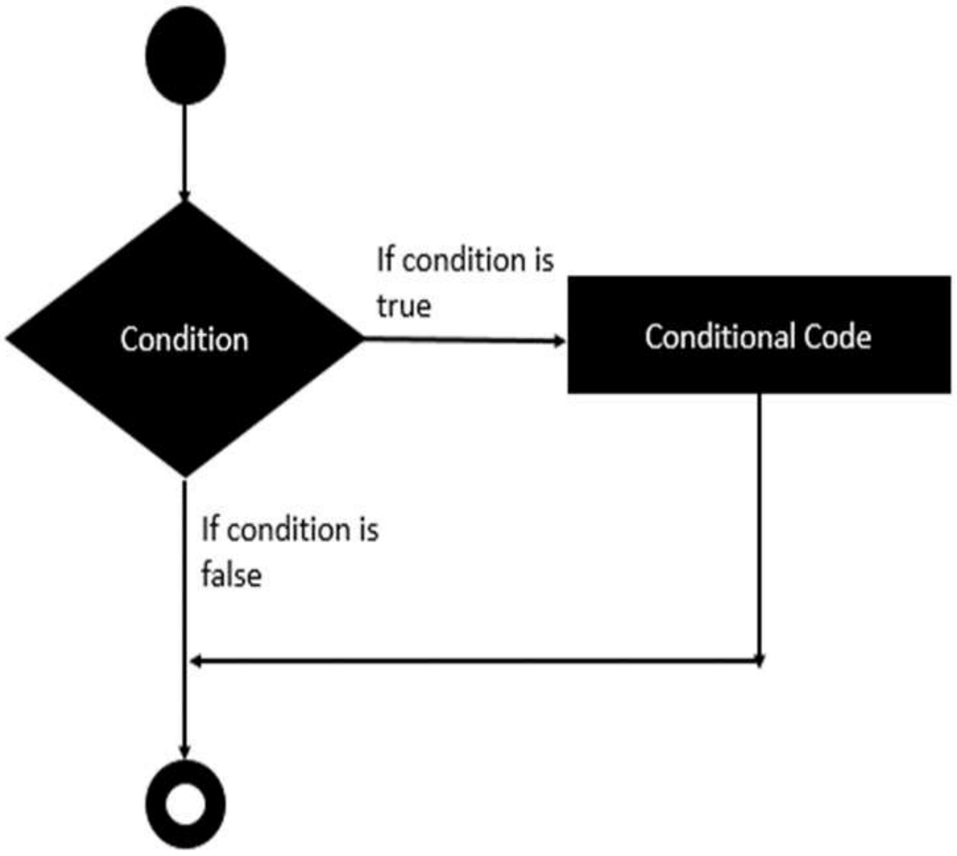
```
if (condition) then
 do
#statement1
```

```
#statement2
end
```

In Rexx, the condition is an expression which evaluates to either true or false. If the condition is true, then the subsequent statements in the loop are executed.

Flow Diagram

The following diagram shows the diagrammatic explanation of this loop.



In the above diagram, you can see that only if the condition is evaluated to true does the conditional code gets executed.

Example

The following program is an example of the **simple if expression** in Rexx.

```
/* Main program */
i=5
if (i<10) then
do
say "i is less than 10"
end
```

The following key things need to be noted about the above program:

- The if statement is used to first evaluate if the **value of i** is less than 10.
- If yes, then the statement inside the do loop is evaluated.

The output of the above program will be:

```
i is less than 10
```

If-else statement

The next decision-making statement is the if-else statement. An **if** statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax

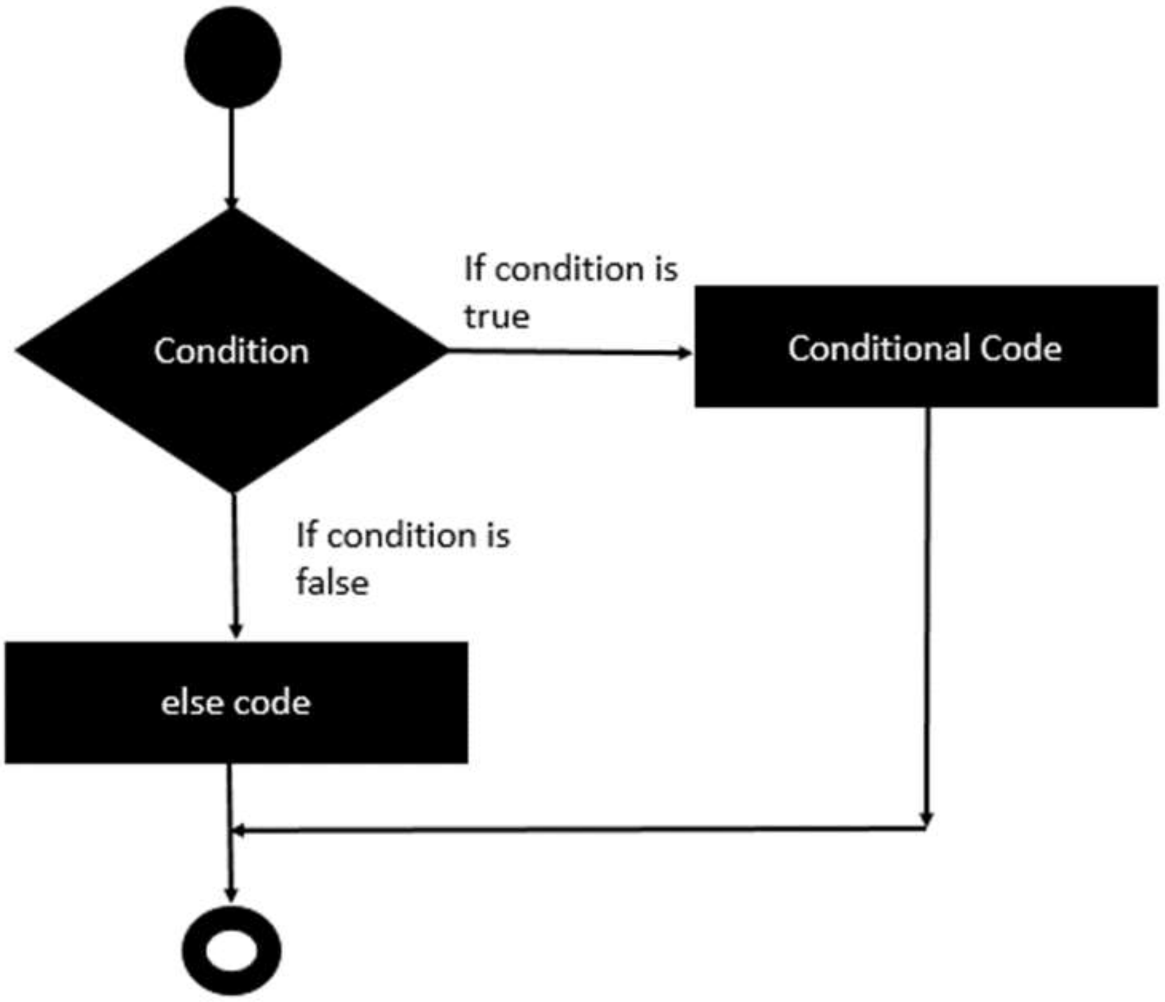
The general form of this statement in Rexx is as follows.

```
if (condition) then

 do

#statement1

#statement2

end

else

do

#statement3

#statement4

end
```

In Rexx, the condition is an expression which evaluates to either true or false. If the condition is true, then the subsequent statements are executed. Else if the condition is **evaluated to false**, then the statements in the else condition are evaluated.

Flow Diagram

The flow diagram of the if-else statement is as follows:



From the above diagram, it can be noted that we have two code blocks. One gets executed if the condition is **evaluated to true** and the other if the code is **evaluated to false**.

Example

The following program is an example of the simple if-else expression in Rexx.

```
/* Main program */
i=50
if (i<10) then
do
say "i is less than 10"
end
```

```
else
do
say "i is greater than 10"
end
```

The output of the above program is as follows:

```
i is greater than 10
```

Nested If Statements

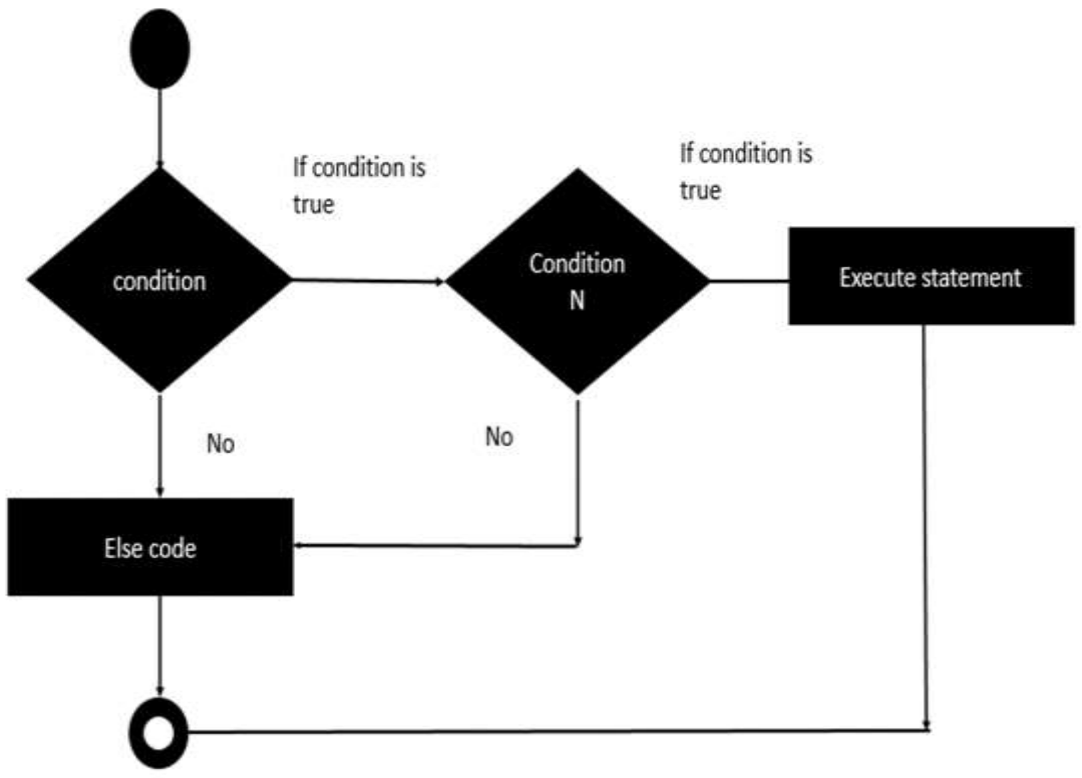
Sometimes there is a requirement to have **multiple if statements** embedded inside each other, as is possible in other programming languages. In Rexx also this is possible.

Syntax

```
if (condition1) then
 do
#statement1
end
else
if (condition2) then
do
#statement2
end
```

Flow Diagram

The flow diagram of nested **if** statements is as follows:



Example

Let’s take an example of nested **if** statement:

```
/* Main program */
i=50
if (i<10) then
do
say "i is less than 10"
end
else
if (i<7) then
do
say "i is less than 7"
end
else
do
say "i is greater than 10"
end
```

The output of the above program will be:

```
i is greater than 10
```

Select Statements

Rexx offers the select statement which can be used to execute expressions based on the output of the select statement.

Syntax

The general form of this statement is –

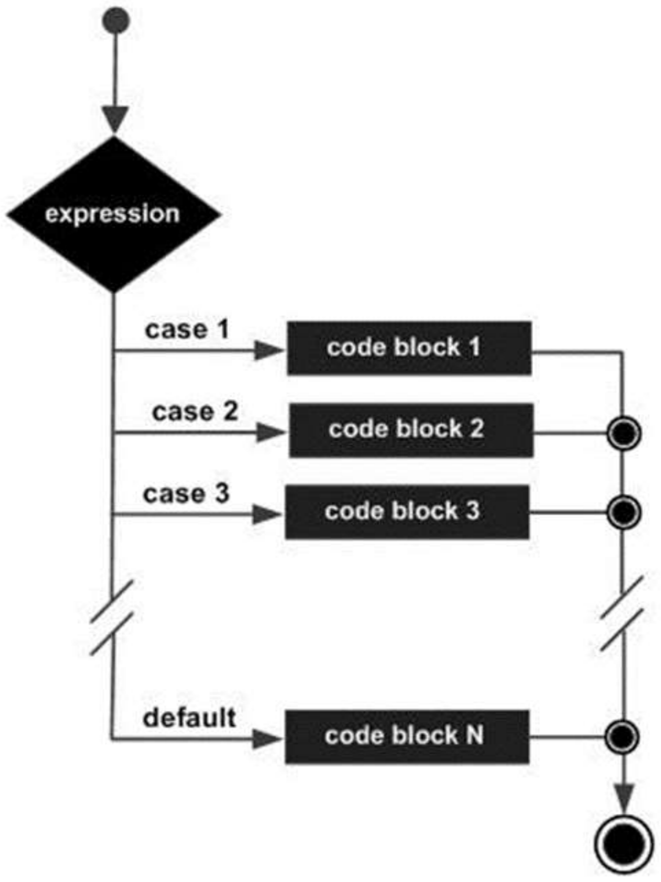
```
select

when (condition#1) then

statement#1

when (condition#2) then

statement#2

otherwise

defaultstatement

end
```

The general working of this statement is as follows:

- The select statement has a range of when statements to evaluate different conditions.
- Each **when clause** has a different condition which needs to be evaluated and the subsequent statement is executed.
- The otherwise statement is used to run any default statement if the previous when conditions do not **evaluate to true**.

Flow Diagram

The flow diagram of the **select** statement is as follows:



Example

The following program is an example of the **case** statement in Rexx.

```
/* Main program */
i=50
select
when(i<=5) then
say "i is less than 5"
when(i<=10) then
say "i is less than 10"
otherwise
say "i is greater than 10"
end
```

The output of the above program would be:

```
i is greater than 10
```

12. Rexx – Numbers

Rexx has the following data types when it comes to numbers.

- **Integer –** A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -2147483648 and 2147483647, inclusive.
- **Big Integer –** A string of numbers that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -9223372036854775808 and -2147483648, inclusive, or between 2147483648 and 9223372036854775807.
- **Decimal –** One of the following formats:
    - A string of numerics that contains a decimal point but no exponent identifier, where **p** represents the precision and **s** represents the scale of the decimal number that the string represents. The first character can be a plus (+) or minus (-) sign.
    - A string of numerics that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented is less than -9223372036854775808 or greater than 9223372036854775807.
- **Float –** A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus (+) or minus (-) sign and a series of numerics). The string can begin with a plus (+) or minus (-) sign.

Let’s now look at the different **methods available for numbers**.

ABS

This method returns the absolute value of an input number.

Syntax

```
ABS(i)
```

Parameter

i – This is the input number for which the absolute value needs to be determined

Return Value

The return value is the absolute value of the input number.

Example

```
/* Main program */
i=50.5
j=-40.4
say ABS(i)
say ABS(j)
```

Output

When we run the above program, we will get the following result.

```
50.5
40.4
```

MAX

This method returns the maximum value from a list of numbers.

Syntax

```
MAX(number1,number2, …numberN)
```

Parameters

number1, number2, …numberN – The list of numbers for which the maximum values need to be determined.

Return Value

The maximum value from the list of values.

Example

```
/* Main program */
say MAX(10,20,30)
```

Output

When we run the above program we will get the following result.

```
30
```

MIN

This method returns the minimum value from a list of numbers.

Syntax

```
MIN(number1,number2, …numberN)
```

Parameters

number1, number2, …numberN – The list of numbers for which the minimum values need to be determined.

Return Value

The minimum value from the list of values.

Example

```
/* Main program */
say MIN(10,20,30)
```

Output

When we run the above program we will get the following result.

```
10
```

RANDOM

This method returns a random generated number.

Syntax

```
RANDOM()
```

Parameters

None

Return Value

This method returns a random generated number.

Example

```
/* Main program */
say RANDOM()
```

Output

When we run the above program, we will get the following result.

```
971
```

SIGN

Returns 1 if number is greater than 0, or 0 if the number is 0, or -1 if the number is less than 0.

Syntax

```
SIGN(i)
```

Parameters

i – This the input number for which the sign needs to be determined.

Return Value

Returns 1 if number is greater than 0, or 0 if the number is 0, or -1 if the number is less than 0.

Example

```
/* Main program */
say SIGN(12)
say SIGN(-12)
```

Output

When we run the above program we will get the following result

```
1
-1
```

TRUNC

This method truncates a number.

Syntax

```
TRUNC(i)
```

Parameters

i – This the input number for which the value needs to be truncated.

Return Value

Returns the truncated number.

Example

```
/* Main program */
say TRUNC(12.22)
```

Output

When we run the above program, we will get the following result.

```
12
```

13. Rexx – Strings

Strings in Rexx are denoted by a sequence of characters. The following program is an example of strings:

```
/* Main program */
a = "This is a string"
say a
```

The output of the above program is as follows:

```
This is a string
```

Let's discuss some methods which are available in Rexx for strings.

left

This method returns a certain number of characters from the left of the string.

Syntax

```
left(str,count)
```

Parameters

- **str –** The source string.
- **count –** The number of characters to return from the left of the string.

Return Value

This method returns a certain number of characters from the left of the string.

Example

```
/* Main program */
a="Hello World"
say left(a,5)
```

Output

When we run the above program we will get the following result.

```
Hello
```

right

This method returns a certain number of characters from the right of the string.

Syntax

```
right(str,count)
```

Parameters

- **str –** The source string
- **count –** The number of characters to return from the right of the string.

Return Value

This method returns a certain number of characters from the right of the string.

Example

```
/* Main program */
a="Hello World"
say right(a,5)
```

Output

When we run the above program we will get the following result.

```
World
```

length

This method returns the number of characters in the string.

Syntax

```
length(str)
```

Parameter

- **str –** The source string

Return Value

This method returns the number of characters in the string.

Example

```
/* Main program */
a="Hello World"
say length(a)
```

Output

When we run above program we will get the following result.

```
11
```

reverse

This method returns the characters in a reverse format.

Syntax

```
reverse(str)
```

Parameters

- **str –** The source string

Return Value

This method returns the characters in a reverse format.

Example

```
/* Main program */
a="Hello World"
say reverse(a)
```

Output

When we run the above program we will get the following result

```
dlroW olleH
```

compare

This method compares 2 strings. Returns "0" if "string1" and "string2" are identical. Otherwise, it returns the position of the first character that does not match.

Syntax

```
compare(string1,string2)
```

Parameters

- **string1 –** The source string
- **string2 –** the string to compare against

Return Value

Returns "0" if "string1" and "string2" are identical otherwise, returns the position of the first character that does not match.

Example

```
/* Main program */
a="Hello World"
b="HELLO World"
say compare(a,b)
```

Output

When we run the above program we will get the following result.

```
2
```

copies

This method copies a **string n** number of times.

Syntax

```
copies(string1,count)
```

Parameters

- **string1 –** The source string
- **count –** The number of times to copy the string.

Return Value

Returns the string, copied count number of times.

Example

```
/* Main program */
a="Hello"
say copies(a,3)
```

Output

When we run the above program we will get the following result.

```
HelloHelloHello
```

substr

This method gets a substring from a particular string

Syntax

```
substr(string1,index,count)
```

Parameters

- **string1 –** The source string
- **index –** The starting index position for the substring
- **count –** The number of characters for the sub string

Return Value

Returns the sub-string.

Example

```
/* Main program */
a="Hello World"
say substr(a,2,3)
```

Output

When we run the above program, we will get the following result.

```
ell
```

pos

This method returns the position of one string within another.

Syntax

```
pos(substr,string1)
```

Parameters

- **substr –** the sub string to search for in the main string.
- **string1 –** The source string

Return Value

This method returns the position of one string within another.

Example

```
/* Main program */
a="Hello World"
say pos("Hello",a)
```

Output

When we run the above program, we will get the following result.

```
1
```

delstr

This method deletes a substring from within a string.

Syntax

```
delstr(string1,count)
```

Parameters

- **string1 –** The source string
- **count –** The count of characters to keep in the string.

Return Value

This method returns the string after deletion of the characters post the count of characters.

Example

```
/* Main program */
a="Hello World"
say delstr(a,6)
```

Output

When we run the above program, we will get the following result.

```
Hello
```

14. Rexx – Functions

The code in Rexx is normally divided into Functions and Subroutines. Using functions helps in segregating the code into many more logical units. Let's look at these functions in detail

Defining a Function

The syntax of a function declaration is as follows:

```
FunctionName:
PARSE ARG arguement1, arguement2… arguementN
Return value
```

Where,

- **FunctionName –** This is the name assigned to the function
- **PARSE ARG –** These are keywords in Rexx which are used to mention that parameters are being passed onto the function.
- **arguement1, arguement2... arguementN –** These are the arguments passed to the function.
- **Return value –** This is the value returned by the function.

The following program is a simple example of how functions are used in Rexx.

```
/* Main program */
say add(5,6)
exit
add:
PARSE ARG a,b
return a+b
```

The following things should be noted about the above program:

- We are defining a function called add which accepts 2 parameters a and b.
- The function uses the return statement to return the sum of a and b.
- The exit statement has to be used to signify the end of the main program.

The output of the above program would be as follows:

```
11
```

Working with Arguments

In Rexx, there are specific functions which can be made to work with arguments. Let's look at a couple of such arguments.

arg

This method is used to return the number of arguments defined for the function.

Syntax:

```
arg()
```

Parameters: None

Return Value: This method returns the number of arguments defined for the function.

