NetRexx Programming Guide

RexxLA

Version 4.01-GA of March 20, 2021

THE REXX LANGUAGE ASSOCIATION
NetRexx Programming Series
ISBN 978-90-819090-0-6

Publication Data

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

This edition is registered under ISBN 978-90-819090-0-6

ISBN 978-90-819090-0-6

9 789081 909006 >

Contents

The NetREXX Programming Series

This book is part of a library, the *NetREXX Programming Series*, documenting the NetREXX programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

Quick Start Guide	This guide is meant for an audience that has done some programming and wants to start quickly. It starts with a quick tour of the language, and a section on installing the NetREXX translator and how to run it. It also contains help for troubleshooting if anything in the installation does not work as designed, and states current limits and restrictions of the open source reference implementation.
Programming Guide	The Programming Guide is the one manual that at the same time teaches programming, shows lots of examples as they occur in the real world, and explains about the internals of the translator and how to interface with it.
Language Reference	Referred to as the NRL, this is meant as the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementors.
Pipelines Guide & Reference	The Data Flow oriented companion to NetREXX, with its CMS Pipelines compatible syntax, is documented in this manual. It discusses running Pipes for NetREXX in the command shell and the Workspace, and has ample examples of defining your own stages in NetREXX.

Introduction

The Programming Guide is the book that has the broadest scope of the publications in the *NetRexx Programming Series*. Where the *Language Reference* and the *Quickstart Guide* need to be limited to a formal description and definition of the NetRexx language for the former, and a Quick Tour and Installation instructions for the latter, this book has no such limitations. It teaches programming, discusses computer language history and comparative linguistics, and shows many examples on how to make NetRexx work with diverse techologies as TCP/IP, Relational Database Management Systems, Messaging and Queuing (MQ™) systems, J2EE Containers as JBOSS™ and IBM WebSphere Application Server™, discusses various rich- and thin client Graphical User Interface Options, and discusses ways to use NetRexx on various operating platforms. For many people, the best way to learn is from examples instead of from specifications. For this reason this book is rich in example code, all of which is part of the NetRexx distribution, and tested and maintained.

Terminology

The *NetRexx Language Reference (NRL)* is the source of the definitive truth about the language. In this *Programming Guide*, terminology is sometimes used more loosely than required for the more formal approach of the NRL. For example, there is a fine line distinguishing *statement*, *instruction* and *clause*, where the latter is a more Rexx-like concept that is not often mentioned in relation to other languages (if they are not COBOL or SQL). While we try not to be confusing, *clause* and *statement* will be interchangibly used, as are *instruction* and *keyword instruction*.

Acknowledgements

As this book is a compendium of decades of Rexx and NetRexx knowledge, it stands upon the shoulders of many of its predecessors, many of which are not available in print anymore in their original form, or will never be upgraded or actualized; we are indebted to many anonymous[1] authors of IBM product documentation, and many others that we do know, and will thank in the following. If anyone knows of a name not mentioned here that should be, please be in

[1]because they are unacknowledged in the original publications

touch. Dave Woodman, thank you for your contributions to this guide. A big IOU goes out to Alan Sampson, who singlehandedly contributed more than one hundred NetRexx programming examples. The Redbook authors (Peter Heuchert, Frederik Haesbrouck, Norio Furukawa, Ueli Wahli, Kris Buelens, Bengt Heijnesson, Dave Jones and Salvador Torres) have provided some important documents that have shown, in an early stage, how almost everything on the JVM is better and easier done in NetRexx. Kermit Kiser also provided examples and did maintenance on the translator. Bill Finlason provided the Eclipse instructions. If anyone feels their copyright is violated, please do let us know, so we can properly attribute offending passages, or take them out.[2]

[2]As the usage of all material in this publication is quoted for educational use, and consists of short fragments, a fair use clause will apply in most jurisdictions.

1

Meet the REXX Family

1.1 Once upon a Virtual Machine

On the 22nd of March 1979, to be precise, Mike Cowlishaw of IBM had a vision of an easier to use command processor for VM, and wrote down a specification over the following days. VM™ (now called z/VM) is the original Virtual Machine operating system, stemming from an era in which time sharing was acknowledged to be the wave of the future and when systems as CTSS (on the IBM 704) and TSS (on the IBM 360 Family of computers) were early timesharing systems, that offered the user an illusion of having a large machine for their exclusive use, but fell short of virtualising the entire hardware. The CP/CMS system changed this; CP virtualised the hardware completely and CMS was the OS running on CP. CMS knew a succession of command interpreters, called EXEC, EXEC2 and REXX™ (originally REX - until it was found out, by the IBM legal department, that a product of another vendor had a similar name) - the EXEC roots are the explanation why some people refer to a NetREXX program as an "exec". As a prime example of a *backronym*, Rexx stands for "Restructured Extended Executor". It can be defended that REXX came to be as a reaction on EXEC2, but it must be noted that both command interpreters shipped around the same time. From 1988 on REXX was available on MVS/TSO and other systems, like DOS, Amiga and various Unix systems. REXX was branded the official SAA procedures language and was implemented on all IBM's Operating Systems; most people got to know REXX on OS/2. In the late eighties the Object-Oriented successor of Rexx, Object Rexx, was designed by Simon Nash and his colleagues in the IBM Winchester laboratory. REXX was thereafter known as Classic Rexx. Several open source versions of Classic REXX were made over the years, of which Regina is a good example.

1.2 Once upon another Virtual Machine

In 1995 Mike Cowlishaw ported Java™to OS/2™ and soon after started with an experiment to run REXX on the JVM™. With REXX generally considered the first of the general purpose scripting languages, NetREXX™ is the first alternative language for the JVM. The 0.50 release, from April 1996, contained the NetREXX runtime classes and a translator written in REXX but tokenized and turned into an OS/2 executable. The 1.00 release came available in January 1997 and contained a translator bootstrapped to NetREXX. The REXX string type that can also handle unlimited precision numerics is called Rexx in Java and NetRexx. Where Classic Rexx was positioned as a system *glue* language and application macro language, NetRexx is seen as the one language that does it all, delivering system level programs or large applications.

Release 2.00 became available in August 2000 and was a major upgrade, in which interpreted execution was added. Until that release, NetRexx only knew *ahead of time* compilation (AOT).

Mike Cowlishaw took early retirement from IBM in March 2010. IBM announced the transfer of NetRexx source code to the Rexx Language Association (RexxLA) on June 8, 2011, 14 years after the v1.0 release, and on the same day, it released the NetRexx source code to RexxLA under the ICU open source license. RexxLA shortly after released this as NetRexx 3.00 and has followed with updates.

1.3 Features of NetRexx

Ease of use The NetRexx language is easy to read and write because many instructions are meaningful English words. Unlike some lower level programming languages that use abbreviations, NetRexx instructions are common words, such as **say**, **ask**, **if...then...else**, **do...end**, and **exit**.

Free format There are few rules about NetRexx format. You need not start an instruction in a particular column, you can also skip spaces in a line or skip entire lines, you can have an instruction span many lines or have multiple instructions on one line, variables do not need to be pre-defined, and you can type instructions in upper, lower, or mixed case.

Convenient built-in functions NetRexx supplies built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations.

Easy to debug When a NetRexx exec contains an error, messages with meaningful explanations are displayed on the screen. In addition, the **trace** instruction provides a powerful debugging tool.

Interpreted The NetRexx language is an interpreted language. When a NetRexx exec runs, the language processor directly interprets each language statement, or translates the program in JVM bytecode.

Extensive parsing capabilities NetRexx includes extensive parsing capabilities for character manipulation. This parsing capability allows you to set up a pattern to separate characters, numbers, and mixed input.

Seamless use of JVM Class Libraries NetRexx can use any class, and class library for the JVM (written in Java or other JVM languages) in a seamless manner, that is, without the need for extra declarations or definitions in the source code.