Example:

```
/* Main program */
say add(5,6)
exit
add:
PARSE ARG a,b
say arg()
return a+b
```

Output: When we run the above program we will get the following result

```
2
11
```

arg(index)

This method is used to return the value of the argument at the specific position.

Syntax:

```
arg(index)
```

Parameter:

- **Index -** Index position of the argument to be returned.

Return Value: This method returns the value of the argument at the specific position.

Example:

```
/* Main program */
say add(5,6)
exit
add:
PARSE ARG a,b
say arg(1)
return a+b
```

Output: When we run the above program we will get the following result.

```
5
11
```

Recursive Functions

A recursive function or routine is one that calls itself. Any recursive function could be coded in a traditional non-recursive fashion (or iteratively), but sometimes recursion offers a better problem solution. Not all programming languages support recursion; Rexx does.

Let's see an example of the famous factorial program using recursive functions in Rexx.

```
/* Main program */
do n=1 to 5
say 'The factorial of' n 'is:' factorial( n )
end
return

/* Function to get factorial */
factorial : procedure
n = arg(1)
if n = 1 then
return 1
return n * factorial( n - 1 )
```

The output of the above program is as follows:

```
The factorial of 1 is: 1
The factorial of 2 is: 2
The factorial of 3 is: 6
The factorial of 4 is: 24
The factorial of 5 is: 120
```

15. Rexx – Stacks

The stack is sometimes called the external data queue, but we follow common usage and refer to it as the stack. It is a block of memory that is logically external to Rexx. Instructions like push and queue place data into the stack, and instructions like pull and parse pull extract data from it. The queued built-in function reports how many items are in the stack.

Let’s take a look at an example of a stack.

```
/* STACK: */
/* */
/* This program shows how to use the Rexx Stack as either a */
/* stack or a queue. */
do j=1 to 3
push ‘Stack: line #’ || j /* push 3 lines onto the stack */
end
do j=1 to queued() /* retrieve and display LIFO */
pull line
say line
end
do j=1 to 3
queue ‘Queue: line #’ || j /* queue 3 lines onto the stack */
end
do queued() /* retrieve and display FIFO */
pull line
say line
end
exit 0
```

The first do loop in the program places three lines of data onto the stack. It uses the push instruction to do this. We number the lines so that when they are retrieved in the LIFO order their order is apparent.

The items placed into the stack by the push instruction are retrieved in the LIFO order:

```
do j=1 to 3

push ‘Stack: line #’ || j /* push 3 lines onto the stack */

end
```

The next code block shows the use of the queued built-in function to discover the number of lines on the stack, as well as a loop to retrieve all the lines from the stack:

```
do j=1 to queued()       /* retrieve and display LIFO */

pull line

say line

end
```

Since the three items were placed on the stack via push, they are retrieved in the LIFO order.

The output of the above program will be as follows.

```
STACK: LINE #3
STACK: LINE #2
STACK: LINE #1
```

16. Rexx – File I/O

Rexx provides a number of methods when working with I/O. Rexx provides easier classes to provide the following functionalities for files.

- Reading files
- Writing to files
- Seeing whether a file is a file or directory

The functions available in Rexx for File I/O are based on both line input and character input and we will be looking at the functions available for both in detail.

Let's explore some of the file operations Rexx has to offer. For the purposes of these examples, we are going to assume that there is a file called **NewFile.txt** which contains the following lines of text:

Example1

Example2

Example3

This file will be used for the read and write operations in the following examples. Here we will discuss regarding how to read the contents on a file in different ways.

Reading the Contents of a File a Line at a Time

The general operations on files are carried out by using the methods available in the Rexx library itself. The reading of files is the simplest of all operations in Rexx.

Let's look at the function used to accomplish this.

linein

This method returns a line from the text file. The text file is the filename provided as the input parameter to the function.

Syntax:

```
linein(filename)
```

Parameter:

- **filename –** This is the name of the file from where the line needs to be read.

Return Value: This method returns one line of the file at a time.

Example:

```
/* Main program */
line_str = linein(Example.txt)
say line_str
```

The above code is pretty simple in the fact that the **Example.txt** file name is provided to the linein function. This function then reads a line of text and provides the result to the variable **line_str**.

Output: When we run the above program we will get the following result.

```
Example1
```

Reading the Contents of a File at One Time

In Rexx, reading all the contents of a file can be achieved with the help of the while statement. The while statement will read each line, one by one till the end of the file is reached.

An example on how this can be achieved is shown below.

```
/* Main program */
do while lines(Example.txt) > 0

line_str = linein(Example.txt)
say line_str
end
```

In the above program, the following things need to be noted:

- The lines function reads the **Example.txt** file.
- The while function is used to check if further lines exist in the Example.txt file.
- For each line read from the file, the **line_str** variable holds the value of the current line. This is then sent to the console as output.

Output: When we run the above program we will get the following result.

```
Example1
Example2
Example3
```

Writing Contents to a File

Just like reading of files, Rexx also has the ability to write to files. Let’s look at the function which is used to accomplish this.

lineout

This method writes a line to a file. The file to which the line needs to be written to is provided as the parameter to the lineout statement.

Syntax:

```
lineout(filename)
```

Parameter:

- **filename –** This is the name of the file from where the line needs to be written to.

Return Value: This method returns the status of the lineout function. The value returned is 0 if the line was successfully written else the value of 1 will be returned.

Example:

```
/* Main program */
out=lineout(Example.txt,"Example4")
```

Output: Whenever the above code is run, the line “Example4” will be written to the file **Example.txt**.

17. Rexx – Functions for Files

In this chapter, we will discuss regarding some of the other functions that are available for files.

Lines

This function returns either the value 1 or the number of lines left to read in an input stream. The filename is given as the input to the function.

Syntax:

```
lines(filename)
```

Parameter:

- **filename –** This is the name of the file.

Return Value: This function returns either the value 1 or the number of lines left to read in an input stream.

Example:

```
/* Main program */
do while lines(Example.txt) > 0
line_str = linein(Example.txt)
say line_str
end
```

In the above program the following things need to be noted.

- The lines function reads the Example.txt file.
- The while function is used to check if further lines exist in the Example.txt file.
- For each line read from the file, the line_str variable holds the value of the current line. This is then sent to the console as output.

Output: When we run the above program we will get the following result.

```
Example1
Example2
Example3
```

There is another variation of the lines command which is as follows:

Syntax:

```
lines(filename,C)
```

Parameters:

- **filename –** This is the name of the file.
- **C –** This is a constant value provided to the function. This value which specifies the number of lines left to read from the file.

Return Value: The return value is the count of lines that are left to be read from the file.

Example:

```
/* Main program */
count=lines(Example.txt,C)
say count
line_str = linein(Example.txt)
say line_str
count=lines(Example.txt,C)
say count
```

Output: When we run the above program we will get the following result.

```
3
Example1
2
```

stream

This function is used to check the status of a file. Sometimes it is required to check the status of a file before it is used. If the file is corrupt or not available, then no further operations can be performed on the file. So it makes more sense to first check the status of the file.

Syntax:

```
stream(filename)
```

Parameter:

- **filename –** This is the name of the file.

Return Value: This function can return any of the following values:

- **READY –** The file is ready and can be used for any subsequent operations.

- **NOTREADY –** The file is not ready. There could be cases wherein the file is being used by another operation and waiting for waiting for an input/output operation to occur. In such cases, the file might have this status.
- **ERROR –** There could be cases wherein the file is being used by another operation and the file could be in an error state. In such cases, the file might have this (ERROR) status.
- **UNKNOWN –** This is returned wherein Rexx cannot understand what is the current status of the file.

Example:

```
/* Main program */
status=stream(Example.txt)
say status
```

Output: When we run the above program we will get the following result.

```
READY
```

charin

This function is used to read one character at a time from a file. Sometimes programs require to read files character wise and hence this function can be used for this purpose.

Syntax:

```
charin(filename)
```

Parameters:

- **filename –** This is the name of the file from which the character needs to be read from.

Return Value: A character read from the file.

Example:

```
/* Main program */
chr=charin(Example.txt)
say chr
chr=charin(Example.txt)
say chr
chr=charin(Example.txt)
say chr
```

Output: When we run the above program, we will get the following result.

```
E
x
a
```

chars

This function returns either 1 or the number of characters left to read in the file itself. The filename is mentioned as a parameter to the function.

Syntax:

```
chars(filename)
```

Parameter:

- **filename –** This is the name of the file for which the end of file needs to be determined.

Return Value: This function returns either 1 or the number of characters left to read in the file itself.

Example:

```
/* Main program */
str = ''

do j=1 while chars(Example.txt) > 0
str = ' ' (charin(Example.txt))
call charout ,str
end
```

The following things need to be noted about the above program:

- First a string variable is assigned to a null value.
- Then the 'do loop' is used to read each character at a time.
- Each character is read and then sent to the console.

Output: When we run above program we will get the following result.

```
E x a m p l e 1
E x a m p l e 2
E x a m p l e 3
```

charout

This function is used to write one character at a time to a file. The filename is entered as a parameter to the function.

Syntax:

```
charout(filename)
```

Parameter:

- **filename –** This is the name to which the character needs to be written to.

Return Value: This function returns 0 if all characters were successfully written to the file. If the function fails, then it returns the number of characters that failed to write to the file.

Example:

```
/* Main program */
status=charout(Example.txt,E)
status=charout(Example.txt,x)
status=charout(Example.txt,a)
```

Output: When we run the above program, the characters 'E', 'X' and 'a' will be added to the file Example.txt.

18. Rexx – Subroutines

In any programming language, the entire program is broken into logical modules. This makes it easier to write code that can be maintained easily. This is a basic requirement for any programming language.

In Rexx, modules can be written using Subroutines and functions. Let's look at the subroutines in detail.

Defining a Subroutine

The syntax of a function declaration is as follows:

```
FunctionName:
Statement#1
Statement#2
….
Statement#N
```

Where,

- **FunctionName –** This is the name assigned to the subroutine.
- **Statement#1 .. Statement#N –** These are the list of statements that make up the subroutine.

The following program is a simple example showing the use of subroutines.

```
/* Main program */
call add
exit
add:
a=5
b=10
c=a+b
say c
```

The following things should be noted about the above program:

- We are defining a subroutine called **add**.
- The subroutine does a simple add functionality.
- The exit statement has to be used to signify the end of the main program.

The output of the above program would be as follows:

```
15
```

Working with Arguments

It is also possible to work with arguments in Rexx. The following example shows how this can be achieved.

```
/* Main program */
call add 1,2
exit
add:
PARSE ARG a,b
c=a+b
say c
```

The following things should be noted about the above program:

- We are defining a subroutine called add which takes on 2 parameters.
- In the subroutines, the 2 parameters are parsed using the PARSE and ARG keyword.

The output of the above program would be as follows:

```
3
```

Different Methods for Arguments

Let's look at some other methods available for arguments.

arg

This method is used to return the number of arguments defined for the subroutine.

Syntax:

```
arg()
```

Parameters: None

Return Value: This method returns the number of arguments defined for the subroutine.

Example:

```
/* Main program */
call add 1,2
exit
add:
PARSE ARG a,b
say arg()
c=a+b
say c
```

Output: When we run the above program we will get the following result.

```
2
3
```

arg(index)

This method is used to return the value of the argument at the specific position.

Syntax:

```
arg(index)
```

Parameter

- **Index -** Index position of the argument to be returned.

Return Value: This method returns the value of the argument at the specific position.

Example:

```
/* Main program */
call add 1,2
exit
add:
PARSE ARG a,b
say arg(1)
c=a+b
say c
```

Output:

When we run the above program we will get the following result.

```
1
3
```

19. Rexx – Built-in Functions

Every programming language has some built-in functions that help the programmer do some routine tasks. Rexx also has a lot of built in functions.

Let's look at all of these functions available in Rexx.

ADDRESS

This method returns the name of the environment in the which the Rexx commands are currently running in.

Syntax:

```
ADDRESS()
```

Parameters: None

Return Value: This method returns the name of the environment in the which the Rexx commands are currently running in.

Example:

```
/* Main program */
say ADDRESS()
```

Output: When we run the above program we will get the following result.

```
SYSTEM
```

BEEP

This method produces a sound in the system at a particular frequency and duration.

Syntax:

```
BEEP(Frequency,duration)
```

Parameters:

- **Frequency -** The frequency can be any whole number in the range of 37 to 32767 Hertz.
- **Duration –** The duration will be in milliseconds.

Return Value:

None – A beep sound will be triggered in the system.

Example:

```
/* Main program */
BEEP(1000,200)
```

Output: A beep sound will be triggered in the system.

DataType

This method returns the value of 'NUM' if the input is a valid number else it will return the value of 'CHAR'. You can also specify if you want to compare the input value to a NUM or CHAR value. In each case, the value returned will be either 1 or 0 depending on the result.

Syntax:

```
DATATYPE(String,type)
```

Parameters:

- **String –** The string value for which the datatype needs to be determined.
- **Type –** Optional type against which the datatype need to be compared to.

Return Value: This method returns the value of 'NUM' if the input is a valid number else it will return the value of 'CHAR'. You can also specify if you want to compare the input value to a NUM or CHAR value. In each case, the value returned will be either 1 or 0 depending on the result.

Example:

```
/* Main program */
say DATATYPE(" 12345 ")
say DATATYPE("")
say DATATYPE("12345*")
say DATATYPE("123.4","N")
say DATATYPE("123.4","W")
```

Output: When we run the above program we will get the following result.

```
NUM
CHAR
CHAR
1
0
```

DATE

This method returns the local date in the following format.

dd mon yyyy

Syntax:

```
DATE(options)
```

Parameters:

- **Options –** This is the option for formatting the date value. They can be anyone of the following options given.
    - **Base -** This parameter returns the number of complete days since the day 1 January 0001.
    - **Days -** This parameter returns the number of days that have passed for the current year.
    - **European -** This parameter returns the date in the format dd/mm/yy.
    - **Full -** This parameter returns the number of microseconds since the day 1 January 0001.
    - **Month -** This parameter returns the full English name of the current month. An example can be, September.
    - **Normal -** This parameter returns the date in the normal format as shown below. This is the default parameter option which is dd mon yyyy.
    - **Ordered -** This parameter returns the date in the format – yy/mm/dd.
    - **Standard –** This parameter returns the date in the format – yyyymmdd.
    - **Weekday -** This parameter returns the English name for the day of the week. An example can be, Wednesday.

Return Value: Returns, by default, the local date in the format: dd mon yyyy.

Example:

```
/* Main program */
say DATE()
say DATE("B")
say DATE("D")
say DATE("E")
```

Output: When we run the above program we will get the following result. This depends on the current date on the system.

The following program is just an example.

```
2 Jun 2016

736116

154

02/06/16
```

DIGITS

This method returns the current setting of NUMERIC DIGITS as defined in the current system.

Syntax:

```
DIGITS()
```

Parameters: None

Return Value: This method returns the current setting of NUMERIC DIGITS as defined in the current system.

Example:

```
/* Main program */

say DIGITS()
```

Output: When we run the above program we will get the following result.

```
9
```

ERRORTEXT

This method returns the Rexx error message associated with error number 'errorno'. Please note that the error number needs to be a value from 0 to 99. This is useful in cases wherein your program returned an error code and you want to know what the error code means.

Syntax:

```
ERROTEXT(errorno)
```

Parameter:

- **errorno –** This is the error number for which the error message needs to be determined.

Return Value: This method returns the Rexx error message associated with error number errorno. Please note that the error number needs to be a value from 0 to 99.

Example:

```
/* Main program */
say ERRORTEXT(16)
```

Output: When we run the above program we will get the following result. The below error text is the message defined for the Rexx error with error number 16.

```
Label not found
```

FORM

This method returns the current setting of 'NUMERIC FORM' which is used to do mathematic calculations on the system.

Syntax

```
FORM()
```

Parameters: None

Return Value: This method returns the current setting of 'NUMERIC FORM' which is used to do mathematic calculations on the system.

Example:

```
/* Main program */
say FORM()
```

Output: When we run the above program we will get the following result.

```
SCIENTIFIC
```

TIME

This method returns the local time in the 24-hour clock format as shown in the following program.

```
hh:mm:ss (hours, minutes, and seconds)
```

Syntax:

```
TIME(options)
```

Parameters:

- Options – This is the options for formatting the time value. They can be anyone of the following options:
  - **Civil –** This parameter returns the time in the format () wherein hh is hours, mm is minutes and xx is seconds.
  - **Elapsed -** This parameter returns the number of seconds and microseconds since the elapsed-time clock was started or reset.
  - **Full –** This parameter returns the number of microseconds since the date 1 January 0001.
  - **Hours -** This parameter returns the number of hours that have passed since midnight.
  - **Minutes -** This parameter returns the number of minutes that have passed since midnight.
  - **Normal -** This parameter returns the time in the default format hh:mm:ss.
  - **Offset -** This parameter returns the offset of the local time from UTC in microseconds.

Return Value: This method returns the local time in the 24-hour clock format – hh:mm:ss (hours, minutes, and seconds)

Example:

```
/* Main program */
say TIME()
say TIME("C")
say TIME("H")
```

Output: When we run the above program we will get the following result. This depends on the current time on the system.

The following program is just an example.

```
10:54:12
10:54am
10
```

USERID

This method returns the current user id logged into the system.

Syntax

```
USERID()
```

Parameters: None

Return Value: This method returns the current user id logged into the system.

Example:

```
/* Main program */
say USERID()
```

Output: When we run the above program we will get the following result. This depends on the current logged on user of the system.

The following program is only a sample output.

```
Administrator
```

XRANGE

This method returns the characters in the range specified by the start and end character.

Syntax:

```
XRANGE(start,end)
```

Parameters:

- **start –** start the hexadecimal value
- **end –** end the hexadecimal value

Return Value: This method returns the characters in the range specified by the start and end character.

Example:

```
/* Main program */
say XRANGE("a","d")
say XRANGE("i","k")
```

Output: When we run the above program, we will get the following result.

```
abcd
ijk
```

X2D

This method returns the decimal conversion of a **hexstring value**.

Syntax:

```
X2D(value)
```

Parameter:

- **value –** This is the hexadecimal value which needs to be converted to a decimal value.

Return Value: This method returns the decimal conversion of a hexstring value.

Example:

```
/* Main program */
say X2D("0E")
say X2D("725")
say X2D("F725")
say X2D("FF725")
```

Output: When we run the above program we will get the following result.

```
14
1829
63269
1046309
```

X2C

This method returns the character conversion of a hexstring value.

Syntax:

```
X2C(value)
```

Parameters:

- **value –** This is the hexadecimal value which needs to be converted to a string value.

Return Value: This method returns the character conversion of a hexstring value.

Example:

```
/* Main program */
say X2C("465 6c 6f")
say X2C("37 73")
```

Output: When we run the above program we will get the following result.

```
elo

7s
```

20. Rexx – System Commands

One of the biggest advantages in Rexx is the ability to create re-usable scripts. Often in organizations nowadays, having re-usable scripts is a big value add in saving time to do common repetitive tasks.

For example, technology teams in an IT organization can have the need to have scripts which do common everyday tasks. These tasks can include interacting with the operating systems. These scripts can then be programmed to handle bad return codes or errors.

Rexx offers a lot of system commands that can be used to perform such repetitive tasks. Let’s look at some of the system commands available in Rexx.

dir

This is the normal directory listing command which is used in Windows.

Syntax:

```
dir
```

Parameters: None

Return Value: This method returns the current directory listing on the system.

Example:

```
/* Main program */
dir
```

Output: The output depends on the directory in the system.

The following program is just an example.

```
Volume in drive D is LENOVO
Volume Serial Number is BAC9-9E3F


Directory of D:\
04/06/2016  12:52 AM          268,205 100008676689.pdf
10/20/2015  08:51 PM    <DIR>          data
06/01/2016  10:23 AM               31 Example.txt
10/28/2014  06:55 PM    <DIR>          Intel
06/02/2016  11:15 AM               23 main.rexx
12/22/2014  08:49 AM    <DIR>          PerfLogs
```

```
12/13/2015  11:45 PM    <DIR>          Program Files
12/24/2015  10:26 AM    <DIR>          Program Files (x86)
07/17/2015  01:21 AM    <DIR>          Users
12/23/2015  10:01 AM    <DIR>          Windows
               3 File(s)        268,259 bytes
               7 Dir(s)     202,567,680 bytes free
```

Another example of the **dir command** is shown in the following program. Only this time we are making use of the **special rc variable**. This variable is special in Rexx and gives you the status of the execution of system commands. If the value returned is 0, then that means the command is executed successfully. Else the error number will be given in the rc variable name.

Example:

```
/* Main program */
dir
if rc = 0 then
say 'The command executed successfully'
else
say 'The command failed, The error code is =' rc
```

Output: When we run the above program we will get the following result.

```
The command executed successfully.
```

Redirection Commands

Rexx also has the facility of using redirection commands. The following redirection commands are available in Rexx.

- **< -** This command is used to take in the input which comes from a file.
- **> -** This command is used to output the content to a file. If the file does exist, it will be over-written.
- **>> -** This is also used to output the content to a file. But the output is added to the end of the file to preserve the existing content of the file.

Let's look at an example of how we can use redirection commands. In the following example, we are using the sort command to sort a file called **sortin.txt**. The data from the file is sent to the sort command. The output of the sort command is then sent to the sortout.txt file.

Example:

```
/* Main program */
'sort <sortin.txt >sortout.txt'
```

Output: Assume that the file sortin.txt has the following data.

```
b
c
a
```

The file **sortout.txt** will then have the following data.

```
a
b
c
```

The ADDRESS Function

This method is used to find out what is the default environment used for the Input, Error and Output streams.

Syntax:

```
ADDRESS(options)
```

Parameter:

- Options for what is the address of a particular system.

Return Value: This method returns the name of the environment for the Input, Error and Output streams.