2

Learning to program

2.1 Console Based Programs

One way that a computer can communicate with a user is to ask questions and then compute results based on the answers typed in. In other words, the user has a conversation with the computer. You can easily write a list of NetREXX instructions that will conduct a conversation. We call such a list of instructions a program. The following listing shows a sample NetREXX program. The sample program asks the user to give his name, and then responds to him by name. For instance, if the user types in the name Joe, the reply Hello Joe is displayed. Or else, if the user does not type anything in, the reply Hello stranger is displayed. First, we shall discuss how it works; then you can try it out for yourself.

```
/* A conversation */
say "Hello! What's your name?"
who=ask
if who = '' then say "Hello stranger"
else say "Hello" who
```

Briefly, the various pieces of the sample program are:

`/* ... */` A comment explaining what the program is about. Where REXX programs on several platforms must start with a comment, this is not a hard requirement for NetREXX anymore. Still, it is a good idea to start every program with a comment that explains what it does.

`say` An instruction to display Hello! What' s your name? on the screen.

`ask` An instruction to read the response entered from the keyboard and put it into the computer's memory.

`who` The name given to the place in memory where the user's response is put.

`if` An instruction that asks a question.

`who = ''` A test to determine if who is empty.

`then` A direction to execute the instruction that follows, if the tested condition is true.

`say` An instruction to display Hello stranger on the screen.

`else` An alternative direction to execute the instruction that follows, if the tested condition is not true. Note that in NetREXX, else needs to be on a separate line.

`say` An instruction to display Hello, followed by whatever is in who on the screen.

The text of your program should be stored on a disk that you have access to with the help of an *editor* program. On Windows, notepad or (notepad++), jEdit, X2 or SlickEdit are suitable candidates. On Unix based systems, including macOS, vim or emacs are plausible editors. If you are on z/VM or z/OS, XEDIT or ISPF/PDF are a given. More about editing NetREXX code in chapter 26.1, *Editor Support*, on page 90.

When the text of the program is stored in a file, let's say we called it `hello.nrx`, and you installed NetREXX as indicated in the *NetREXX QuickStart Guide*, we can run it with

```
    nrc -exec hello
```

and this will yield the result:

```
NetRexx portable processor, version NetRexx after3.01, build 1-20120406-1326
Copyright (c) RexxLA, 2011.  All rights reserved.
Parts Copyright (c) IBM Corporation, 1995,2008.
Program hello.nrx
===== Exec: hello =====
Hello! What's your name?
```

If you do not want to see the version and copyright message every time, which would be understandable, then start the program with:

```
nrc -exec -nologo hello
```

This is what happened when Fred tried it.

```
Program hello.nrx
===== Exec: hello =====
Hello! What's your name?
Fred
Hello Fred
```

The **ask** instruction paused, waiting for a reply. Fred typed Fred on the command line and, when he pressed the ENTER key, the **ask** instruction put the word Fred into the place in the computer's memory called "who". The **if** instruction asked, is "who" equal to nothing:

```
who = ''
```

meaning, is the value of "who" (in this case, Fred) equal to nothing:

```
"Fred = ''
```

This was not true; so, the instruction after `then` was not executed; but the instruction after `else`, was.

But when Mike tried it, this happened:

```
Program hello.nrx
===== Exec: hello =====
Hello! What's your name?
```

```
Hello stranger
Processing of 'hello.nrx' complete
```

Mike did not understand that he had to type in his name. Perhaps the program should have made it clearer to him. Anyhow, he just pressed ENTER. The **ask** instruction put ” (nothing) into the place in the computer’s memory called “who”. The **if** instruction asked, is:

```
who = ''
```

meaning, is the value of “who” equal to nothing:

```
'' = ''
```

In this case, it was true. So, the instruction after **then** was executed; but the instruction after **else** was not.

2.2 Comments in programs

When you write a program, remember that you will almost certainly want to read it over later (before improving it, for example). Other readers of your program also need to know what the program is for, what kind of input it can handle, what kind of output it produces, and so on. You may also want to write remarks about individual instructions themselves. All these things, words that are to be read by humans but are not to be interpreted, are called comments. To indicate which things are comments, use:

```
/* to mark the start of a comment
*/ to mark the end of a comment.
```

The `/*` causes the translator to stop compiling and interpreting; this starts again only after a `*/` is found, which may be a few words or several lines later. For example,

```
/* This is a comment. */
say text /* This is on the same line as the instruction */
/* Comments may occupy more
than one line. */
```

NetREXX also has line mode comments - those turn a line at a time into a comment. They are composed of two dashes (hyphens, in listings sometimes fused to a typographical *em dash* - remember that in reality they are two *n dashes*.

```
-- this is a line comment
```

2.3 Strings

When the translator sees a quote (either ” or ’) it stops interpreting or compiling and just goes along looking for the matching quote. The string of characters inside the quotes is used just as it is. Examples of strings are:

```
'Hello'
"Final result: "
```

If you want to use a quotation mark within a string you should use quotation marks of the other kind to delimit the whole string.

```
"Don't panic"
'He said, "Bother"'
```

There is another way. Within a string, a pair of quotes (of the same kind as was used to delimit the string) is interpreted as one of that kind.

```
'Don''t panic' (same as "Don't panic" )
 "He said, ""Bother""" (same as 'He said, "Bother"')
```

2.4 Clauses

Your NetRexx program consists of a number of *clauses*. A clause can be:

1. A *keyword instruction* that tells the interpreter to do something; for example,

   ```
   say  "the word"
   ```

   In this case, the interpreter will display the word on the user’s screen.
2. An *assignment*; for example,

   ```
   Message = 'Take care!'
   ```
3. A *null* clause, such as a completely blank line, or

   ```
       ;
   ```
4. A *method call instruction* which invokes a *method* from a *class*

   ```
   'hiawatha'.left(2)
   ```

2.5 When does a Clause End?

It is sometimes useful to be able to write more than one clause on a line, or to extend a clause over many lines. The rules are:

- Usually, each clause occupies one line.
- If you want to put more than one clause on a line you must use a semicolon (;) to separate the clauses.

- If you want a clause to span more than one line you must put a dash (hyphen) at the end of the line to indicate that the clause continues on the next line. If a line does not end in a dash, a semicolon is implied.

What will you see on the screen when this exec is run?

```
/* Example: there are six clauses in this program */ say "Everybody
   cheer!"
say "2"; say "4" ; say "6" ; say "8" ; say "Who do we" -
"appreciate?"
```

2.6 Long Lines

Ever since the days of the punch card images are over, the lines in program sources have become longer and longer, and with NetRexx being a free format language, there is no real technical reason to limit line length. Still, for readability and for ease access to words within a line, it is often indicated to keep lines relatively short and tidy. For this reason, the *continuation character* '-' can be used. This also makes it possible to split long literal strings over lines.

```
say 'good' -
'night'
```

This example will concatenate 'good' and 'night' with a space inbetween. When you want to avoid that, use the '||' concatenation operator.

```
say 'good' -
||'night'
```

2.7 Loops

We can go on and write clause after clause in a program source files, but some repetitive actions in which only a small change occurs, are better handled by the **loop** statement.

Imagine an assignment to neatly print out a table of exchange rates for dollars and euros for reference in a shop. We could of course make the following program:

```
say  1 'euro equals'  1  * 1.19 'dollars'
say  2 'euro equals'  2  * 1.19 'dollars'
say  3 'euro equals'  3  * 1.19 'dollars'
say  4 'euro equals'  4  * 1.19 'dollars'
say  5 'euro equals'  5  * 1.19 'dollars'
say  6 'euro equals'  6  * 1.19 'dollars'
say  7 'euro equals'  7  * 1.19 'dollars'
say  8 'euro equals'  8  * 1.19 'dollars'
say  9 'euro equals'  9  * 1.19 'dollars'
say 10 'euro equals' 10  * 1.19 'dollars'
```

This is valid, but imagine the alarming thought that the list is deemed a success and you are tasked with making a new one, but now with values up to 100. That will be a lot of typing.