Example:

```
/* Main program */
say ADDRESS('I')
say ADDRESS('O')
say ADDRESS('E')
```

Output: When we run the above program we will get the following result.

```
INPUT NORMAL
REPLACE NORMAL
REPLACE NORMAL
```

21. Rexx – XML

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of the operating system and/or developmental language. This is one of the most common languages used for exchanging data between applications.

What is XML?

The Extensible Markup Language XML is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard. XML is extremely useful for keeping track of small to medium amounts of data without requiring a SQL-based backbone.

For all our XML code examples, let's use the following simple XML file **movies.xml** for construction of the XML file and reading the file subsequently.

```
<collection shelf="New Arrivals">

<movie title="Enemy Behind">

<type>War, Thriller</type>

<format>DVD</format>

<year>2003</year>

<rating>PG</rating>

<stars>10</stars>

<description>Talk about a US-Japan war</description>

</movie>

<movie title="Transformers">

<type>Anime, Science Fiction</type>

<format>DVD</format>

<year>1989</year>

<rating>R</rating>

<stars>8</stars>

<description>A schientific fiction</description>

</movie>

<movie title="Trigun">

<type>Anime, Action</type>

<format>DVD</format>

<year>1986</year>

<rating>PG</rating>
```

```
<stars>10</stars>
<description>Vash the Stam pede!</description>
</movie>
<movie title="Ishtar">
<type>Comedy</type>
<format>VHS</format>
<year>1987</year>
<rating>PG</rating>
<stars>2</stars>
<description>Viewable boredom </description>
</movie>
</collection>
```

Getting Started

By default, the xml functionality is not included in the Rexx interpreter. In order to work with XML in Rexx, the following steps need to be followed.

- Download the following files –
    - Rexxxml – http://www.interlog.com/~ptjm/
    - Libxml2 - http://ctindustries.net/libxml/
    - iconv-1.9.2.win32 - http://xmlsoft.org/sources/win32/oldreleases/
    - libxslt-1.1.26.win32 - http://xmlsoft.org/sources/win32/oldreleases/
- Extract all of the files and ensure they are included in the system path.

Loading XML Functions

Once all the files in the above section have been downloaded and successfully registered, the next step is to write the code to load the Rexx XML functions. This is done with the following code.

```
rcc = rxfuncadd('XMLLoadFuncs', 'rexxxml', 'xmlloadfuncs')
if rcc then do
say rxfuncerrmsg()
exit 1
end
call xmlloadfuncs
```

The following things can be noted about the above program:

- The function **rxfuncadd** is used to load external libraries. The **xmlloadfuncs** function is used to load all the libraries in the **rexxxml** file into memory.
- If the value of rcc<>0, then it would result in an error. For this , we can call the **rxfuncerrmsg** to give us more details on the error message.
- We finally make a call to **xmlloadfuncs**, so that all xml related functionality can now be enabled in the Rexx program.

Let's look at the various **methods available for XML in Rexx**.

xmlVersion

This method returns the version of the XML and XSLT libraries used on the system.

Syntax:

```
xmlVersion()
```

Parameters: None

Return Value: This method returns the version of the XML and XSLT libraries used on the system.

Example:

```
rcc = rxfuncadd('XMLLoadFuncs', 'rexxxml', 'xmlloadfuncs')
if rcc then do
say rxfuncerrmsg()
exit 1
end
call xmlloadfuncs
say xmlVersion()
```

Output: When we run above program we will get the following result. This again depends on the version of the XML libraries being used on the system.

```
1.0.0 20631 10126
```

xmlParseXML

This function is used to parse the XML data sent to the function. The document tree is returned by the function.

Syntax:

```
xmlParseXML(filename)
```

Parameters:

- **Filename –** This is the name of the XML file which needs to be parsed.

Return Value:

The document tree is returned by the function. Else returns 0, if there is an error.

Example:

```
rcc = rxfuncadd('XMLLoadFuncs', 'rexxxml', 'xmlloadfuncs')
if rcc then do
say rxfuncerrmsg()
exit 1
end
call xmlloadfuncs
say xmlVersion()
sw = xmlParseXML('test.xml')
```

Output: No general output.

xmlFindNode

This method evaluates the **XPath expression** passed to it. This is used for parsing the document tree to result a **nodeset** which can be processed further.

Syntax:

```
xmlParseXML(XPath,document)
```

Parameters:

- **XPath –** This is the path of the node in the xml file.
- **document –** This the XML document

Return Value: Evaluates XPath expression and returns result as a nodeset which can be used later on.

Example:

```
rcc = rxfuncadd('XMLLoadFuncs', 'rexxxml', 'xmlloadfuncs')
if rcc then do
say rxfuncerrmsg()
exit 1
end
call xmlloadfuncs
```

```
say xmlVersion()
document = xmlParseXML('test.xml')
nodeset = xmlFindNode('//movie', document)
say xmlNodesetCount(nodeset)
```

Output: When we run above program we will get the following result.

```
4
```

The output shows the number of movie nodes in our xml list

xmlEvalExpression

The below method is used to Evaluate an XPath expression and return a string as a result.

Syntax:

```
xmlParseXML(XPath,Node)
```

Parameters:

- **XPath –** This is the path of the node in the xml file.
- **document –** The specific node element

Return Value: A string is returned based on the XPath expression sent to it.

Example:

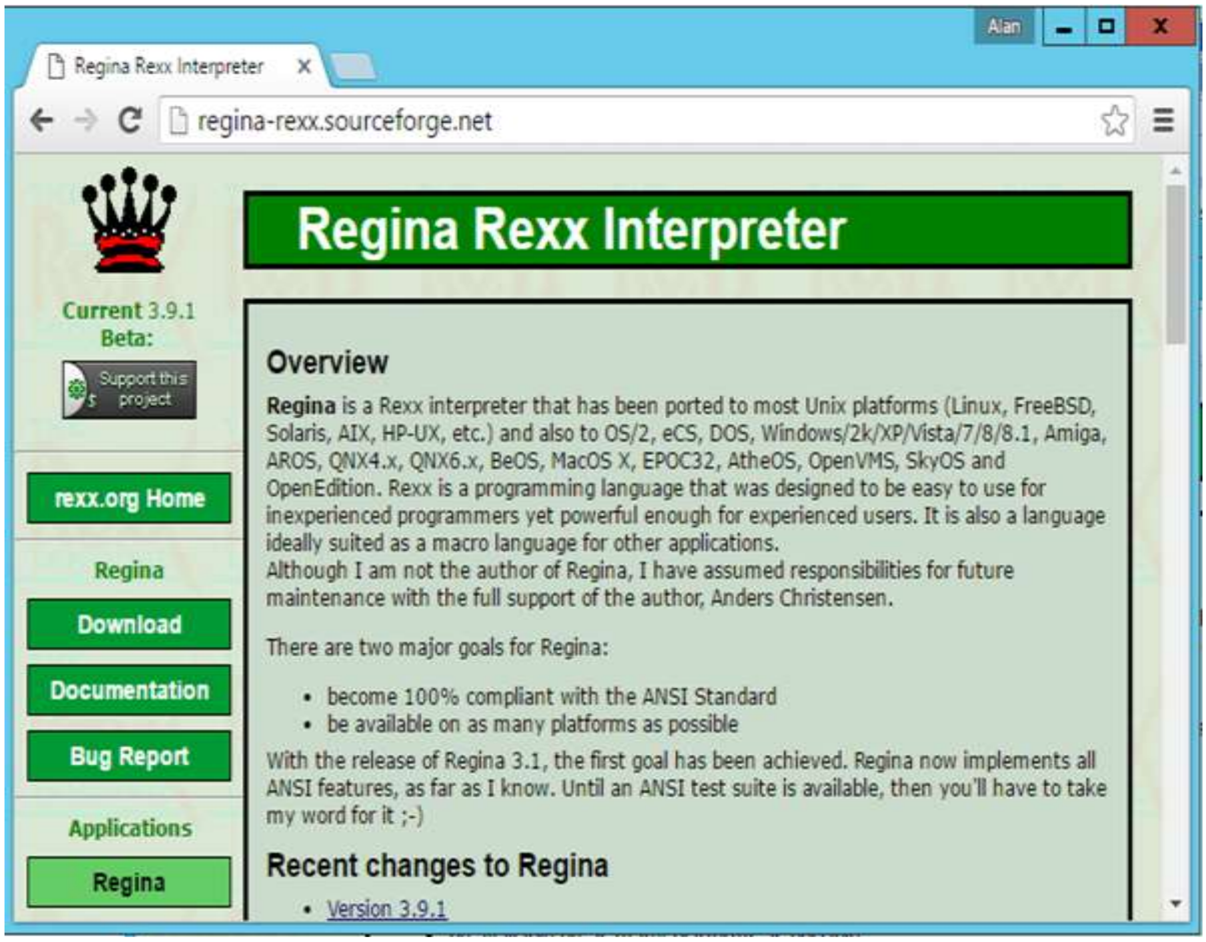
```
rcc = rxfuncadd('XMLLoadFuncs', 'rexxxml', 'xmlloadfuncs')
if rcc then do
say rxfuncerrmsg()
exit 1
end
call xmlloadfuncs
document = xmlParseXML('test.xml')
nodeset = xmlFindNode('//movie', document)
do j = 1 to xmlNodesetCount(nodeset)

value = xmlEvalExpression('type', xmlNodesetItem(nodeset, j))
say value
end
```

Output: When we run above program we will get the following result.

```
War, Thriller

Anime, Science Fiction

Anime, Action

Comedy
```

22. Rexx – Regina

Regina is another Rexx interpreter available to compile and run Rexx programs. The official site for Regina is – http://regina-rexx.sourceforge.net/



Regina Rexx Interpreter

Overview

Regina is a Rexx interpreter that has been ported to most Unix platforms (Linux, FreeBSD, Solaris, AIX, HP-UX, etc.) and also to OS/2, eCS, DOS, Windows/2k/XP/Vista/7/8/8.1, Amiga, AROS, QNX4.x, QNX6.x, BeOS, MacOS X, EPOC32, AtheOS, OpenVMS, SkyOS and OpenEdition. Rexx is a programming language that was designed to be easy to use for inexperienced programmers yet powerful enough for experienced users. It is also a language ideally suited as a macro language for other applications.
Although I am not the author of Regina, I have assumed responsibilities for future maintenance with the full support of the author, Anders Christensen.

There are two major goals for Regina:

- become 100% compliant with the ANSI Standard
- be available on as many platforms as possible

With the release of Regina 3.1, the first goal has been achieved. Regina now implements all ANSI features, as far as I know. Until an ANSI test suite is available, then you'll have to take my word for it ;-)

Recent changes to Regina

- Version 3.9.1

Some of the advantages of using Regina are as follows:

- Regina can run on any platform whether it be Windows, Linux or the Mac OS.
- Regina works as per all available standards.
- Regina has a big community following and hence there are a lot of forums and learning material available for Regina.
- Regina has a lot of tools available for writing and testing Rexx programs.
- In Regina, you can run commands which are not possible in the default Rexx Interpreter. For example, if you include certain configuration settings, you can actually run basic system level commands, which is not possible in Rexx.

When you install Rexx via the installation documented in **Chapter 2 – Rexx Environment**, the Regina interpreter gets installed along with it.

Now let's see some of the common methods available when using Regina. These functions are the extended functions which are not available via normal use.

To make use of the extended functions, you need to include the following line of code. This enables the use of Regina extended functions.

```
options arexx_bifs
```

Secondly while running all Rexx programs, use the following command.

```
regina main.rexx
```

Where,

- **regina –** This is the interpreter used for Rexx programs.
- **main.rexx –** Your Rexx program

We will now discuss in detail the various **functions of Regina Rexx Interpreter**.

b2c

This method is used to convert a binary value to a string value.

Syntax:

```
b2c(binstring)
```

Parameter:

- **binstring –** The binary string value.

Return Value: This method is used to convert a binary value to a string value.

Example:

```
/* Main program */
options arexx_bifs
say b2c('00110011')
```

Output: When we run the above program we will get the following result.

```
3
```

bitcomp

The method is used to compare 2 bit strings, bit by bit.

Syntax:

```
bitcomp(binstring1, binstring2)
```

Parameters:

- **binstring1, binstring2 –** The binary string value that needs to be compared.

Return Value: The returned value is the bit number of the first bit which is different in both strings.

Example:

```
/* Main program */
options arexx_bifs
say bitcomp('7F'x,'FF'x)
```

Output: When we run the above program we will get the following result.

```
7
```

bittst

This method is used to indicate the state of the specified bit in the bit string.

Syntax:

```
bitcomp(binstring, bit)
```

Parameters:

- **binstring –** The binary string value.
- **bit –** The bit value that needs to be tested.

Return Value: The Boolean return indicates the state of the specified bit in the argument string.

Example:

```
/* Main program */
options arexx_bifs
say bittst('0313'x,4)
```

Output: When we run above program we will get the following result.

```
1
```

find

This method is used to search for the first occurrence of a string in another string.

Syntax:

```
find(string, phrase)
```

Parameters:

- **string –** The string value that needs to be searched for.
- **phrase –** This is the value that needs to be searched for in the string.

Return Value: The method returns the word number of the first word of phrase in string.

Example:

```
/* Main program */
options arexx_bifs
say find(' Hello World ','World')
```

Output: When we run the above program we will get the following result.

```
2
```

getenv

This method returns the value of an environment variable on the system.

Syntax:

```
getenv(var)
```

Parameters:

- **var –** This the name of the environment variable for which the value needs to be displayed.

Return Value: This method returns the value of an environment variable on the system.

Example:

```
/* Main program */
options arexx_bifs
say getenv('PATH')
```

Output: When we run the above program we will get the following result. The following program is a sample output, this output will vary from system to system.

```
C:\ProgramData\Oracle\Java\javapath;C:\Python27\;C:\Python27\Scripts;C:\Program

Files (x86)\NVIDIA
Corporation\PhysX\Common;C:\Windows\system32;C:\Windows;C:\Wi

ndows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program
Files

\Intel\WiFi\bin\;C:\Program Files\Common Files\Intel\WirelessCommon\;C:\Program

Files\Microsoft SQL
Server\110\Tools\Binn\;C:\PROGRA~2\Groovy\GROOVY~1.5\bin;C:\

Program Files (x86)\nodejs\;C:\Program Files\MySQL\MySQL Server
5.7\bin;C:\Progr

am Files\MongoDB\Server\3.2\bin;C:\Program Files (x86)\Microsoft SQL
Server\110\

Tools\Binn\;C:\Program Files\Microsoft SQL Server\110\DTS\Binn\;C:\Program
Files

 (x86)\Microsoft SQL Server\110\Tools\Binn\ManagementStudio\;C:\Program Files
(x

86)\Microsoft SQL Server\110\DTS\Binn\;C:\Program
Files\TortoiseGit\bin;C:\Progr

am Files\Microsoft\Web Platform
Installer\;C:\.lein\bin;C:\Windows\Microsoft.NET

\Framework\v4.0.30319;C:\Program Files\Java\jdk1.7.0_79\bin;C:\Program
Files\Jav

a\jre7\bin;C:\Program
Files\Git\cmd;C:\Users\Administrator\AppData\Roaming\npm;C

:\Program Files\erl7.3\bin;C:\Program Files (x86)\Windows Kits\8.1\Windows
Perfo

rmance Toolkit\;C:\Program
Files\rexx.org\Regina;C:\RexxTrans;D:\rexxxml100\rexx

xml\win32;D:\libxslt-1.1.17.win32\libxslt-1.1.17.win32\bin;D:\libxml2-
2.6.26.win

32\libxml2-2.6.26.win32\bin;D:\iconv-1.9.2.win32\iconv-1.9.2.win32\bin;D:\zlib-
1

.2.3.win32\zlib-1.2.3.win32\bin;D:\rexxtrans18w32;C:\Program
Files\ooRexx;D:\rex

xxml100\rexxxml
```

getpid

This method is used to get the value of the current running process id.

Syntax:

```
getpid()
```

Parameters: None

Return Value: Gets the value of the current running process id.

Example:

```
/* Main program */
options arexx_bifs
say getpid()
```

Output: When we run the above program we will get the following result. The following program is a sample output, this output will vary from system to system.

```
6884
```

hash

This method returns the hash attribute of a string as a decimal number. It also updates the internal hash value of the string.

Syntax:

```
hash(string)
```

Parameters:

- **String –** The string value for which the hash value needs to be generated.

Return Value: Returns the hash attribute of a string as a decimal number and also updates the internal hash value of the string.

Example:

```
/* Main program */
options arexx_bifs
say hash('hello')
```

Output: When we run the above program we will get the following result.

```
20
```

justify

This method is used to add justify or trim the value of a string based on the length value.

Syntax:

```
justify(string,length)
```

Parameters:

- **String –** The string value for which the justification needs to be done.
- **Length –** The value for justification.

Return Value: Returns the justified string value.

Example:

```
/* Main program */
options arexx_bifs
say justify('This is a tutorial',14)
```

Output: When we run the above program we will get the following result.

```
This is a tuto
```

putenv

This method is used to set the value of an environment variable.

Syntax:

```
putenv('name=value')
```

Parameters:

- **name –** The name of the environment variable.
- **value –** The value of the environment variable.

Return Value: The existing value is returned if the environment variable has a value or if this environment variable is not defined, a **nullstring** is returned.

Example:

```
/* Main program */
options arexx_bifs
say putenv('JAVA_HOME=C:\')
```

Output: When we run the above program we will get the following result. The results depend from system to system.

```
C:\Program Files\Java\jre7
```

directory

This method gets the value of the current directory on the system.

Syntax:

```
directory()
```

Parameters: None

Return Value: This method gets the value of the current directory on the system.

Example:

```
/* Main program */
options arexx_bifs
say directory()
```

Output:

When we run the above program we will get the following result. The results depend from system to system

```
D:\
```

chdir

This method changes the value of the current working directory on the system.

Syntax:

```
chrdir(newdir)
```

Parameters:

- **newdir –** The new directory which should become the current working directory.

Return Value: Returns 0 if the change was successful.

Example:

```
/* Main program */
options arexx_bifs
say chdir('\rexxxml100')
say directory()
```

Output: When we run the above program we will get the following result.

```
0
D:\rexxxml100
```

randu

This method returns a pseudo-random number between 0 and 1.

Syntax:

```
randu()
```

Parameters: None

Return Value: Returns a pseudo-random number between 0 and 1.

Example:

```
/* Main program */
options arexx_bifs
say randu()
```

Output: When we run the above program we will get the following result.

```
0.396464774
```

23. Rexx – Parsing

One of the most powerful features of Rexx is its ability to parse text values. You probably will not see this in any other programming languages.

The general format of the parse statement is as follows:

Syntax:

```
PARSE {UPPER|LOWER|CASELESS} source {template}
```

Where,

- **UPPER -** The source is converted to upper case before parsing.
- **LOWER -** The source is converted to lower case before parsing.
- **CASELESS –** When this parameter is passed, the casing is ignored.
- **source –** This is the source which needs to be parsed. There are many options available for this and can be any one of the following:
    - **ARG –** The arguments for the program or procedure can be used as the source.
    - **LINEIN –** The next line input can be used as the source.
    - **SOURCE –** The source information of the program can be used as the source.
    - **VAR name –** The value of a variable name can be used as the source.
- **template -** This parameter specifies how to parse the source. There are many options available for this. Some of them are mentioned below.
    - **variable name –** This is the value assigned to the variable.
    - **literal string -** A literal string which can be used a pattern to split the strung.
    - **# -** An absolute character position within the source itself. So if you specify a value of 5, the 5th character will be used.
    - **+# -** A relative character position within the source itself. So if you specify a value of 5, the 5th character will be used relatively.

Let’s look at a simple example of how parsing can be accomplished in Rexx.

```
/* Main program */
parse value 'This is a Tutorial' with word1 word2 word3 word4
say "'"word1"'"
say "'"word2"'"
say "'"word3"'"
say "'"word4"'"
```

The above program parses the words in the phrase. When a value consists of words that are separated by only one space, and there are no leading or trailing spaces, the value is easy to parse into a known number of words as follows.

The parse function is used in Rexx to take a string value and then break them down into words. In the above example, the words are then split and then stored in the word variables.

The output of the above program would be as follows:

```
'This'
'is'
'a'
'Tutorial'
```

Another example of parsing is shown in the following program. This time we are using a while clause to do the parsing.

```
/* Main program */
phrase = 'This is a Tutorial'
do while phrase <> ''
parse var phrase word phrase
say "'"word"'"
end
```

The above program will give the following output:

```
'This'
'is'
'a'
'Tutorial'
```

Positional Parsing

Rexx also allows one to work with positional parsing. Let's see an example of how we can achieve positional parsing with the parse statement.

Example:

```
/* Main program */
testString = "Doe       John M.   03/03/78  Mumbai              India";
parse var testString name1 11 name2 21 birthday 31 town 51 country
say name1
say name2
say birthday
say town
say country
```

From the above example, you can note that along with the variable name, we are also specifying where the string should end. So for name1, we should end by the 11th character and then starting parsing name2.

Output: The output of the above program will be as follows:

```
Doe
John M.
03/03/78
Mumbai
India
```

You can also use **relative positional parsing** in this case.

Example:

```
/* Main program */
testString = "Doe       John M.   03/03/78  Mumbai              India";
parse var testString name1 +10 name2 +10 birthday +10 town +20 country
say name1
say name2
say birthday
say town
say country
```

Output: The output of the above program will be as shown below.

```
Doe
John M.
03/03/78
Mumbai
India
```

24. Rexx – Signals

In Rexx, the signal instruction is used generally for two purposes, which are:

- One is to transfer control to another part of the program. This is normally like the go-to label which is used in other programming languages.
- The other is to go to a specific trap label.