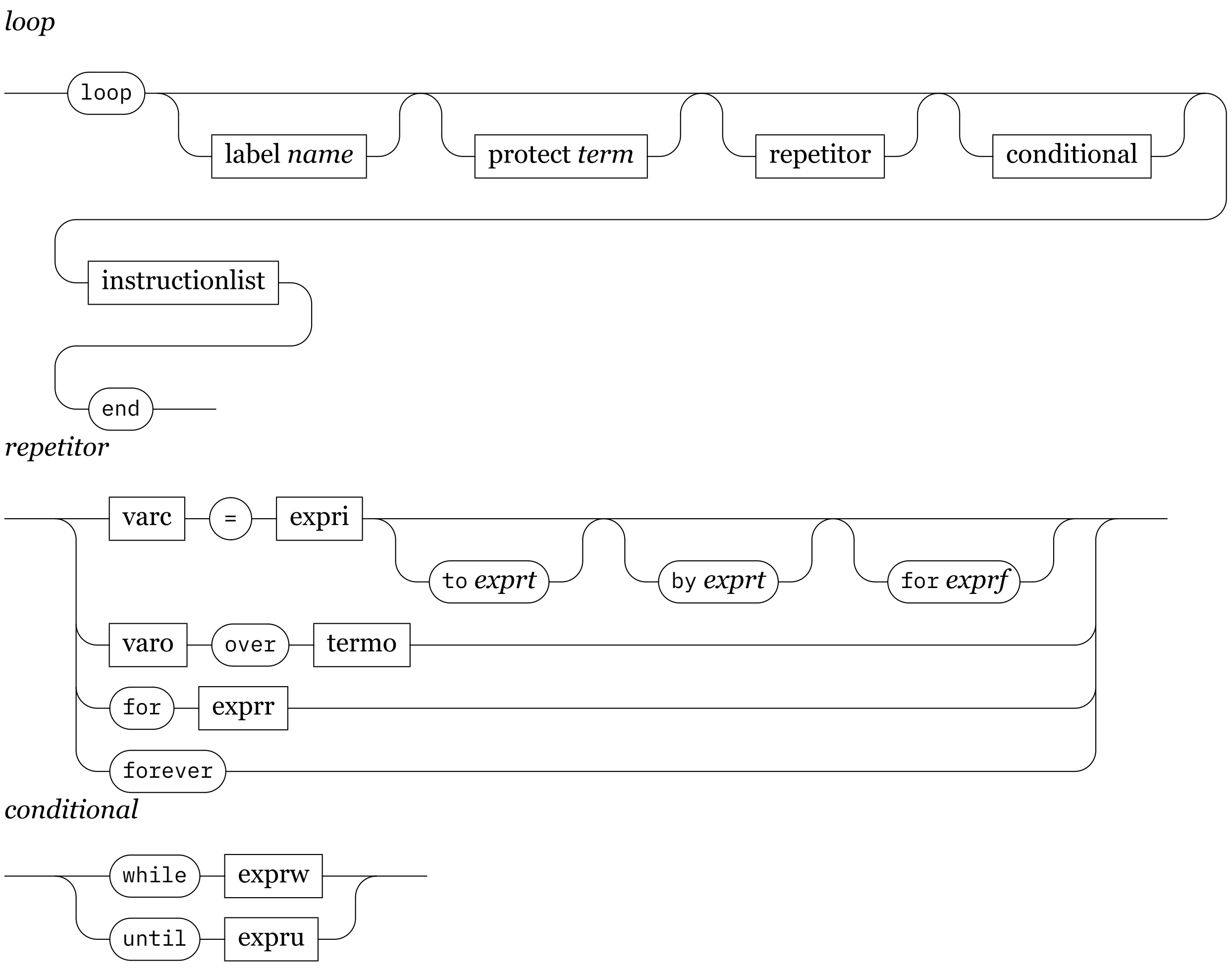
The way to do this is using the **loop**[3] statement.

```
loop i=1 to 100
  say i 'euro equals' i * 1.19 'dollars'
end
```

Now the *loop index variable* `i` varies from 1 to 100, and the statements between `loop` and `end` are repeated, giving the same list, but now from 1 to 100 dollars.

We can do more with the **loop** statement, it is extremely flexible. The following diagram is a (simplified, because here we left out the *catch* and *finally* options) rundown of the ways we can loop in a program.

FIGURE 1: Loop



A few examples of what we can do with this:

- Looping forever - better put, without deciding beforehand how many times

[3]Note that Classic REXX uses **do** for this purpose. In recent Open Object REXX versions **loop** can also be used.

```
loop forever
  say 'another bonbon?'
  x = ask
  if x = 'enough already' then leave
end
```

The `leave` statement breaks the program out of the loop. This seems futile, but in the chapter about I/O we will see how useful this is when reading files, of which we generally do not know in advance how many lines we will read in the loop.