If the signal command is used in any of the following instruction commands, the pending control structures will automatically be deactivated.

- if ... then ... else ...
- do ... end
- do i=1 to n ... end [and similar do loops]
- select when ... then ... ...etc. otherwise ... end

The general syntax of the signal statement is shown as follows:

Syntax:

```
signal labelName


signal [ VALUE ] labelExpression
```

Let's look at an example of how to use the signal statement.

Example:

```
/* Main program */
n=100.45
if \ datatype( n, wholenumber ) then
signal msg
say 'This is a whole number'
return 0
msg :
say 'This is an incorrect number'
```

Output: The output of the above program will be as shown below.

```
This is an incorrect number.
```

If you change the value of the variable n to a whole number as shown in the following program:

```
/* Main program */
n=100
if \ datatype( n, wholenumber ) then
signal msg
say ' This is a whole number '
return 0
msg :
say ' This is an incorrect number '
```

You will get the following output:

```
This is a whole number
```

One can also transfer to the value of the label as shown in the following program:

```
/* Main program */
n=1
if \ datatype( n, wholenumber ) then
signal msg
if n < 1 | n > 3 then
signal msg

signal value n
3 : say 'This is the number 3'
2 : say ' This is the number 2'
1 : say ' This is the number 1'
return n
msg :
say ' This is an incorrect number '
exit 99
```

Output: The output of the above program will be shown as follows:

```
This is the number 1
```

Trap Label Transfer Activation / Deactivation

As we have mentioned earlier, the signal instruction can also be used to transfer control to a trap label.

The general syntax of the Trap label transfer is given as follows:

Syntax:

```
signal ON conditionName [ NAME Label ]


signal OFF conditionName
```

Where,

- **conditionName –** This is the condition for which the signal should be either be turned on or off.
- **Label –** The optional label to which the program should be diverted to.

Let’s see an example of using a trap label transfer.

Example:

```
/* Main program */
signal on error
signal on failure
signal on syntax
signal on novalue
beep(1)
signal off error
signal off failure
signal off syntax
signal off novalue
exit 0
error: failure: syntax: novalue:
say 'An error has occured'
```

In the above example, we first turn the error signals on. We then add a statement which will result in an error. We then have the error trap label to display a custom error message.

Output: The output of the above program will be as shown follows:

```
An error has occurred.
```

25. Rexx – Debugging

Debugging is an important feature in any programming language. It helps the developer to diagnose errors, find the root cause and then resolve them accordingly. In Rexx, the trace utility is used for debugging. The trace instruction can be implemented in 2 ways, one is the batch mode and the other is the interactive mode. Let's look at how to implement both options.

Trace in Batch Mode

The trace command is used to give a detailed level of each Rexx command which is executed.

The general syntax of the trace statement is shown as follows:

Syntax:

```
trace [setting]
```

Where the setting can be anyone of the following options:

- A – Traces all the commands.
- C – Only traces the host commands which are sent to the operating system.
- E – Only traces the host commands which are sent to the operating system which have resulted in an error.
- F - Only traces the host commands which are sent to the operating system which have resulted in a failure.
- I – This provides an intermediate level tracing of Rexx commands.
- L – This option is if you want to label the tracing as it happens.
- N – This is the default option in which no tracing happens.

Let's take a look at an example of the trace command.

Example:

```
/* Main program */
trace A
/* Main program */
n=100.45
if \ datatype( n, wholenumber ) then
signal msg
say 'This is a whole number'
return 0
msg :
say ' This is an incorrect number '
```

Output: The output of the above program will be as follows:

```
     4 *-* n=100.45
     5 *-* if \ datatype( n, wholenumber ) then
     6 *-* signal msg
    10 *-* say 'This is an incorrect number'
This is an incorrect number
```

From the output, you can see that an additional trace was added to the output of the program. The following things can be noted about the output:

- The line number along with the statement executed is added to the trace output.
- Each line that gets executed is shown in the trace output.

Trace Function

Trace can also be enabled with the help of the trace function. The general syntax and example are shown below.

Syntax:

```
trace()
```

The above function returns the current trace level.

Parameters: None

Return type: The above function gives the current trace level

Example:

```
/* Main program */
say trace()
/* Main program */
n=100.45
if \ datatype( n, wholenumber ) then
signal msg
say 'This is a whole number'
return 0
msg :
say 'This is an incorrect number '
```

Output: The output of the above program will be as follows.

```
N
This is an incorrect number
```

The first line of N denotes that the trace is set to Normal.

Setting Trace Value

The trace level can be set with the trace function. The general syntax and example are shown below.

Syntax:

```
trace(travel_level)
```

Parameters:

- **trace_level –** This is similar to the options available for setting the trace level.

Return Type: The above function gives the current trace level.

Example:

```
/* Main program */
say trace()
current_trace=trace('A')
say current_trace
/* Main program */
n=100.45
if \ datatype( n, wholenumber ) then
signal msg
say 'This is a whole number'
return 0
msg :
say ' This is an incorrect number '
```

Output: The output of the above program will be as follows:

```
N

     4 *-* say current_trace

N

     6 *-* n=100.45

     7 *-* if \ datatype( n, wholenumber ) then

     8 *-* signal msg

    12 *-* say 'This is an incorrect number'

'This is an incorrect number'
```

Interactive Tracing

Interactive tracing is wherein, tracing is carried out as the program runs. Just like in an IDE such as Visual Studio for .Net, in which you can add breakpoints and see how each statement executes, similarly here also you can see the program as each code line runs.

The general syntax is as follows:

Syntax:

```
trace ?options
```

Where, options are the same for the trace command as shown below.

- A – Traces all the commands.
- C – Only traces the host commands which are sent to the operating system.
- E – Only traces the host commands which are sent to the operating system which have resulted in an error.
- F - Only traces the host commands which are sent to the operating system which have resulted in a failure.
- I – This provides an intermediate level tracing of Rexx commands.
- L – This option is if you want to label the tracing as it happens.
- N – This is the default option in which no tracing happens.

Let's take a look at an example of implementing active tracing.

Example:

```
/* Main program */
trace ?A
/* Main program */
n=100.45
if \ datatype( n, wholenumber ) then
signal msg
say 'This is a whole number'
return 0
msg :
say 'This is an incorrect number'
```

Output: The output of the above program will be as shown in the following program. The trace will stop at each line of code; then you need to press the Enter button to move onto the next line of code.

```
     4 *-* n=100.45
       +++ Interactive trace. "Trace Off" to end debug. ENTER to continue.


     5 *-* if \ datatype( n, wholenumber ) then


     6 *-* signal msg
    10 *-* say 'Incorrect number , should be a whole number'
Incorrect number , should be a whole number
```

26. Rexx – Error handling

Rexx has the ability to also work on Error handling as in other programming languages.

The following are some of the various error conditions that are seen in Rexx.

- **ERROR –** This even is raised whenever a command which is sent to the operating system results in an error.
- **FAILURE –** This even is raised whenever a command which is sent to the operating system results in a failure.
- **HALT –** This is normally raised whenever an operation is dependent on another operation. An example is if an I/O operation is being halted for any reason.
- **NOVALUE –** This event is raised when a value has not been assigned to a variable.
- **NOTREADY –** This is raised by any I/O device which is not ready to accept any operation.
- **SYNTAX –** This event is raised if there is any syntax error in the code.
- **LOSTDIGITS –** This event is raised when an arithmetic operation results in a loss of digits during the operation.

Trapping Errors

Errors are trapped with the help of the signal command. Let's take a look at the syntax and an example of this.

Syntax:

```
signal on [Errorcondition]
```

Where,

- **Errorcondition –** This is the error condition which is given above.

Example:

Let's take a look at an example on this.

```
/* Main program */
signal on error
signal on failure
signal on syntax
signal on novalue
beep(1)
signal off error
signal off failure
signal off syntax
signal off novalue
exit 0
error: failure: syntax: novalue:
say 'An error has occured'
```

In the above example, we first turn the error signals on. We then add a statement which will result in an error. We then have the error trap label to display a custom error message.

Output: The output of the above program will be as shown below.

```
An error has occurred.
```

An example of error codes is shown in the following program.

```
/* Main program */
signal on error
signal on failure
signal on syntax
signal on novalue
beep(1)
exit 0
error: failure: syntax: novalue:
say 'An error has occured'
say rc
say sigl
```

Output: The output of the above program will be as shown below.

```
An error has occured

40

6
```

27. Rexx – Object Oriented

When you install ooRexx as per the environment chapter, you will also have the ability to work with classes and objects. Please note that all of the following code needs to be run in the ooRexx interpreter. The normal Rexx interpreter will not be able to run this object oriented code.

Class and Method Declarations

A class is defined with the following Syntax declaration.

Syntax:

```
::class classname
```

where **classname** is the name given to the class.

A method in a class is defined with the following Syntax declaration.

Syntax:

```
::method methodname
```

Where **methodname** is the name given to the method.

A property in a class is defined with the below Syntax declaration.

Syntax:

```
::attribute propertyname
```

Where **propertyname** is the name given to the property.

Example:

The following is an example of a class in Rexx.

```
::class student
::attribute StudentID
::attribute StudentName
```

The following points need to be noted about the above program.

- The name of the class is student.
- The class has 2 properties, StudentID and StudentName.

Getter and Setter Methods

The Getter and Setter methods are used to automatically set and get the values of the properties. In Rexx, when you declare a property with the attribute keyword, the getter and setter methods are already put in place.

Example:

```
::class student
::attribute StudentID
::attribute StudentName
```

In the above example, there would be Getter and Setter methods for StudentId and StudentName.

An example of how they can be used is shown in the following program.

```
/* Main program */
value = .student~new
value~StudentID=1
value~StudentName='Joe'
say value~StudentID
say value~StudentName
exit 0
::class student
::attribute StudentID
::attribute StudentName
```

Output: The output of the above program will be as shown below.

```
1
Joe
```

Instance Methods

Objects can be created from the class via the **~new operator**. A method from the class can be called in the following way.

```
Object~methodname
```

Where **methodname** is the method which needs to be invoked from the class.

Example:

The following example shows how an object can be created from a class and the respective method invoked.

```
/* Main program */
value = .student~new
value~StudentID=1
value~StudentName='Joe'
value~Marks1=10
value~Marks2=20
value~Marks3=30
total=value~Total(value~Marks1,value~Marks2,value~Marks3)
say total
exit 0
::class student
::attribute StudentID
::attribute StudentName
::attribute Marks1
::attribute Marks2
::attribute Marks3
::method 'Total'
use arg a,b,c
return (ABS(a) + ABS(b) + ABS(c))
```

Output: The output of the above program will be as shown below.

```
60
```

Creating Multiple Objects

One can also create multiple objects of a class. The following example will show how this can be achieved.

In here we are creating 3 objects (st, st1 and st2) and calling their instance members and instance methods accordingly.

Let’s take a look at an example of how multiple objects can be created.

Example:

```
/* Main program */
st = .student~new
st~StudentID=1
st~StudentName='Joe'
st~Marks1=10
st~Marks2=20
st~Marks3=30
total=st~Total(st~Marks1,st~Marks2,st~Marks3)
say total

st1 = .student~new
```

```
st1~StudentID=2
st1~StudentName='John'
st1~Marks1=10
st1~Marks2=20
st1~Marks3=40
total=st1~Total(st1~Marks1,st1~Marks2,st1~Marks3)
say total

st2 = .student~new
st2~StudentID=3
st2~StudentName='Mark'
st2~Marks1=10
st2~Marks2=20
st2~Marks3=30
total=st2~Total(st2~Marks1,st2~Marks2,st2~Marks3)
say total

exit 0
::class student
::attribute StudentID
::attribute StudentName
::attribute Marks1
::attribute Marks2
::attribute Marks3
::method 'Total'
use arg a,b,c
return (ABS(a) + ABS(b) + ABS(c))
```

Output: The output of the above program will be as shown below.

```
60

70

60
```

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as **subclass** (derived class, child class) and the class whose properties are inherited is known as **superclass** (base class, parent class).

Let's see an example of inheritance in Rexx. In the following example we are creating a class called **Person**. From there we use the subclass keyword to create the **Student class** as a **sub-class of Person**.

```
/* Main program */
st = .student~new
st~StudentID=1
st~StudentName='Joe'
st~Marks1=10
st~Marks2=20
st~Marks3=30
say st~Total(st~Marks1,st~Marks2,st~Marks3)

exit 0
::class Person
::class student subclass Person
::attribute StudentID
::attribute StudentName
::attribute Marks1
::attribute Marks2
::attribute Marks3
::method 'Total'
use arg a,b,c
return (ABS(a) + ABS(b) + ABS(c))
```

Output: The output of the above program will be as shown below.

```
60
```

28. Rexx – Portability

Portability is an important aspect in any programming language. As one knows, Rexx is available in a variety of operating systems such as Windows and Linux. So it need to be ensured that when one develops a program on the Windows platform, that it has the necessary precautions taken if the same programs runs on a Linux platform.

Rexx has the ability to run system level commands. There are commands which can be used to understand what is the operating system on which it is running on. Based on the output it can then take the appropriate actions to see what are the commands that can be run on this operating system.

Example

The following example shows how the parse functions are used to get the details of the operating system on which the program is running.

```
/* Main program */
parse version language level date month year .
parse source system invocation filename .
language = translate(language)
if pos('REGINA',language) = 0 then
say 'Error , the default interpreter is not Regina' language
say 'The Interpreter version/release date is:' date month year
say 'The Language level is: ' level
say 'The Operating System is'

select
when system = 'WIN32' then
'ver'
when system = 'UNIX' | system = 'LINUX' then
'uname -a'
otherwise
say 'Unknown System:' system
end
if rc <> 0 then

say 'Error :' rc
```

Output: The output will vary depending on operating system. A sample output is given below.

```
The Interpreter version/release date: 5 Apr 2015

The Language level is:  5.00

The Operating System is

Unknown System: WIN64

Bad return code: RC
```

29. Rexx – Extended Functions

Rexx provides a number of extended functions that gives a wide variety of functionality, most of them which allow you to interact with the Operating System. Let's look at some of them in detail as explained below.

b2c

This function converts a binary value to a string value.

Syntax:

```
B2C(binstring)
```

Parameters:

- **binstring –** This is the binary value which needs to be converted to a string.

Return Value: The corresponding string value for the input-ed binary value.

Example:

```
/* Main program */
options arexx_bifs
say b2c('01100011')
```

Output: When we run the above program, we will get the following result.

```
c
```

bitclr

This function is used to toggle the specified bit in the binary string to 0.

Syntax:

```
bitclr(binstring,position)
```

Parameters:

- **binstring -** The input binary string.
- **position –** The position in the binary string to which the bit needs to be set to 0.

Return Value: The function will return the new value of the binary string with the positional bit set to 0.

Example:

```
/* Main program */

options arexx_bifs
say c2x(bitclr('0414'x,4))
```

In the above program, in order to display the bit string, the function c2x is used to convert the bit string to a character based representation which can be seen in the console window.

Output:

```
0404
```

bitcomp

This function is used to compare 2 binary strings starting with bit 0.

Syntax:

```
bitcomp(binstring1, binstring2)
```

Parameters:

- **binstring1, binstring2 –** The binary strings which need to be compared.

Return Value: This function returns the bit number of the first bit by which the two strings differ. Else if the two strings are identical, it will return -1.

Example:

```
/* Main program */
options arexx_bifs
say bitcomp('ee'x, 'fe',x)
```

Output: When we run the above program we will get the following result.

```
0
```

buftype

This function is used to display the contents of the stack which is normally used in debugging purposes.

Syntax:

```
buftype()
```

Parameters: None

Return Value: The details of the stack are displayed.

Example:

```
/* Main program */
options arexx_bifs
say buftype()
```

Output: When we run the above program we will get the following result. This depends on the current state of the system.

The following program is just an example.

```
==> Name: SESSION

==> Lines: 0

==> Buffer: 0

==> End of Stack
```

crypt

This function is used to encrypt a string.

Syntax:

```
crypt(source,salt)
```

Parameters:

- **source –** This is the source string which needs to be encrypted.
- **salt –** This is the characters which need to be used for the encryption process.

Return Value: This function returns the encrypted string

Example:

```
/* Main program */
options arexx_bifs
say crypt('abc','gh')
```

Output: When we run the above program we will get the following result.

```
abc
```

Note that if the source string is the same as the output string, then it means that the operating system does not support encryption.

fork

This function is used to spawn a new child process on the system.

Syntax:

```
fork()
```

Parameters: None

Return Value: This function returns the process id of the new process.

Example:

```
/* Main program */
options arexx_bifs
say fork()
```

Output: When we run the above program we will get the following result. The process number returned will differ from system to system.

An example of the result is shown below.

```
1
```

getpid

This function gets the id of the current running process.

Syntax:

```
getpid()
```

Parameters: None

Return Value: This function gets the id of the current running process.

Example:

```
/* Main program */
options arexx_bifs
say getpid()
```

Output: When we run the above program we will get the following result. The process number returned will differ from system to system. An example of the result is shown below.

```
4372
```

hash

This function returns the hash value of a string.

Syntax:

```
hash(str)
```

Parameters:

- **str –** This is the input string.

Return Value: This function returns the hash value of a string.

Example:

```
/* Main program */
options arexx_bifs
say hash('cde')
```

Output: When we run the above program we will get the following result.

```
44
```

30. Rexx – Instructions

Rexx provides a number of instructions that gives a wide variety of functionality, most of them which allow you to interact with the Operating System. Let's look at some of them in detail.

address

This function is used to display the current command environment.

Syntax:

```
address()
```

Parameters: None

Return Value: The return value is the current command environment.

Example:

```
/* Main program */
options arexx_bifs
say address()
```

Output: When we run the above program we will get the following result.

```
SYSTEM
```

Let's take a look at some more variations of the address command.

Syntax:

```
address environment command
```

Parameters:

- **environment –** This is the current environment to which the command will be sent to.
- **command –** This is the operating system command sent to the environment.

Return Value: The return value is the output of the command sent to the operating system.

Example:

```
/* Main program */
options arexx_bifs
say address()
address system dir
```

Output: When we run the above program we will get the following result. The output depends from system to system. The following is an example of how the output would look like.

```
SYSTEM
Volume in drive H is Apps
 Volume Serial Number is 8E66-AC3D


Directory of H:\Apps\Programs


06/29/2016  09:33 PM    <DIR>          .
06/29/2016  09:33 PM    <DIR>          ..
06/30/2016  12:55 PM                73 main.rexx
               1 File(s)             73 bytes
               2 Dir(s)  313,090,973,696 bytes free
```

drop

This function is used to unassign a variable.

Syntax:

```
drop variable_name
```

Parameters:

- **variable_name –** The variable which needs to be dropped.

Return Value: None

Example:

```
/* Main program */
options arexx_bifs
a = 5
say a
drop a
say a
```

Output: When we run the above program we will get the following result.

```
5

A
```

From the output you can see that since the variable **a** is dropped, it does not have a value assigned to it.

interpret

Interprets or executes the defined instruction.

Syntax:

```
Interpret command
```

Parameters:

- **command –** The command sent to the interpret instruction.

Return Value: Returns the output of the command sent to the interpret instruction.

Example:

```
/* Main program */
options arexx_bifs
interpret say 'Hello'
```

Output: When we run the above program we will get the following result.

```
HELLO
```

nop

This function means to perform no operation. This command is normally used in **if statements**.

Syntax:

```
nop
```

Parameters: None

Return Value: None

Example:

```
/* Main program */
options arexx_bifs
status='Yes'
if status = 'YES'
then nop
```

Output: When we run the above program **no result** will be returned.

Pull

This is used to pull input from the stack or from the default stream.

Syntax:

```
Pull variable
```

Parameters:

- **Variable –** The variable to which the input value will be assigned to.

Return Value: None

Example:

```
/* Main program */
options arexx_bifs
pull input
say input
```

Output: When you run the above program, you need to enter some input. If you enter the input value of 'Tutorial', the program will return the word 'TUTORIAL'.

push

This is used to push a value onto the Rexx stack.

Syntax:

```
push value
```

Parameters:

Value – The value which needs to be pushed onto the stack.

Return Value: None

Example:

```
/* Main program */
options arexx_bifs
push Tutorial
pull input
say input
```

Output: When we run the above program, we will get the following result.

```
TUTORIAL
```

31. Rexx – Implementations

The Rexx language has a lot of various implementations as we have already seen in the previous chapters. Each implementation has its own functionality. Let's look at the various implementations available for Rexx.

OoRexx

This is the object oriented version of Rexx. By default, the Rexx basic implementation is all based on procedures. But with ooRexx you can offer greater flexibility by having an Object oriented approach to Rexx. By using ooRexx you can have better re-use through creating re-usable classes and objects.

The following program is an example of a simple Rexx program which can be run with the ooRexx implementer.

```
/* Main program */
say ‘hello’
```

To run this program, run the following command.

```
rexx main.rexx
```

When you run the above command, you will get the following output.

```
hello
```

Netrexx

This is for all Java based developers as it provides a Java based alternative for the Rexx language. So all of the objects are based on the Java Object Model. The advantage of this framework is that since Java is a widely popular language it becomes easier for developers to use this framework. So in this implementation, the Rexx code is converted to a Java program which can then be run on any Java virtual machine.

The following code is an example of a NetRexx program.

Create a file called **main.nrx** and place the following code in the file.

```
/* Main program */
say ‘hello’
```

To compile the code run the following command:

```
NetRexxC main.nrx
```

You will then get the following output. NetRexxC is the compiler which converts the Rexx program to its java equivalent.

```
java -cp ";;G:\NetRexx-3.04GA\lib\NetRexxF.jar;." -Dnrx.compiler=ecj
org.netrexx.process.NetRexxC  main.nrx

NetRexx portable processor 3.04 GA build 4-20150630-1657

Copyright (c) RexxLA, 2011,2015.   All rights reserved.

Parts Copyright (c) IBM Corporation, 1995,2008.

Program main.nrx

Compilation of 'main.nrx' successful
```

You can now run your java program using the following java command.

```
java main
```

When you run the above command, you will get the following output.

```
Hello
```

Brexx

This is a lightweight implementation of Rexx. This is a lighter package than the standard Rexx implementer. But it still has the full functionality of Rexx.

The following code is an example of a BRexx program.

```
/* Main program */
say ‘hello’
```

To run the program, run the following command.

```
rexx32 main.rexx
```

When you run the above command, you will get the following output.

```
hello
```

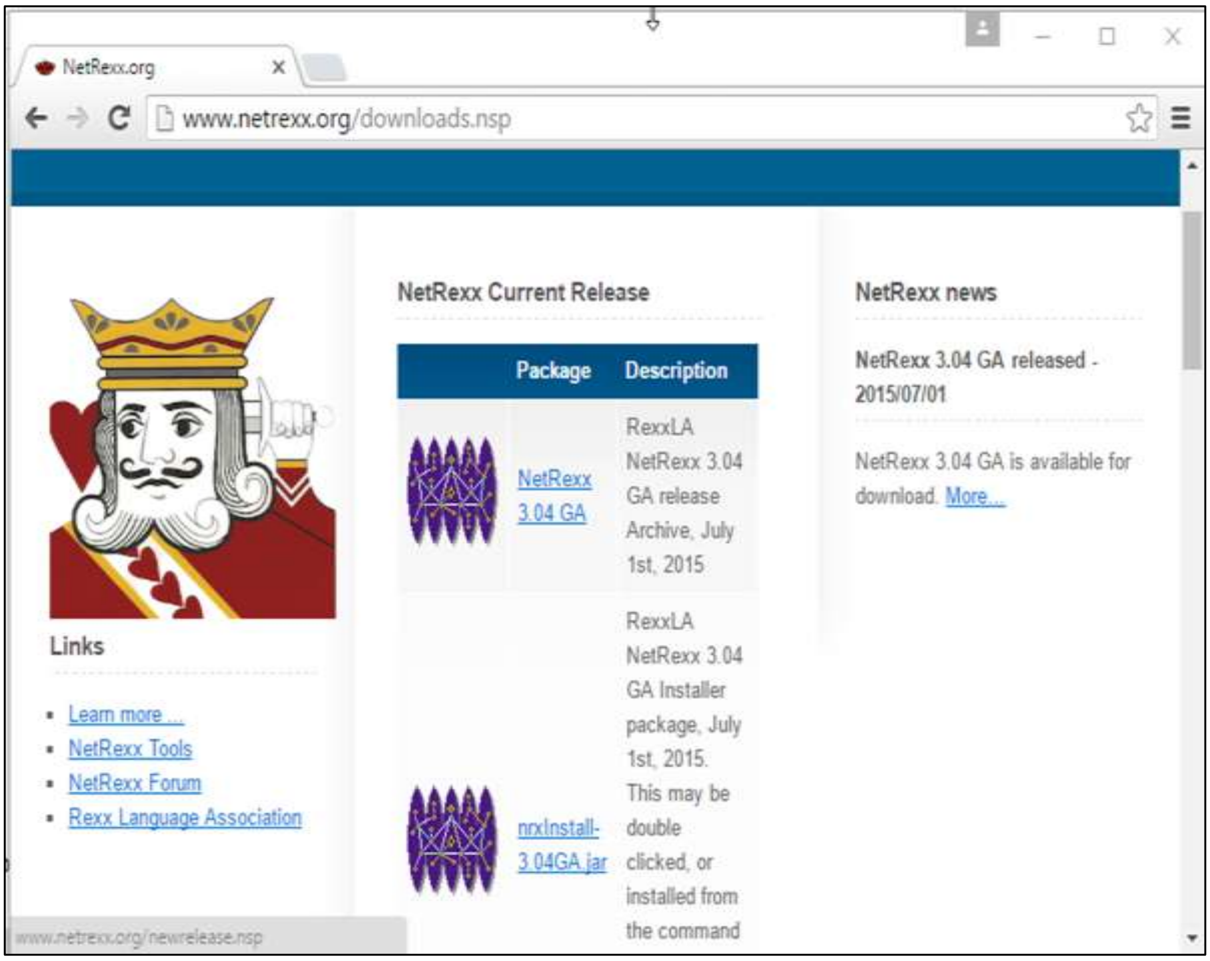
32. Rexx – NetRexx

NetRexx is the java implementation of Rexx. In NetRexx, the implementer is used to convert the Rexx program to a java program which can then be run on any Java virtual machine.

Setting up NetRexx

The first step in NetRexx is to set it up on the local machine. To do this, one needs to perform the following steps:

Step 1: Go to the NetRexx download site - http://www.netrexx.org/downloads.nsp


NetRexx.org
www.netrexx.org/downloads.nsp
Links
Learn more ...
NetRexx Tools
NetRexx Forum
Rexx Language Association
www.netrexx.org/newrelease.nsp

NetRexx Current Release

Package	Description
NetRexx 3.04 GA	RexxLA NetRexx 3.04 GA release Archive, July 1st, 2015
nrxInstall-3.04GA.jar	RexxLA NetRexx 3.04 GA Installer package, July 1st, 2015. This may be double clicked, or installed from the command

NetRexx news

NetRexx 3.04 GA released - 2015/07/01

NetRexx 3.04 GA is available for download. More...

Download the NetRexx.3.04.GA file.

Step 2: Ensure that java is installed and running on your system. You can verify that java is running by using the java–version command.

An example of the output is shown below.

```
H:\>java -version

java version "1.7.0_79"

Java(TM) SE Runtime Environment (build 1.7.0_79-b15)

Java HotSpot(TM) Client VM (build 24.79-b02, mixed mode, sharing)
```

Step 3: Unzip the contents of the Netrexx zipped file. Copy the files from the NetRexx-3.04GA\lib folder to your java installation/lib/etc folder.

Step 4: Add the NetRexx-3.04GA\bin path to the path variable on the system.

Running the First NetRexx Program

Create a file called **main.nrx** and place the following code in the file.

```
/* Main program */
say ‘hello’
```

To compile the code run the following command.

```
NetRexxC main.nrx
```

You will then get the following output. NetRexxC is the compiler which converts the rexx program to its java equivalent.

```
java -cp ";;G:\NetRexx-3.04GA\lib\NetRexxF.jar;." -Dnrx.compiler=ecj
org.netrexx.process.NetRexxC  main.nrx

NetRexx portable processor 3.04 GA build 4-20150630-1657

Copyright (c) RexxLA, 2011,2015.   All rights reserved.

Parts Copyright (c) IBM Corporation, 1995,2008.

Program main.nrx

Compilation of 'main.nrx' successful
```

You can now run your java program using the following java command.

```
java main
```

When you run the above command, you will get the following output.

```
Hello
```

Let us now discuss some of the **special aspects of the Netrexx library**.

Indexed Strings

In NetRexx, strings can become the indexes to arrays. An example is shown below.

Example:

```
/* Main program */
value='unknown'
value['a']='b'
c='a'
say value[c]
```

Output: When we run the above program, we will get the following result.

```
b
```

Multiple Indexes

In NetRexx, you can have multiple indexes for arrays. An example is shown below.

Example:

```
/* Main program */
value ='null'
value['a', 'b'] = 1
say value['a', 'b']
```

Output: When we run the above program we will get the following result.

```
1
```

ask Command

This command is used to read a line from the default input stream

Syntax:

```
variable = ask
```

Parameters:

- **Variable –** The variable to store the value entered by the user.

Return Value: None

Example:

```
/* Main program */
input = ask
say input
```

Output: When we run the above program we will get the following result. If we input the value hello, we will get the output of hello.

digits Command

This command is used to display the current value of the digits' value.

Syntax:

```
digits
```

Parameters: None

Return Value: The current value of the digits' variable as a string.

Example:

```
/* Main program */
say digits
```

Output: When we run the above program we will get the following result.

```
9
```

form Command

This command is used to display the current value of the form value.

Syntax:

```
form
```

Parameters: None

Return Value: The current value of the form variable as a string.

Example:

```
/* Main program */
say form
```

Output: When we run the above program we will get the following result.

```
Scientific
```

length Command

This command is used to display the length of a string value.

Syntax:

```
str.length
```

Parameters: None

Return Value: The length of the string.

Example:

```
/* Main program */
value ='null'
say value.length
```

Output: When we run the above program we will get the following result.

```
4
```

version Command

This command is used to return the current version of NetRexx being used.

Syntax:

```
version
```

Parameters: None

Return Value: The current version of NetRexx being used.

Example:

```
/* Main program */
say version
```

Output: When we run the above program we will get the following result.

```
NetRexx 3.04 01 Jul 2015
```

trace Command

This command is used to return the current trace setting being used by NetRexx.

Syntax:

```
trace
```

Parameters: None

Return Value: The current trace setting being used by NetRexx.

Example:

```
/* Main program */
say trace
```

Output:

When we run the above program we will get the following result.

```
Off
```

33. Rexx – BRexx

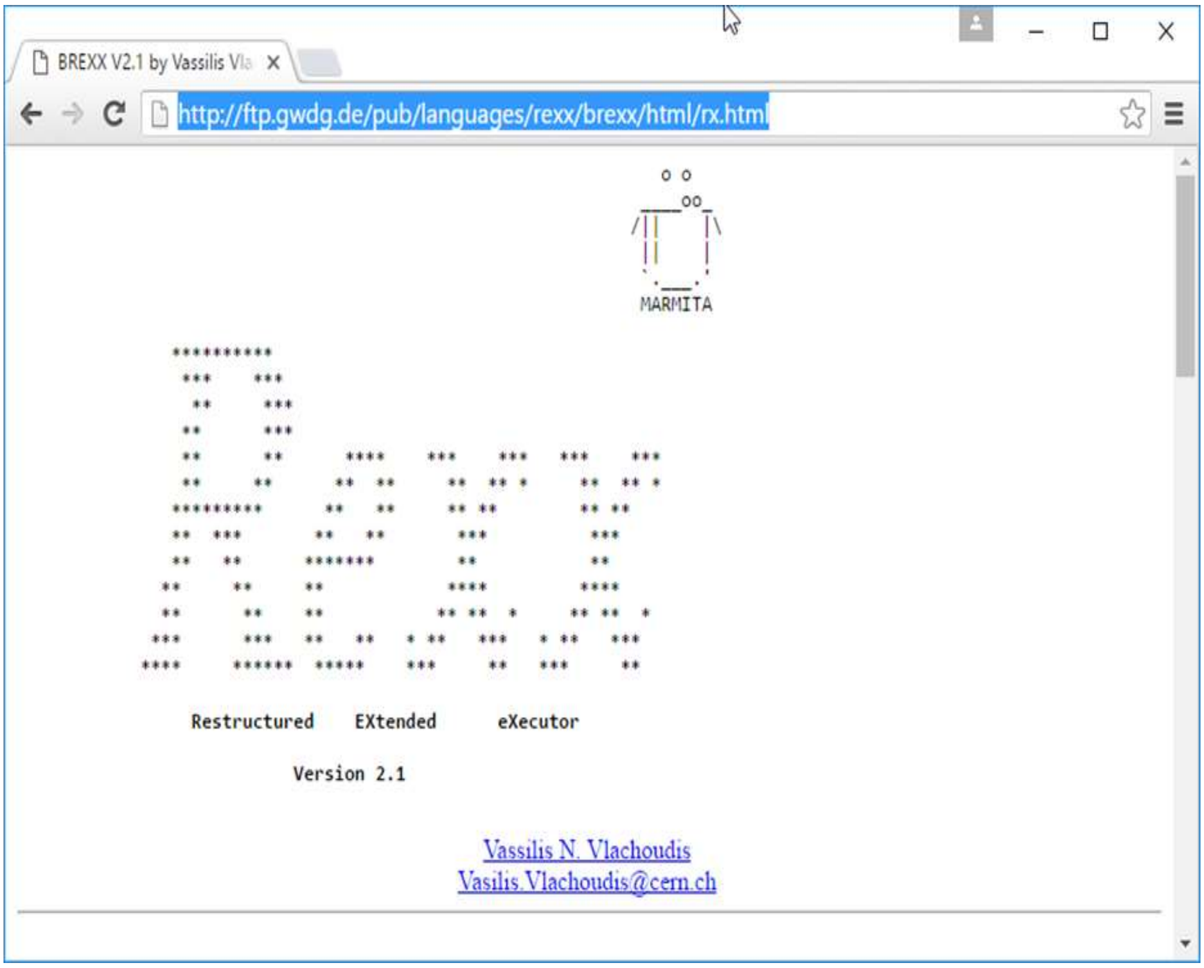
BRexx is a lighter weight implementation of Rexx. It still has a lot of functionality to offer as part of the Rexx implementation.

Setting up BRexx

The first step in BRexx is to set it up on the local machine. To do this, one needs to perform the following steps:

Step 1: Go to the BRexx download site –

http://ftp.gwdg.de/pub/languages/rexx/brexx/html/rx.html


BREXX V2.1 by Vassilis Vla
http://ftp.gwdg.de/pub/languages/rexx/brexx/html/rx.html
MARMITA

Restructured EXtended eXecutor

Version 2.1

Vassilis N. Vlachoudis
Vasilis.Vlachoudis@cern.ch

Go to the downloads section and download the product.

Step 2: Unzip the contents of the Brexx zipped file.

Step 3: Add the BRexx\bin path to the path variable on the system.

Step 4: Create a new variable called RXLIB and point it to the lib folder in the Brexx folder.

Running the First BRexx Program

Create a file called **main.rexx** and place the following code in the file.

```
/* Main program */
say ‘hello’
```

To run the program, run the following command.

```
rexx32 main.rexx
```

When you run the above command, you will get the following output.

```
hello
```

Let us now discuss some of the most commonly used functions available in the BRexx library.

acos Command

This command is used to get the arc-cosine conversion of a number.

Syntax:

```
acos(value)
```

Parameters:

- **value –** The value as an input to the arc-cosine function.

Return Value: This method returns the arc-cosine value of the input-ed number.

Example:

```
/* Main program */
say acos(0.23)
```

Output: When we run the above program we will get the following result.

```
1.3387186
```

cos Command

This command is used to get the cosine conversion of a number.

Syntax:

```
cos(value)
```

Parameters:

- **value –** The value as an input to the cosine function.

Return Value: This method returns the cosine value of the inputed number.

Example:

```
/* Main program */
say cos(45)
```

Output: When we run the above program we will get the following result.

```
0.52532199
```

sin Command

This command is used to get the sine conversion of a number.

Syntax:

```
sin(value)
```

Parameters:

- **value –** The value as an input to the sine function.

Return Value: This method returns the sine value of the input-ed number.

Example:

```
/* Main program */
say sin(45)
```

Output: When we the run above program we will get the following result.

```
0.85090352
```

asin Command

This command is used to get the arc-sine conversion of a number.

Syntax:

```
asin(value)
```

Parameters:

- **value –** The value as an input to the arc-sine function.

Return Value: This method returns the arc-sine value of the input-ed number.

Example:

```
/* Main program */
say asin(0.1)
```

Output: When we run the above program we will get the following result.

```
0.10016742
```

tan Command

This command is used to get the tangent conversion of a number.

Syntax:

```
tan(value)
```

Parameters:

- **value –** The value as an input to the tangent function.

Return Value: This method returns the tangent value of the input-ed number.

Example:

```
/* Main program */
say tan(45)
```

Output: When we run the above program we will get the following result.

```
1.6197752
```

atan Command

This command is used to get the arc-tangent conversion of a number.

Syntax:

```
atan(value)
```

Parameters:

- **value –** The value as an input to the arc-tangent function.

Return Value: This method returns the arc-tangent value of the input-ed number.

Example:

```
/* Main program */
say atan(0.1)
```

Output: When we run the above program we will get the following result.

```
0.099668652
```

mkdir Command

This command is used to create a directory in the current working directory.

Syntax:

```
mkdir dirname
```

Parameters:

- **dirname –** The name of the new directory which needs to be created.

Return Value: None.

Example:

```
/* Main program */
mkdir Test
```

Output: When we run the above program, the directory called Test will be created in the working directory.

rmdir Command

This command is used to remove a directory in the current working directory.

Syntax:

```
rmdir dirname
```

Parameters:

- **dirname –** The name of the directory which needs to be removed.

Return Value: None

Example:

```
/* Main program */
rmdir Test
```

Output: When we run the above program, the directory called Test will be removed.

dir Command

This command is used to return the entire directory listing.

Syntax:

```
dir
```

Parameters: None

Return Value: The entire directory listing.

Example:

```
/* Main program */
say dir
```

Output: An example of the output is shown below.

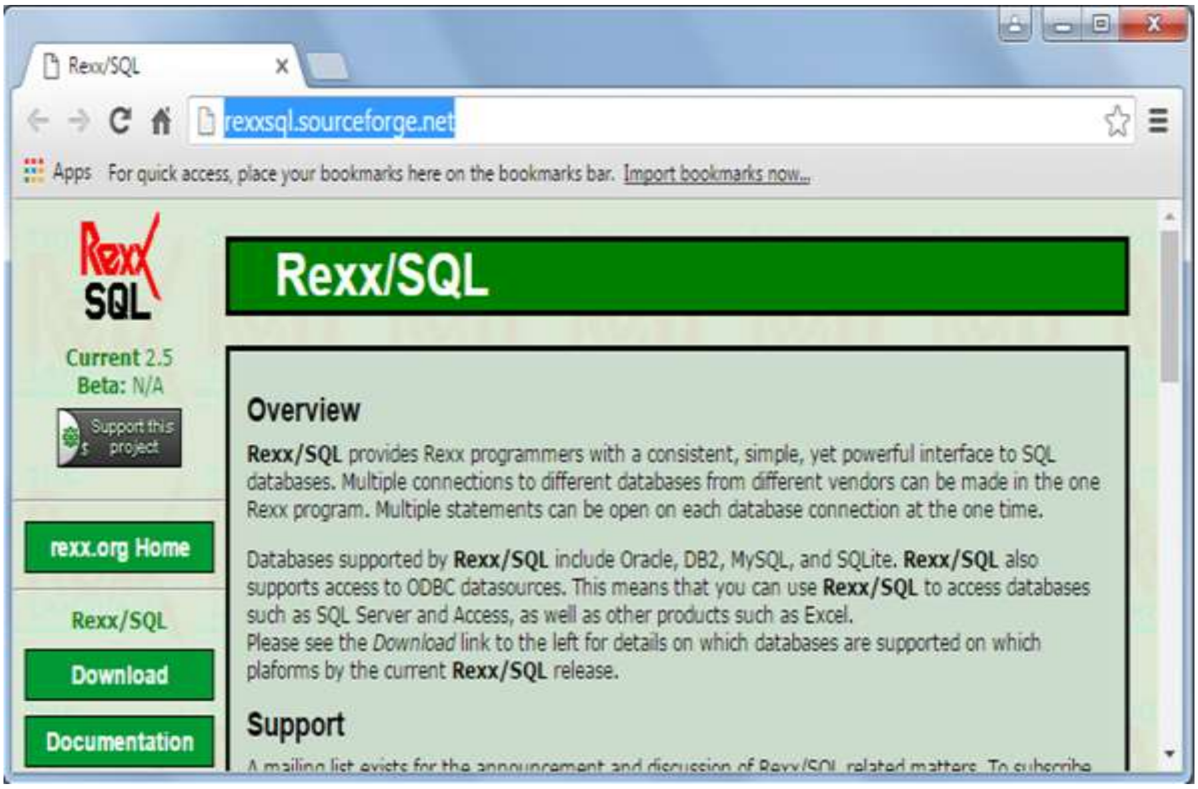
```
29/10/2015  08:45    <DIR>          Apps
11/06/2009  01:42                24 autoexec.bat
11/06/2009  01:42                10 config.sys
10/03/2016  09:09    <DIR>          Data
25/05/2015  00:33    <DIR>          Intel
14/07/2009  06:37    <DIR>          PerfLogs
26/06/2016  08:28    <DIR>          Program Files
26/06/2016  08:25    <DIR>          Temp
04/05/2016  09:44    <DIR>          Users
05/07/2016  14:59    <DIR>          Windows
               3 File(s)             34 bytes
              11 Dir(s)  34,439,139,328 bytes free
```

34. Rexx – Databases

Rexx has the ability to work with a variety of databases which are listed below.

- HSQLDB
- Oracle
- SQL Server
- MySQL
- MongoDB

All the information for Rexx databases can be found once you click on the following link – http://rexxsql.sourceforge.net/



Rexx/SQL

Overview

Rexx/SQL provides Rexx programmers with a consistent, simple, yet powerful interface to SQL databases. Multiple connections to different databases from different vendors can be made in the one Rexx program. Multiple statements can be open on each database connection at the one time.

Databases supported by **Rexx/SQL** include Oracle, DB2, MySQL, and SQLite. **Rexx/SQL** also supports access to ODBC datasources. This means that you can use **Rexx/SQL** to access databases such as SQL Server and Access, as well as other products such as Excel.
Please see the *Download* link to the left for details on which databases are supported on which plaforms by the current **Rexx/SQL** release.

Support

A mailing list exists for the announcement and discussion of Rexx/SQL related matters. To subscribe

In our example, we are going to use MySQL DB as a sample. So the first step is to ensure to download the required drivers from the Rexx SQL site so that Rexx programs can work with SQL accordingly. So follow the subsequent steps to ensure that Rexx programs can work with MySQL databases.