- Looping for a fixed number of times without needing a loop index variable

```
loop for 10
   in.read() /* skip 10 lines from the input file */
end
```

- Looping back into the value of the loop index variable

```
loop i = 100 to 90 by -2
  say  i
end
```

This yields the following output:

```
===== Exec: test =====
100
98
96
94
92
90
Processing of 'test.nrx' complete
```

2.8 Special Variables

We have seen that a *variable* is a place where some data, be it character date or numerical data, can be held. There are some special variables, as shown in the following program.

```
/* NetRexx */
options replace format comments java symbols binary

class RCSpecialVariables

method RCSpecialVariables()
  x = super.toString
  y = this.toString
  say '<super>'x'</super>'
  say '<this>'y'</this>'
  say '<class>'RCSpecialVariables.class'</class>'
  say '<digits>'digits'</digits>'
  say '<form>'form'</form>'
  say '<[1, 2, 3].length>'
  say [1, 2, 3].length
  say '</[1, 2, 3].length>'
  say '<null>'
  say null
  say '</null>'
  say '<source>'source'</source>'
  say '<sourceline>'sourceline'</sourceline>'
  say '<trace>'trace'</trace>'
  say '<version>'version'</version>'

  say 'Type an answer:'
  say '<ask>'ask'</ask>'

  return

method main(args = String[]) public static

  RCSpecialVariables()

  return
```

`this` The special variables **this** and **super** refer to the current instance of the class and its superclass - what this means will be explained in detail in the chapter **Classes** on page 28, as is the case with the **class** variable.

`digits` The special variable **digits** shows the current setting for the number of decimal digits - the current setting of **numeric digits**. The related variable **form** returns the current setting of **numeric form** which is either `scientific` or `engineering`.

`null` The special variable **null** denotes the *empty reference*. It is there when a variable has no value.

`source` The **source** and **sourceline** variables are a good way to show the sourcefile and sourceline of a program, for example in an error message.

`trace` The **trace** variable returns the current trace setting, which can be one of the words `off var methods all results`.

`version` The **version** variable returns the version of the NetREXX translator that was in use at the time the clause we processed; in case of interpreted execution(see chapter 4 on 13, it returns the level of the current translator in use.

The result of executing this exec is as follows:

```
===== Exec: RCSpecialVariables =====
<super>RCSpecialVariables@4e99353f</super>
<this>RCSpecialVariables@4e99353f</this>
<class>class RCSpecialVariables</class>
<digits>9</digits>
<form>scientific</form>
<[1, 2, 3].length>
3
</[1, 2, 3].length>
<null>

</null>
```

```
<source>Java method RCSpecialVariables.nrx</source>
<sourceline>21</sourceline>
<trace>off</trace>
<version>NetRexx 3.02 27 Oct 2011</version>
Type an answer:
hello fifi
<ask>hello fifi</ask>
```

It might be useful to note here that these special variables are not fixed in the sense of that they are not *Reserved Variables*. NetRexx does not have reserved variables and any of these special variables can be used as an ordinary variable. However, when it is used as an ordinary variable, there is no way to retrieve the special behavior.

3

NetRexx as a Scripting Language

The term *scripting* is used here in the sense of using the programming language for quickly composed programs that interact with some application or environment to perform a number of simple tasks.

You can use NetRexx as a simple scripting language without having knowledge of, or using any of the features that is needed in a Java program that runs on the JVM - like defining a class name, and having a `main` method that is static and expects an array of String as its input.

Scripts can be written very fast. There is no overhead, such as defining a class, constructors and methods, and the programs contain only the necessary instructions. In this sense, a NetRexx script looks like an oo-version of a classic script, as the ceremonial aspects of defining class and method can be skipped. These will be automatically generated in the Java language source that is being generated for a script.

The scripting feature can be used for test purposes. It is an easy and convenient way of entering some statements and testing them. The scripting feature can also be used for the start sequence of a NetRexx application.