Step 1: Go to the following drivers download page from the Rexx site –

https://sourceforge.net/projects/rexxsql/files/rexxsql/2.6/

Step 2: Download the MYSQL drivers - rxsql26B3_my_w32_ooRexx

Step 3: Unzip the contents to the local machine.

Step 4: Add the path of the unzipped folder to the path variable on your machine.

For all the subsequent examples, make sure of the following pointers are in place:

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Ensure you have downloaded the mysql jar file and added the file to your classpath.
- You have gone through MySQL tutorial

Database Connection

To establish a database connection, you first need to the Rexxsql DLL and then use the SQLConnect function to establish a connection to the database. The syntax and example of how this can be achieved is given below.

Syntax:

```
SQLConnect(cname,username,password,dbname)
```

Parameters:

- **cname –** This is the name to give to the connection.
- **username –** The user name to connect to the database
- **password –** The password to connect to the database
- **dbname –** The database schema to connect to.

Return Value: A value equal to 0 will mean that the database connection is successful.

Example:

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
say SQLConnect(c1,' testuser ',' test123','testdb')
```

Output: The output of the above program would be as shown below.

```
0
```

Creating a Database Table

The next step after connecting to the database is to create the tables in our database. The following example shows how to create a table in the database using Rexx. All of the commands in Rexx SQL are executed by using the SQLCommand function.

Syntax:

```
SQLConnect(sname,statement)
```

Parameters:

- **sname –** This is the name to give to the statement to execute.
- **statement –** This is the statement which needs to be executed against the database.

Return Value: A value equal to 0 will mean that the command was successful.

Example:

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = 'create table employee (first_name char(20) not null, last_name
char(20),age int, sex
char(1), income float)'
if SQLCommand(c2,sqlstr)==0 then say 'Employee table created'
```

Output: The output of the above program would be as shown below.

```
Connect Succedded

Changed database to testdb

Employee table created
```

Operations on a Database Table

The following types of operations are most commonly performed on a database table.

Insert Operation

It is required when you want to create your records into a database table.

Example: The following example will insert a record in the employee table.

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = "INSERT INTO employee(first_name,last_name,age,sex,income) values
('Mac','Mohan',20,'M',2000)"
say SQLCommand(c2,sqlstr)
```

if you look into the MySQL testDB database, you will see that the record has been inserted.

Read Operation

A READ Operation on any database means to fetch some useful information from the database. Once our database connection is established, you are ready to make a query into this database.

Example: The following example shows how to fetch all the records from the employee table.

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = "select first_name,last_name,age,sex,income from employee"
say SQLCommand(c2,sqlstr)
say c2.first_name.1
say c2.last_name.1
say c2.age.1
say c2.sex.1
say c2.income.1
```

Output: The output of the above program would be as shown below.

```
Connect Succedded

Changed database to testdb

0

Mac

MOhan

20

M

2000
```

Update Operation

The UPDATE Operation on any database means to update one or more records, which are already available in the database. The following procedure updates all the records having SEX as 'M'.

in the following example, we increase the AGE of all the males by one year.

Example:

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = 'M'"
say SQLCommand(c2,sqlstr)
```

Delete Operation

The DELETE operation is required when you want to delete some records from your database. The following program shows the procedure to delete all the records from EMPLOYEE where AGE is more than 20.

Example:

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = "DELETE FROM EMPLOYEE WHERE AGE > 20"
say SQLCommand(c2,sqlstr)
```

Closing a connection

The following command can be used to close a connection to the database.

Syntax:

```
SQLDisconnect(cname)
```

Parameters:

- **cname –** This is the name of the connection.

Return Value: A value equal to 0 will mean that the command was successful.

Example:

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLDisconnect(c1)==0 then say 'Disconnected'
```

Output: The output of the above program would be as shown below.

```
Connect Succedded

Disconnected
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties:

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.

Here is a simple example of how to implement transactions.

```
/* Main program */
Call RxFuncAdd 'SQLLoadFuncs', 'rexxsql', 'SQLLoadFuncs'
Call SQLLoadFuncs
if SQLConnect(c1,'testuser','test123','testdb')==0 then say 'Connect Succedded'
if SQLCommand(u1,"use testdb")==0 then say 'Changed database to testdb'
sqlstr = "DELETE FROM EMPLOYEE WHERE AGE > 20"
if SQLCommand(c2,sqlstr)==0 then
if sqlcommit()==0 then say committed
```

Output: The output of the above program would be as shown below.

```
Connect Succedded

Changed database to testdb

COMMITTED
```

Commit Operation

The commit operation is what tells the database to proceed ahead with the operation and finalize all changes to the database. In our above example, this is achieved by the following command.

```
Sqlcommit()
```

Rollback Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use rollback method. In our above example, this is achieved by the following command.

```
SqlRollback()
```

35. Rexx – Handheld & Embedded Programming

Handheld devices have come a long way and Rexx has a lot of ways in which it can run on these devices. Rexx has support for Pocket PC's, Palm devices, PDA's and other smart phone devices. The advantage of Rexx to work on these platforms is that Rexx is really a small weight programming system which just runs in the span of a few kilobytes. Hence it becomes easier to run Rexx programs on these devices.

Rexx on handheld devices can run in the following modes:

- The first mode is the native node where it runs directly on the operating system itself. The advantage on running in this mode is that it is faster since it runs directly off the operating system.
- The next mode is on top of the DOS or emulator program on top of the hand held device. The advantage of this mode is that is can run on any type of operating system as long as the emulator runs on that operating system.

The Rexx Interpreters for the various hand held devices categories are shown below.

- Windows CE – Brexx
- Palm OS – Rexx for Palm OS
- Symbian OS - Regina

For the DOS emulator, the following steps need to be carried out:

Step 1: First is to download PocketDOS which is a popular DOS emulator. It is designed to run on many operating systems and has support for common VGA screens and serial and parallel ports.

Step 2: The next step is to download the BRexx files for 16-bit DOS to a Windows PC.

Step 3: The final step is to use ActiveSync to sync the Rexx program to the handheld device.

There are some other commercial DOS based products available. XTM is a product which falls under this category. The features of this product are as follows:

- Support for the 80186 CPU and instruction set.
- It kind of works off the BIOS code for better performance.
- It can provide emulation for the Math co-processor, version 8087 MPU
- It provides access to the serial ports.
- It supports a variety of languages such as English, French and German.

36. Rexx – Performance

One of the key aspects of any programming language is the performance of the application. Special practices need to be taken care of to ensure that the application's performance is not hampered. Let's look at some of the considerations described in steps for better understanding:

Step 1: Try to reduce the number of instructions – In Rexx each instruction carries an overhead. So try to reduce the number of instructions in your program. An example of instructions that can be redesigned is shown below.

Instead of using multiple if else statements one can use the parse statement. So like in the following program, instead of having an if condition for each value, and getting the value of word1, word2, word3 and word4, use the parse statement.

```
/* Main program */


parse value 'This is a Tutorial' with word1 word2 word3 word4
say "'"word1"'"
say "'"word2"'"
say "'"word3"'"
say "'"word4"'"
```

Step 2: Try to combine multiple statements into one statement. An example is shown below.

Suppose if you had the following code which did the assignment for – **a and b** and passed it to a method called **proc**.

```
do i=1 to 100
a = 0
b = 1
call proc a,b
end
```

You can easily replace the above given code with the following code using the parse statement.

```
do i=1 for 100
parse value 0 1 with
a,
b,
call proc a,b
end
```

Step 3: Try to replace the **do..to loop** with the **do..for loop** wherever possible. This is normally recommended when the control variable is being iterated through a loop.

```
/* Main program */
do i=1 to 10
say i
end
```

The above program should be replaced by the following program.

```
/* Main program */
do i=1 for 10
say i
end
```

Step 4: If possible, remove the for condition from a do loop as shown in the following program. If the control variable is not required, then just put the end value in the do loop as shown below.

```
/* Main program */
do 10
say hello
end
```

Step 5: In a **select clause**, whatever u feel is the best condition which will be evaluated needs to put first in the **when clause**. So in the following example, if we know that 1 is the most frequent option, we put the **when 1 clause** as the first clause in the select statement.

```
/* Main program */
select
when 1 then say'1'
when 2 then say'2'
otherwise say '3'
end
```

37. Rexx – Best Programming Practices

Every programmer wants their program to be the best when it comes to quality and efficiency. The following are some of the best programing practices or hints when writing Rexx programs which can help one achieve these goals.

Hint 1

Use the address command before you issue any command to the operating system or command prompt. This will help you get the address space beforehand in memory and cause your program to run more efficiently.

An example of the address command is shown below.

```
/* Main program */
address system dir
```

The output of the command is as follows, but it could vary from system to system.

```
Volume in drive H is Apps
 Volume Serial Number is 8E66-AC3D


 Directory of H:\


06/30/2016  01:28 AM    <DIR>          Apps
07/05/2016  03:40 AM               463 main.class
07/07/2016  01:30 AM                46 main.nrx
07/07/2016  01:42 AM                38 main.rexx
3 File(s)            547 bytes
Dir(s)  313,085,173,760 bytes free
```

Hint 2

Ensure all commands to the operating system are in upper case and in quotes wherever possible.

An example for the same is shown below.

```
/* Main program */
options arexx_bifs
say chdir('\REXXML100')
say directory()
```

Output: When we run the above program, we will get the following result.

```
0
D:\rexxxml100
```

Hint 3

Avoid creating big comment blocks as shown in the following program.

```
/******/
/* */
/* */
/* */
/******/
/* Main program */
address system dir
```

Hint 4

Use the Parse statement to assign default values. An example for the same is shown below.

```
parse value 0 1 with
a,
b
```

Hint 5

Use the "Left(var1,2)" statement wherever possible instead of the "substr(var1,1,2)" statement.

Hint 6

Use the "Right(var1,2)" statement wherever possible instead of the "substr(var1,length(var1),2)" statement.

38. Rexx – Graphical User Interface

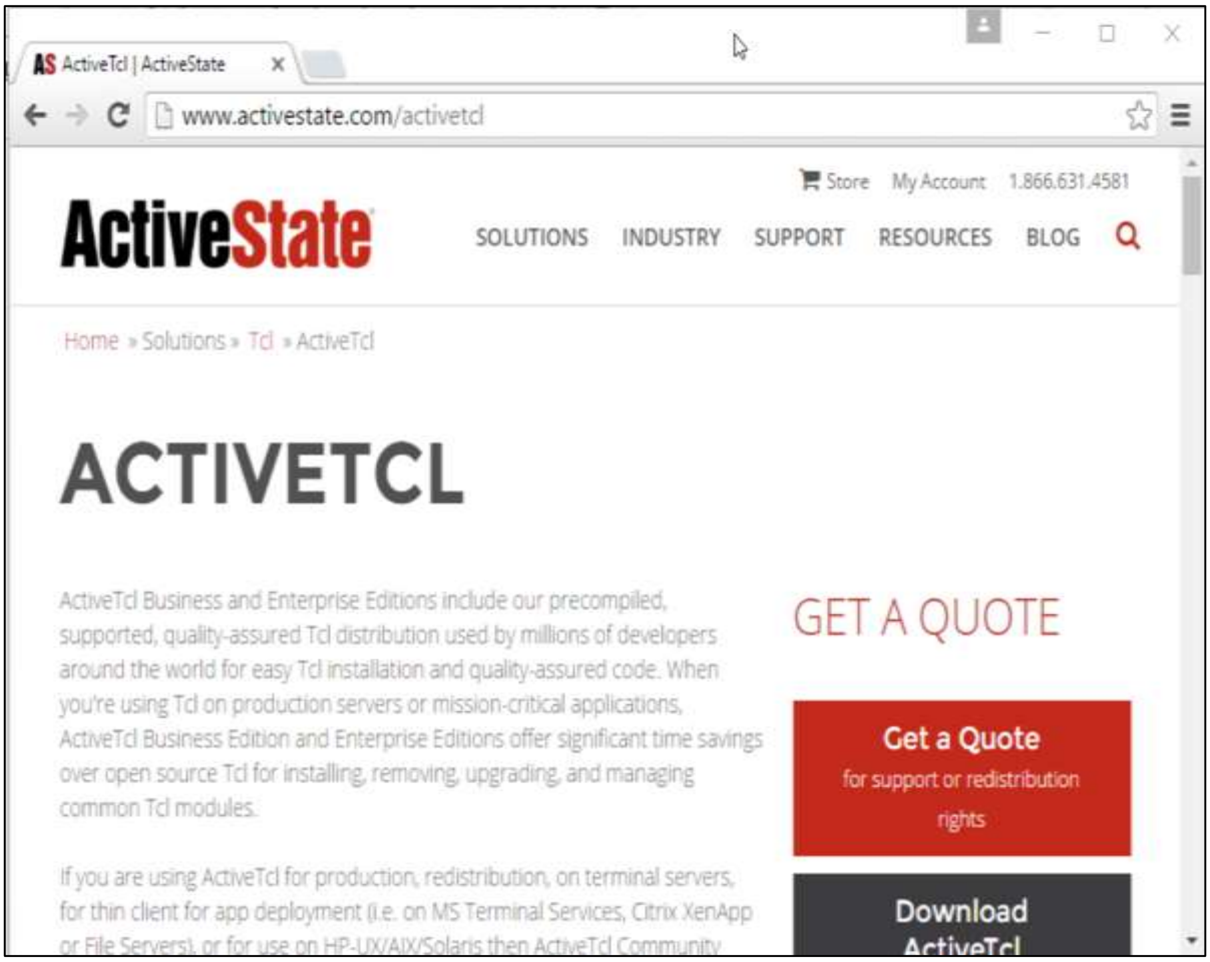
In order to use the graphic user interfaces available in Rexx, one needs to use 2 packages, one is called **ActiveTcl** and the other is the **Rexxtk** package. Along with these 2 packages, one can design normal forms which can have buttons and other controls on the forms.

Environment Setup

The first thing to do is the environment setup. Let's go through the following steps to have the environment in place.

Step 1: Download the Activetcl package from the following website –

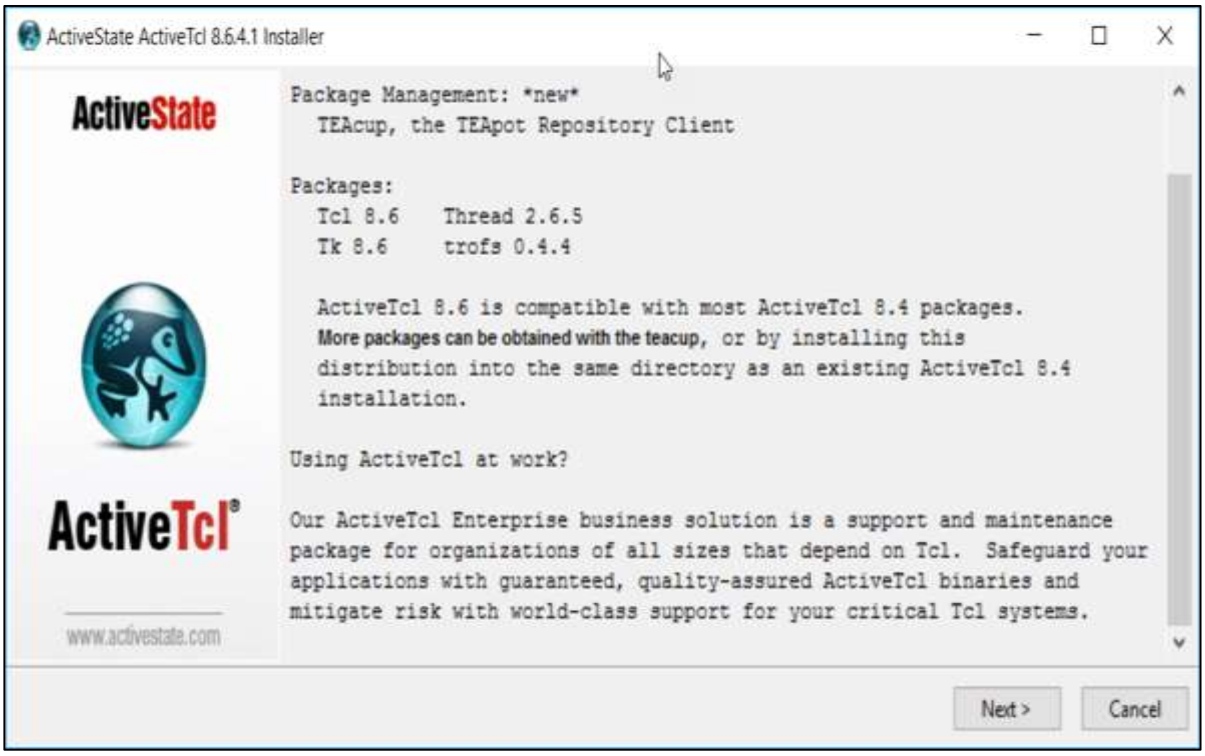
http://www.activestate.com/activetcl



ActiveTcl Business and Enterprise Editions include our precompiled, supported, quality-assured Tcl distribution used by millions of developers around the world for easy Tcl installation and quality-assured code. When you're using Tcl on production servers or mission-critical applications, ActiveTcl Business Edition and Enterprise Editions offer significant time savings over open source Tcl for installing, removing, upgrading, and managing common Tcl modules.

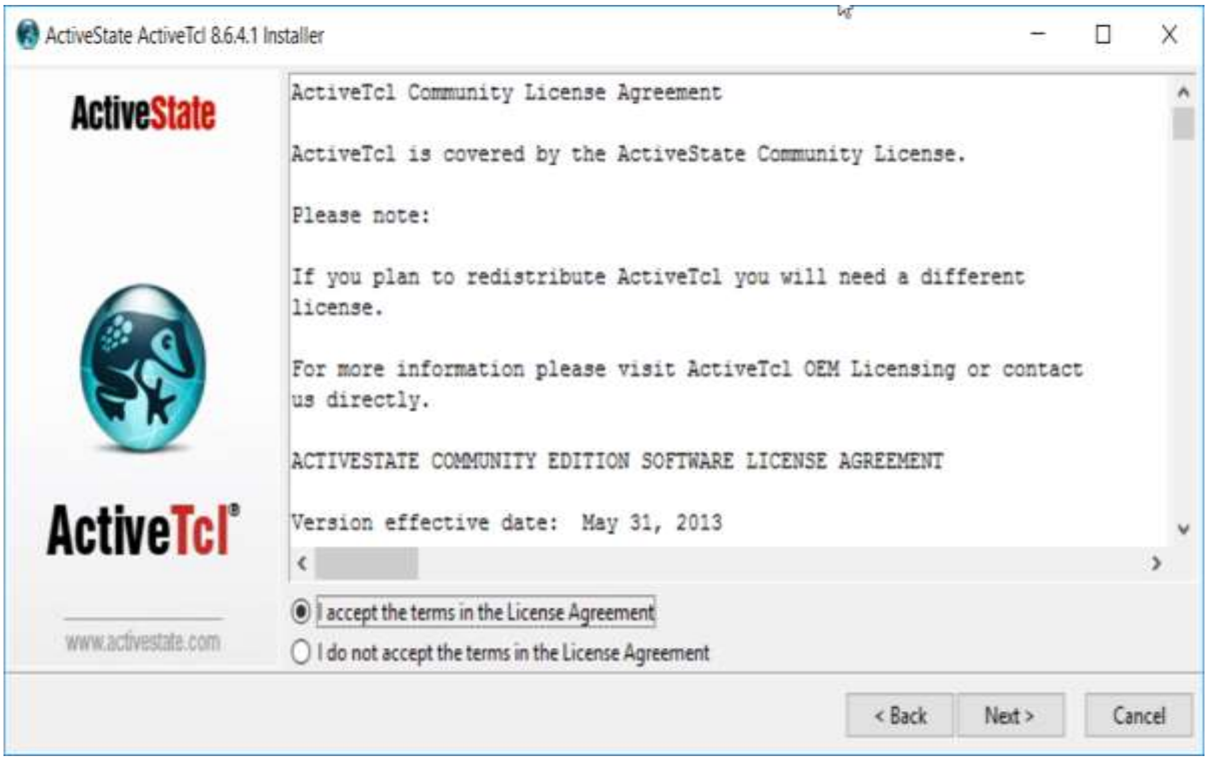
If you are using ActiveTcl for production, redistribution, on terminal servers, for thin client for app deployment (i.e. on MS Terminal Services, Citrix XenApp or File Servers), or for use on HP-UX/AIX/Solaris then ActiveTcl Community

Step 2: The next step is to start the installation of ActiveTCl. Click on the Next button on the screen to proceed.



```
Package Management: *new*
  TEAcup, the TEApot Repository Client

Packages:
  Tcl 8.6    Thread 2.6.5
  Tk 8.6     trofs 0.4.4

  ActiveTcl 8.6 is compatible with most ActiveTcl 8.4 packages.
  More packages can be obtained with the teacup, or by installing this
  distribution into the same directory as an existing ActiveTcl 8.4
  installation.

Using ActiveTcl at work?

Our ActiveTcl Enterprise business solution is a support and maintenance
package for organizations of all sizes that depend on Tcl.  Safeguard your
applications with guaranteed, quality-assured ActiveTcl binaries and
mitigate risk with world-class support for your critical Tcl systems.
```

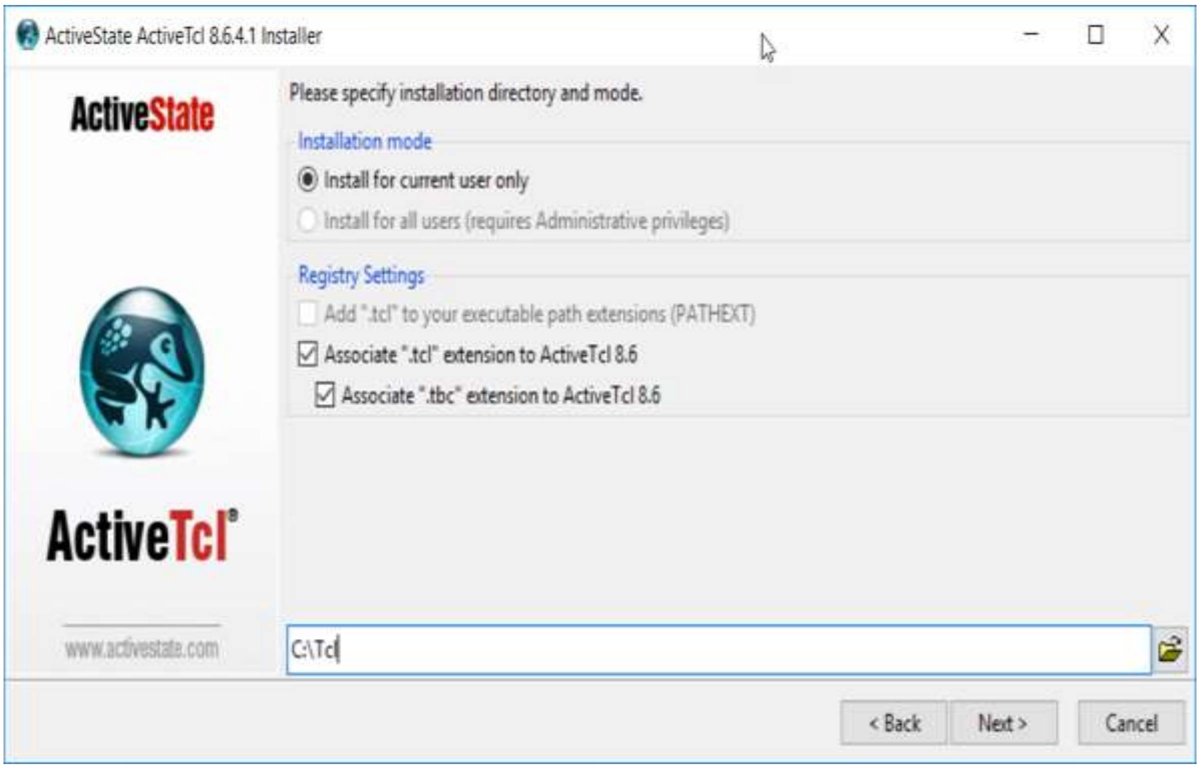
Step 3: Accept the license Agreement and click on the Next button.



```
ActiveTcl Community License Agreement

ActiveTcl is covered by the ActiveState Community License.

Please note:

If you plan to redistribute ActiveTcl you will need a different
license.

For more information please visit ActiveTcl OEM Licensing or contact
us directly.

ACTIVESTATE COMMUNITY EDITION SOFTWARE LICENSE AGREEMENT

Version effective date:  May 31, 2013
```

[✓ I accept the terms in the License Agreement]
I do not accept the terms in the License Agreement

Step 4: Choose a location for the installation and click on the next button.



ActiveState ActiveTcl 8.6.4.1 Installer

Please specify installation directory and mode.

Installation mode

[✓ Install for current user only]

Install for all users (requires Administrative privileges)

Registry Settings

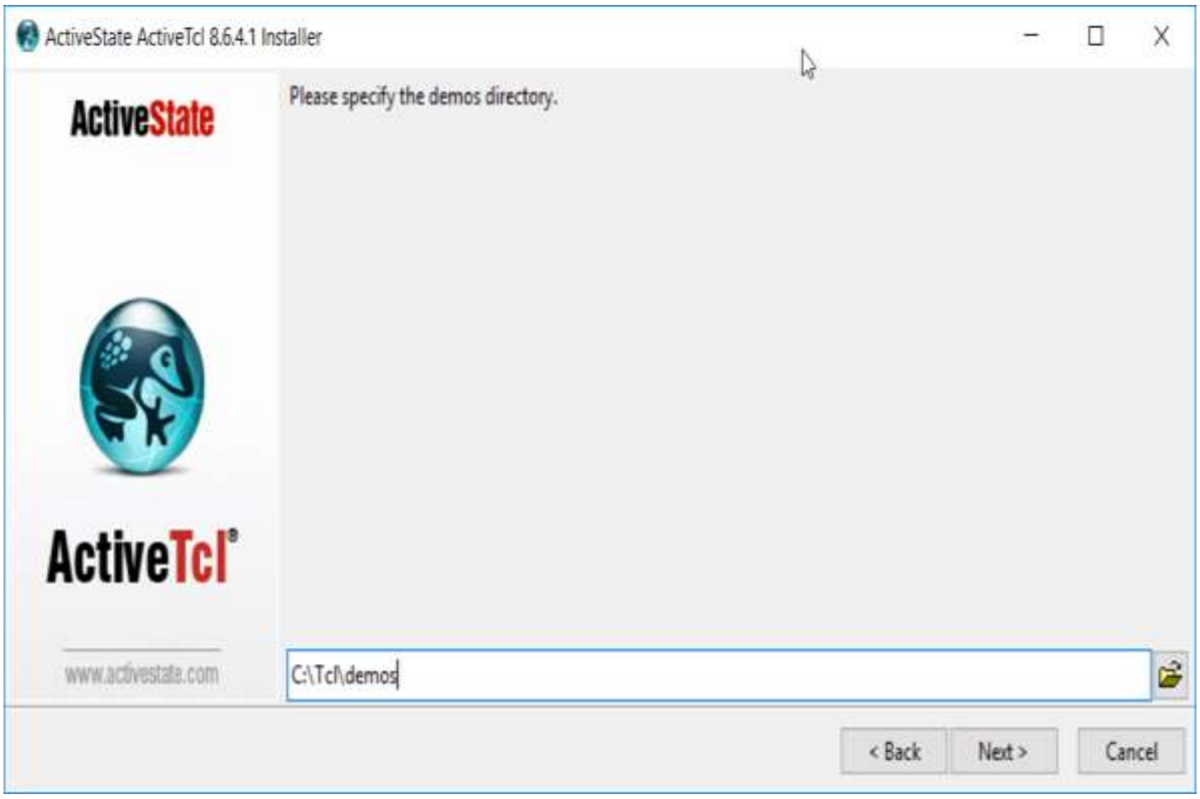
Add ".tcl" to your executable path extensions (PATHEXT)

[✓ Associate ".tcl" extension to ActiveTcl 8.6]

[✓ Associate ".tbc" extension to ActiveTcl 8.6]

C:\Tcl

Step 5: Choose a location for the installation of the demo's and click on the Next button.

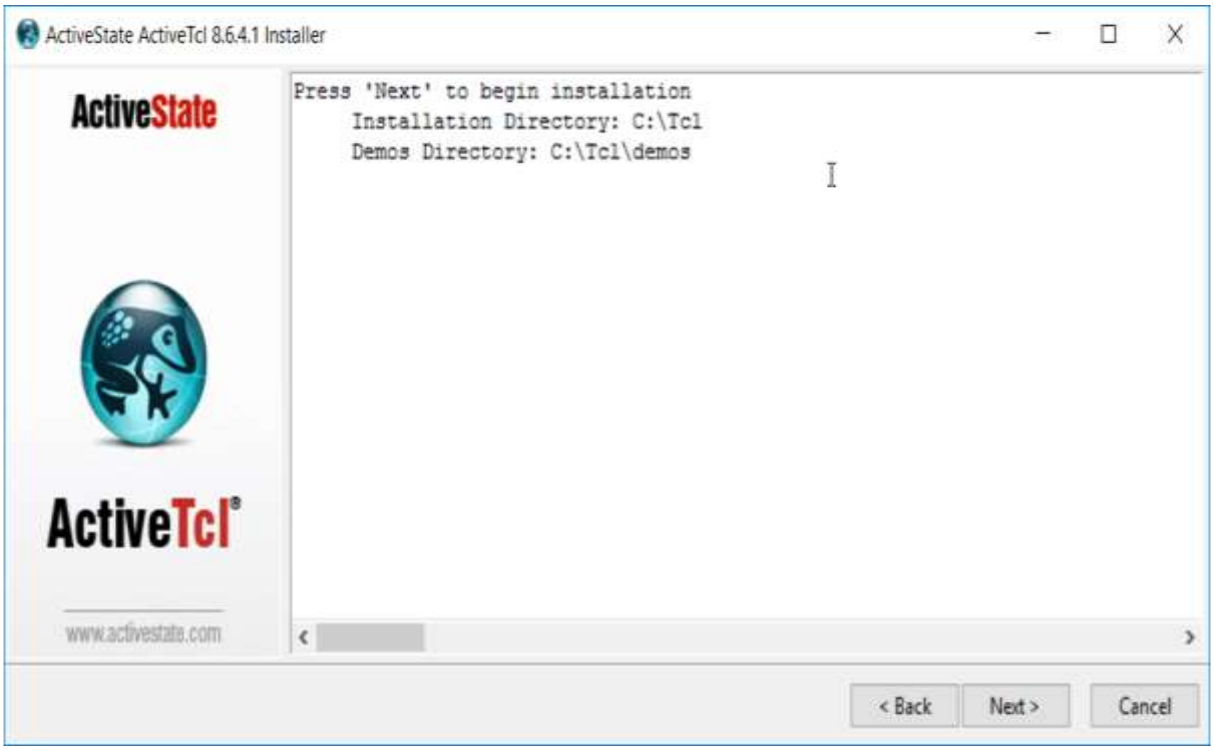


ActiveState ActiveTcl 8.6.4.1 Installer

Please specify the demos directory.

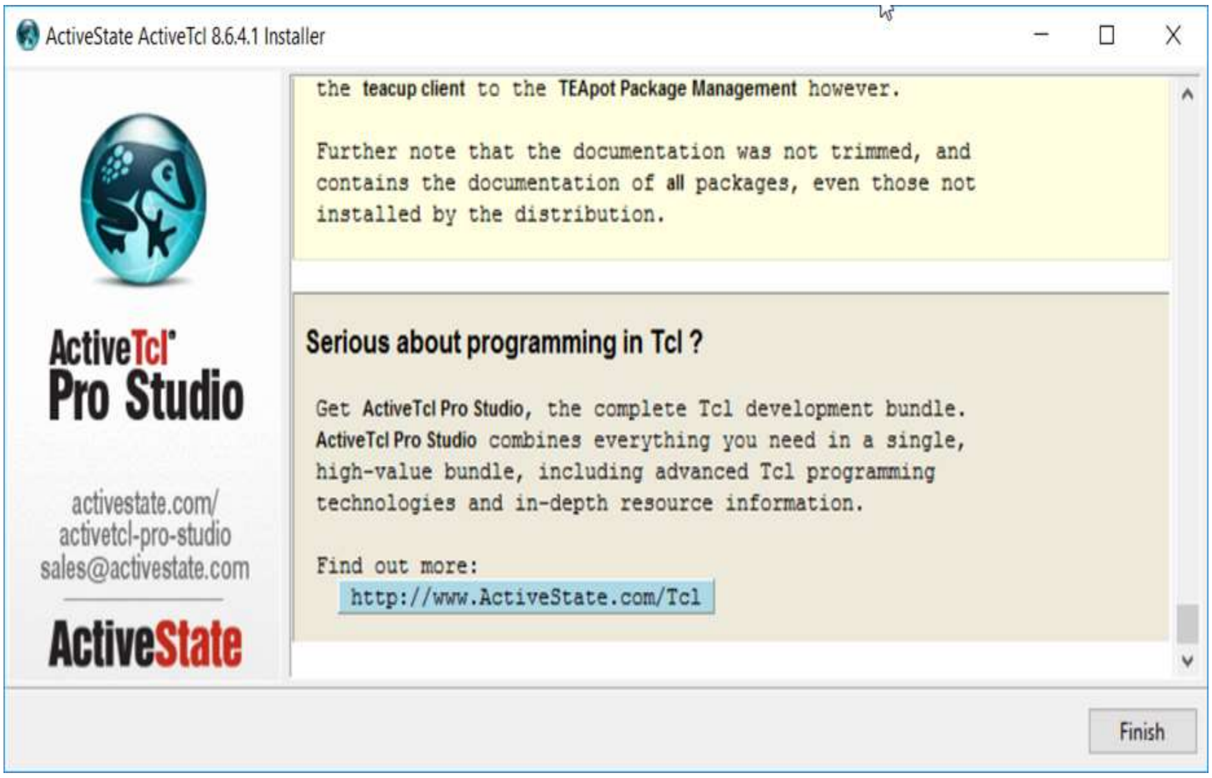
C:\Tcl\demos

Step 6: Click on the Next button to proceed with the installation.



```
Press 'Next' to begin installation
     Installation Directory: C:\Tcl
     Demos Directory: C:\Tcl\demos
```

Step 7: Click on the Finish button to complete the installation.



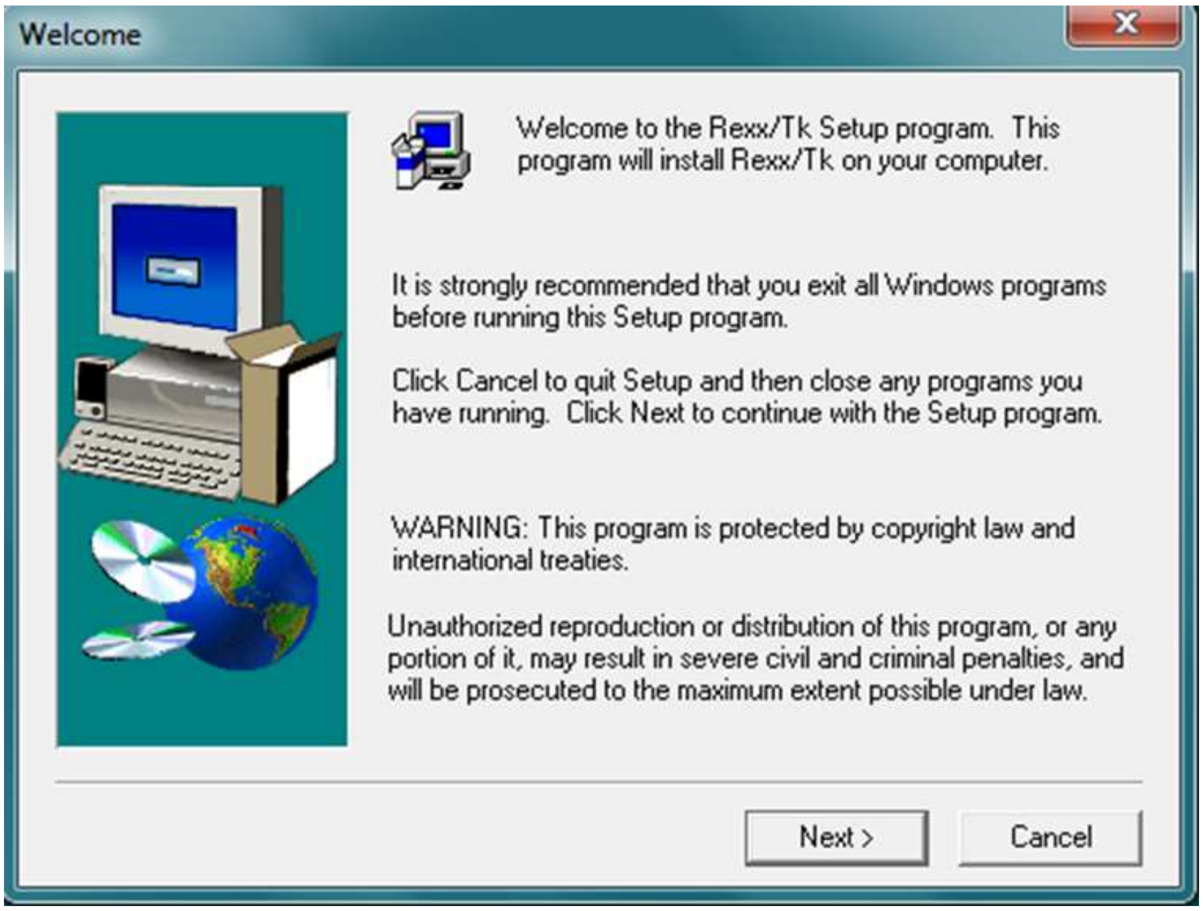
```
the teacup client to the TEApot Package Management however.

Further note that the documentation was not trimmed, and
contains the documentation of all packages, even those not
installed by the distribution.
```

Serious about programming in Tcl ?

```
Get ActiveTcl Pro Studio, the complete Tcl development bundle.
ActiveTcl Pro Studio combines everything you need in a single,
high-value bundle, including advanced Tcl programming
technologies and in-depth resource information.

Find out more:
   http://www.ActiveState.com/Tcl
```

Step 8: The next step is to download the Rexxtk software from the following link – https://sourceforge.net/projects/rexxtk/

Step 9: Double click the installer file from the link in the previous step to start the installation. Click on the next button to proceed.



Welcome to the Rexx/Tk Setup program. This program will install Rexx/Tk on your computer.

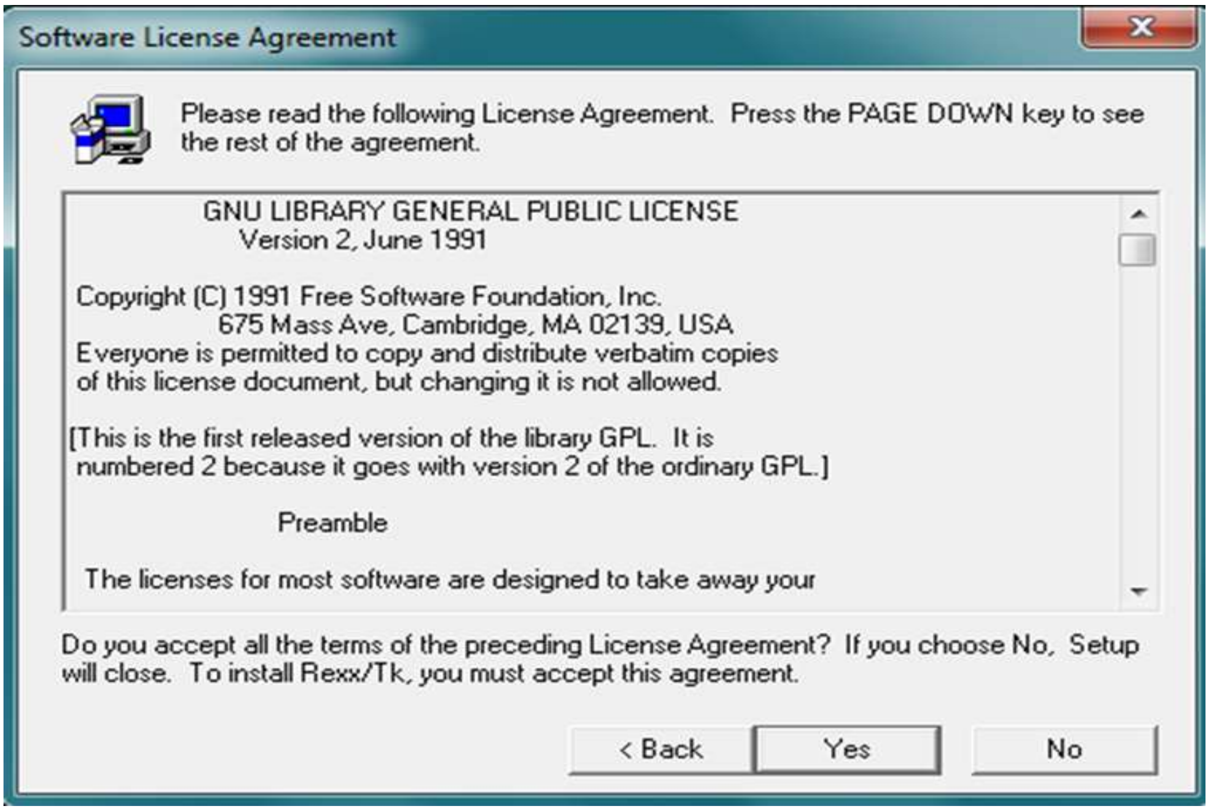
It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

Step 10: In the next screen, click on the Yes button to agree to the License Agreement.



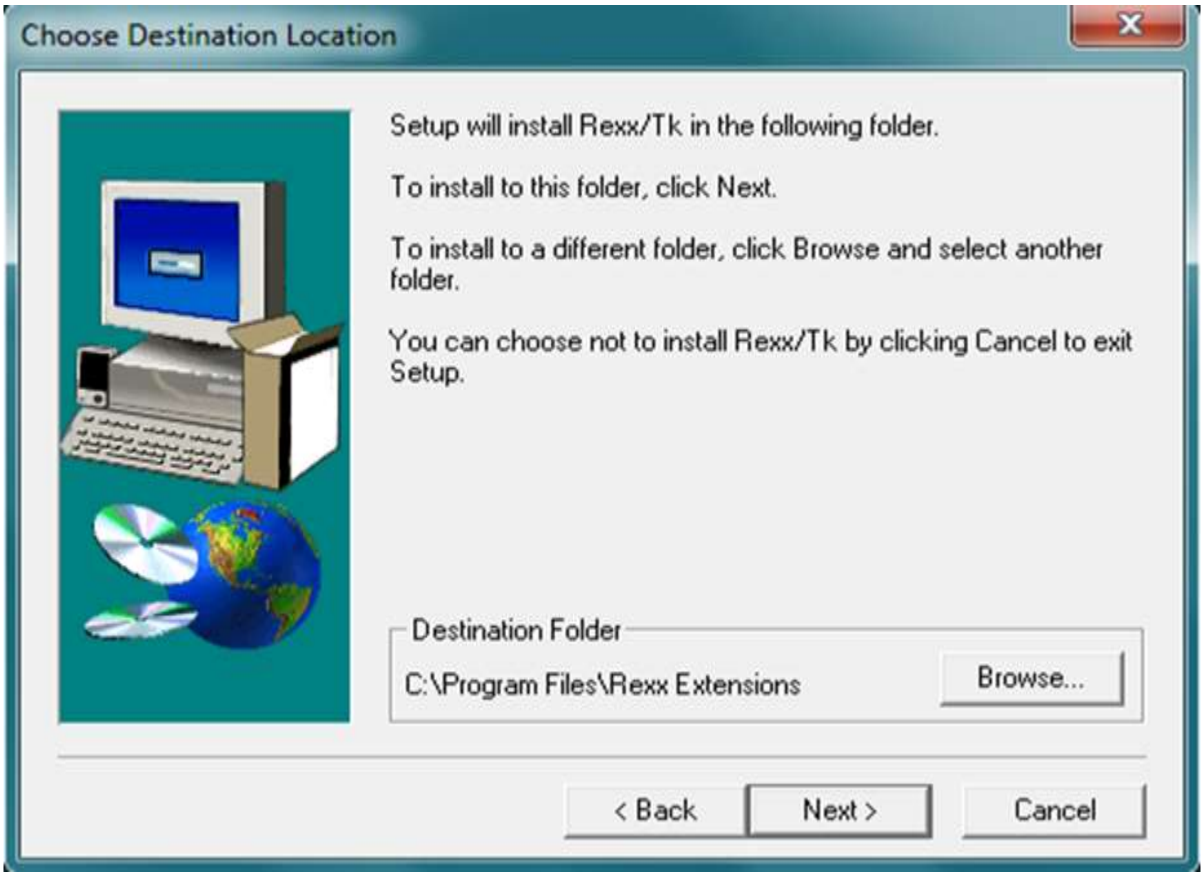
Software License Agreement

Please read the following License Agreement. Press the PAGE DOWN key to see the rest of the agreement.

```
                GNU LIBRARY GENERAL PUBLIC LICENSE
                     Version 2, June 1991

 Copyright (C) 1991 Free Software Foundation, Inc.
                    675 Mass Ave, Cambridge, MA 02139, USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the library GPL.  It is
 numbered 2 because it goes with version 2 of the ordinary GPL.]

                          Preamble

  The licenses for most software are designed to take away your
```

Do you accept all the terms of the preceding License Agreement? If you choose No, Setup will close. To install Rexx/Tk, you must accept this agreement.

Step 11: In the next screen, choose the location for the installation and click on the Next button.



Choose Destination Location

Setup will install Rexx/Tk in the following folder.

To install to this folder, click Next.

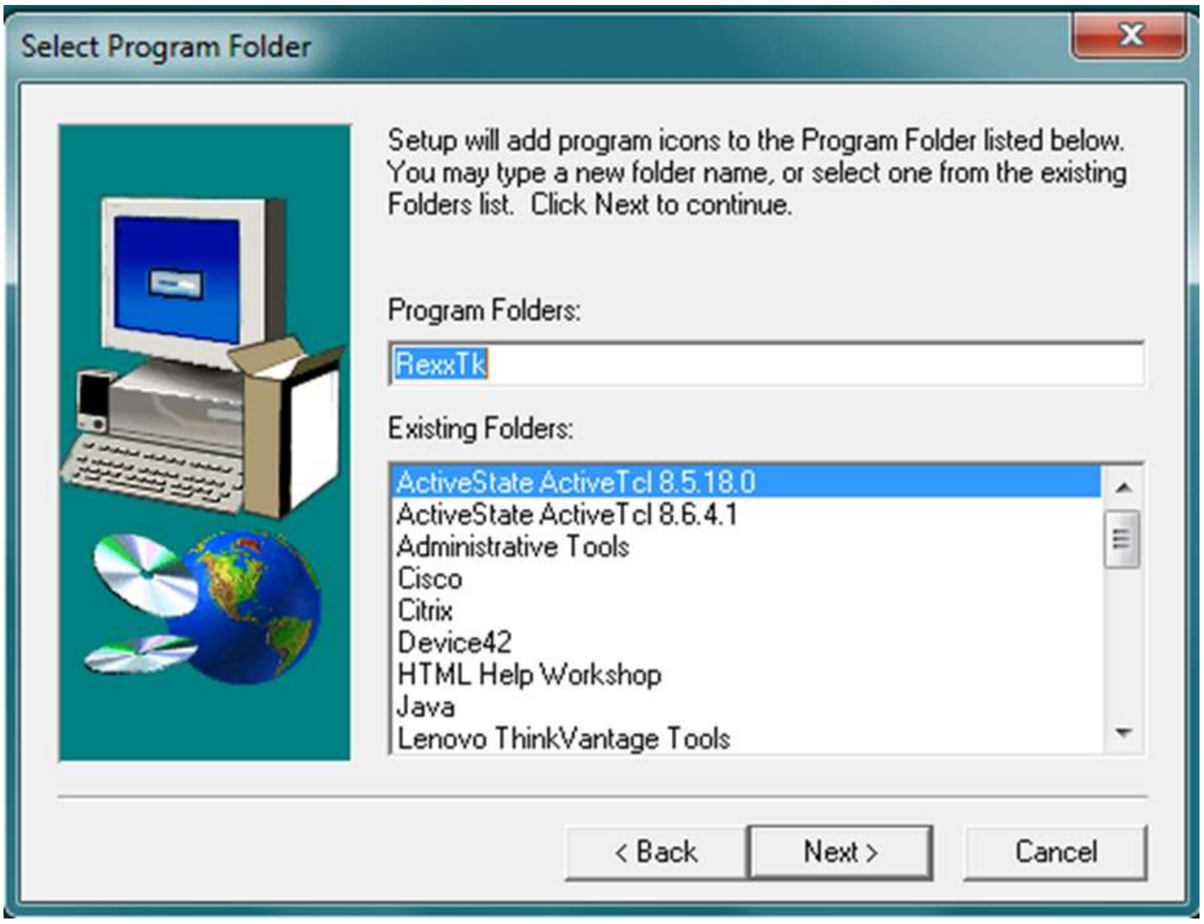
To install to a different folder, click Browse and select another folder.

You can choose not to install Rexx/Tk by clicking Cancel to exit Setup.

Destination Folder

C:\Program Files\Rexx Extensions

Step 12: Choose the Program folder location and click on the next button.


Select Program Folder

Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

Program Folders:

RexxTk

Existing Folders:

ActiveState ActiveTcl 8.5.18.0
ActiveState ActiveTcl 8.6.4.1
Administrative Tools
Cisco
Citrix
Device42
HTML Help Workshop
Java
Lenovo ThinkVantage Tools

Once the installation is complete, we can now start with programming the GUI's in Rexx.

Basic Program

Let's see how we can design a simple basic program with Rexx in a graphical user interface format.

Example:

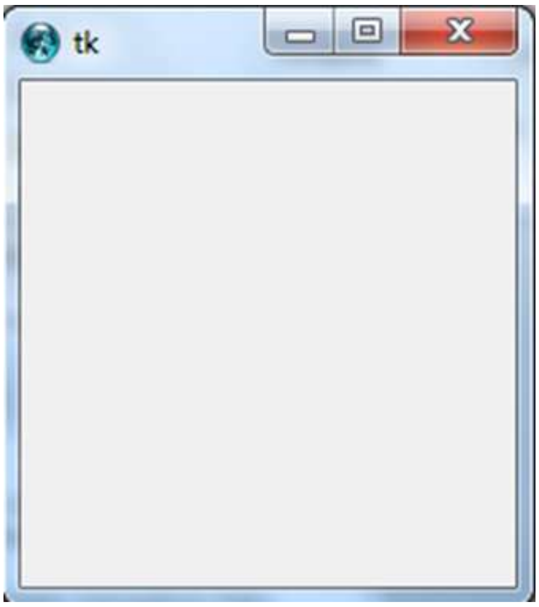
```
/* Main program */
call RxFuncAdd 'TkLoadFuncs','rexxtk','TkLoadFuncs'
call TkLoadFuncs
do forever
interpret 'Call' TkWait()
end
call TkDropFuncs
exit 0
```

The following things need to be noted about the above program:

- The Rexxtk library and all of its functions are loaded using the RxFuncAdd command.
- The do forever loop will keep the window open and will wait for the user input.

- Once the user input is detected, the program will exit.

Output: When the above program is executed, you will get the following output.



Creating Menus

Menus are created with the help of the TkMenu and TkAdd functions. The syntax of these functions are given below.

Syntax:

```
TkMenu(widgetname,options,0)
```

Parameters:

- **Widgetname –** A name to give to the menu.
- Options can be anyone of the following –
    - **selectcolor –** if checkboxes or radio buttons are used as menu options, then this option specifies the color to choose when any menu option is selected.
    - **tearoff –** This option is used for adding sub menus to the main menu.
    - **title –** The string that needs to be used to give the window a title.

Return Value: A handle to the menu created.

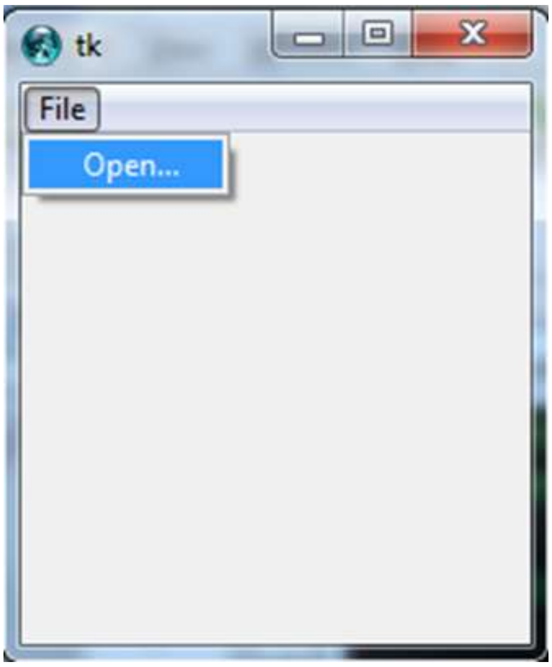
Example:

```
/* Main program */
call RxFuncAdd 'TkLoadFuncs','rexxtk','TkLoadFuncs'
call TkLoadFuncs
menubar = TkMenu('.m1')
filemenu = TkMenu('.m1.file','-tearoff', 0)
call TkAdd menubar, 'cascade', '-label', 'File', '-menu', filemenu
call TkAdd filemenu, 'command', '-label', 'Open...', '-rexx', 'getfile'
call TkConfig '.', '-menu', menubar
do forever
interpret 'Call' TkWait()
end
call TkDropFuncs
exit 0
```

The following things need to be noted about the above program:

- The menubar is created using the TkMenu function. The 'tearoff' parameter means that we need to create submenus which is going to be attached to the main menu.
- We then add 2 menu options called File and Open using the TkAdd function.

Output: When the above program is executed, you will get the following output.



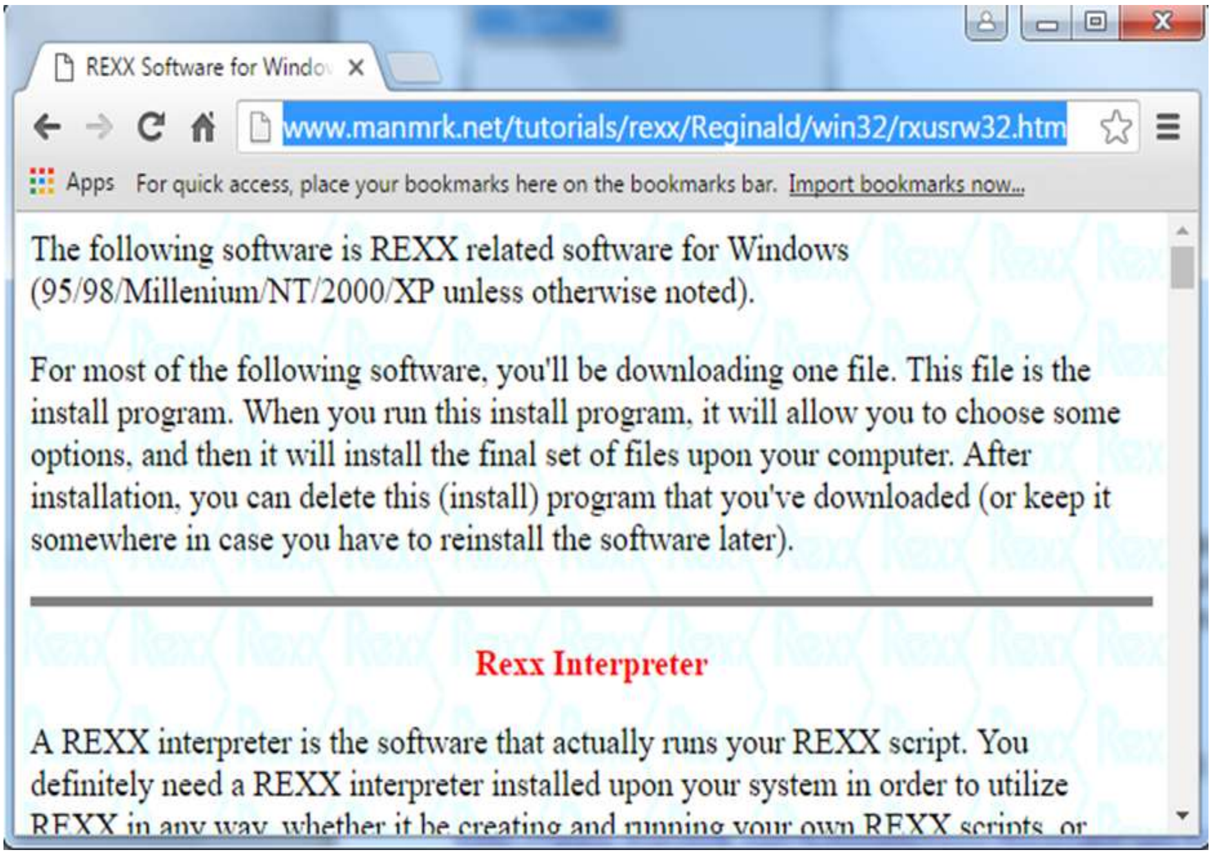
39. Rexx – Reginald

Reginald is another Rexx interpreter which was developed by Jeff Glantt and has some customizations on how Rexx programs can be run. In this section, we will see how to get Reginald setup and run a few Rexx programs in it.

Environment Setup

The first step is the environment setup which is to download the Reginald files. This can be done from the following website link –

http://www.manmrk.net/tutorials/rexx/Reginald/win32/rxusrw32.htm


REXX Software for Windo
www.manmrk.net/tutorials/rexx/Reginald/win32/rxusrw32.htm
Apps
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

The following software is REXX related software for Windows (95/98/Millenium/NT/2000/XP unless otherwise noted).

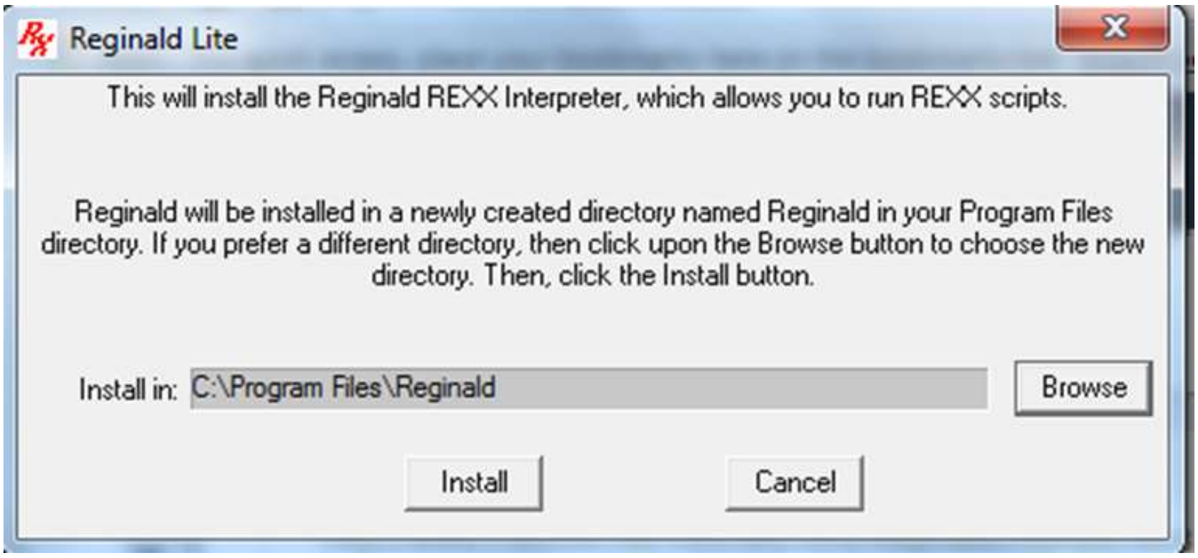
For most of the following software, you'll be downloading one file. This file is the install program. When you run this install program, it will allow you to choose some options, and then it will install the final set of files upon your computer. After installation, you can delete this (install) program that you've downloaded (or keep it somewhere in case you have to reinstall the software later).

Rexx Interpreter

A REXX interpreter is the software that actually runs your REXX script. You definitely need a REXX interpreter installed upon your system in order to utilize REXX in any way, whether it be creating and running your own REXX scripts, or

Once the download is complete and you launch the installer, the next screen will allow you to choose the install location.

Click the Install button to proceed.



This will install the Reginald REXX Interpreter, which allows you to run REXX scripts.

Reginald will be installed in a newly created directory named Reginald in your Program Files directory. If you prefer a different directory, then click upon the Browse button to choose the new directory. Then, click the Install button.

Install in: C:\Program Files\Reginald

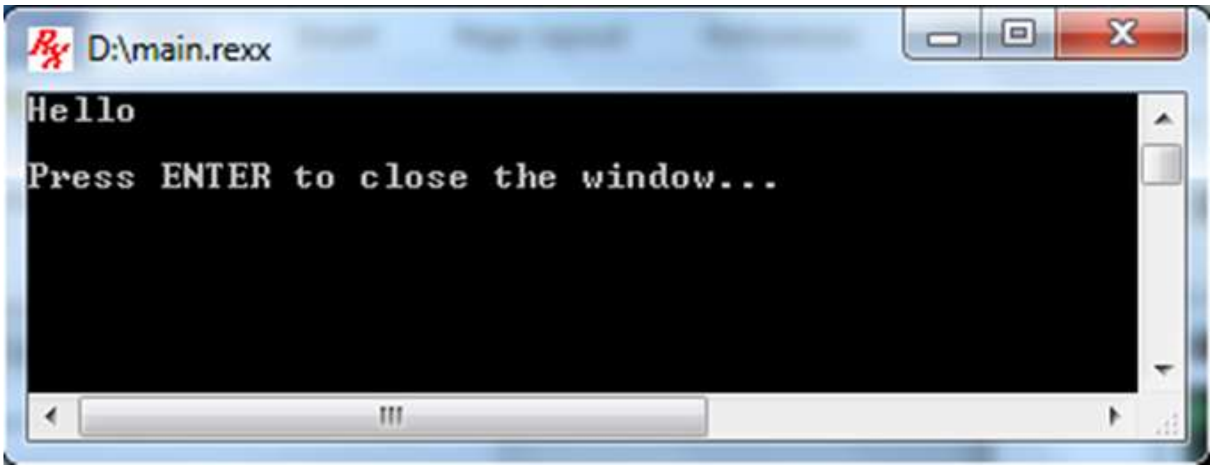
Once complete, we can now start to run one sample program in the Reginald interpreter. Create a simple program as shown below.

```
/* Main program */
say 'Hello'
```

Then run the following command –

```
RxLaunch.exe main.rexx
```

You will then get the following output. This program will now be running in the Reginald interpreter.



```
Hello
Press ENTER to close the window...
```

Other Functions Available

Apart from the normal Rexx commands, Reginald had some specific commands that are tailor-made for the Windows operating system. DriveMap is one such command:

DriveMap

This function gives information on the drive.

Syntax:

```
Drivemap(,options)
```

Parameters:

- **Options –** These are a list of keywords which can be used to get various information on the drives of the local computer.

Return Value: A string value which has information on the drive.

Example:

```
/* Main program */
say 'Drives on system : ' DriveMap(,'FIXED')
```

Output: If the above program is run, you will get the following output. This output depends from system to system.

```
List of disk drives :  C:\ D:\
```

40. Rexx – Web Programming

Rexx has the facility to work with web servers as well. The most common being the apache web server. In order to use Rexxw with the Apache web server, you need to first download the Rexx modules from the following link –

https://sourceforge.net/projects/modrexx/?source=typ_redirect

Once done, make sure to add the mod Rexx modules to the class path.

The following lines need to be added and modified to the Apache configuration file.

The following lines need to be added to the end of the appropriate:

- httpd.conf LoadModule list.
- LoadModule rexx_module modules/mod_rexx.dll

The following lines should be added at the end of the **http.conf** file.

- AddType application/x-httpd-rexx-script .rex .rexx
- AddType application/x-httpd-rexx-rsp .rsp
- Add these for REXX Server Page support
- RexxRspCompiler "c:/Program Files/Apache Group/Apache2/bin/rspcomp.rex"

Once the above changes have been made, you need to shut down and restart your apache web server.

The above lines also allow you to have Rexx based server pages just like Java server pages. You can add the Rexx code directly to the html pages.

An example is shown below:

```
<p>The current date and time is
<?rexx
/* Inserting the rexx statement */
say date() time()
?>
```

When a Rexx based server page is run, the following things are carried out:

- First a temporary file is created.
- Then the Rexx Server compiler compiles the file into a Rexx program and places it in the temporary file.
- The next step is to actually run the Rexx program.

Finally, the temporary file is removed.