Scripts can be interpreted or compiled - there is no rule that a script needs to be interpreted. In interpreted mode, the edit-compile-run cycle is shortened, in the sense that there is no separate compilation step necessary and incremental editing and testing can be done very efficiently. In both cases, interpreted or compiled, the NetRexx translator adds the necessary syntactic overhead into the Java source to enable the JVM to execute the resulting program.

The scripting facility and its automatic generation of a class statement can lead to one surprising message when there is an error in the first part of the program: *class x already implied* when the automatically generated class statement (using the program file name) somehow clashes with the specified name that contains the error. When not in scripting mode, this error message nearly always indicates an error that occurred before the first class statement.

4

NetRexx as an Interpreted Language

In the JVM environment, compilation and interpretation are concepts that are not as straightforward as in other environments; JVM code is interpreted on several levels. When we are referring to *interpreted* NetRexx code, we indicate that there is no intermediate Java compilation step involved. A JVM .class file is always interpreted by the JVM runtime; the NetRexx translator is able to execute programs without generating either .java or .class files.

This enables a very quick edit-debug-run cycle, especially when combined with the command line feature that keeps the translator classes resident (the -prompt option), or one of the IDE plugins for NetRexx.

For NetRexx to deliver this functionality, the translator has been designed to have an analogous interpret facility for every code generation part.[4]

[4]This is the right order in which to explain this feature, because historically, the compiler was first (1996) and the interpretation facility was added later (in 2000) -(but not without an extensive redesign of the compiler).

5

Using the translator

This section of the document tells you how to use the translator package.

The NetRexx translator may be used as a compiler or as an interpreter (or it can do both in a single run, so parsing and syntax checking are only carried out once). It can also be used as simply a syntax checker.

When used as a compiler, the intermediate Java source code may be retained, if desired. Automatic formatting, and the inclusion of comments from the NetRexx source code are also options.

5.1 Using the translator as a compiler

The installation instructions for the NetRexx translator describe how to use the package to compile and run a simple NetRexx program (*hello.nrx*). When using the translator in this way (as a compiler), the translator parses and checks the NetRexx source code, and if no errors were found then generates Java source code. This Java code is then compiled into bytecodes (*.class* files) using a Java compiler, in a process called AOT compilation. By default, the *javac* compiler in the Java toolkit is used.

This section explains more of the options available to you when using the translator as a compiler.

5.2 The translator command

The translator is invoked by running a Java program (class) which is called

```
org.netrexx.process.NetRexxC
```

(`NetRexxC`, for short). This can be run by using the Java interpreter, for example, by the command:

```
java org.netrexx.process.NetRexxC
```

or by using a system-specific script (such as *NetRexxC.cmd.* or *nrc.bat*). In either case, the compiler invocation is followed by one or more file specifications (these are the names of the files containing the NetRexx source code for the programs to be compiled).

File specifications may include a path; if no path is given then NetRexxC will look in the current (working) directory for the file. NetRexxC will add the extension *.nrx* to input program names (file specifications) if no extension was given.

So, for example, to compile *hello.nrx* in the current directory, you could use any of:

```
java org.netrexx.process.NetRexxC hello
java org.netrexx.process.NetRexxC hello.nrx
NetRexxC hello.nrx
nrc hello
```

(the first two should always work, the last two require that the system-specific script be available). The resulting *.class* file is placed in the current directory, and the *.crossref* (cross-reference) file is placed in the same directory as the source file (if there are any variables and the compilation has no errors).

Here is an example of compiling two programs, one of which is in the directory *d:\myprograms*:

```
nrc hello d:\myprograms\test2.nrx
```

In this case, again, the *.class* file for each program is placed in the current directory.

Note that when more than one program is specified, they are all compiled within the same class context. That is, they can see the classes, properties, and methods of the other programs being compiled, much as though they were all in one file. [5] This allows mutually interdependent programs and classes to be compiled in a single operation. Note that if you use the **package** instruction you should also read the more detailed *Compiling multiple programs* section.

On completion, the NetRexxC class will exit with one of three return values: 0 if the compilation of all programs was successful, 1 if there were one or more Warnings, but no errors, and 2 if there were one or more Errors. The result can be forced to 0 for warnings only with the *-warnexit0* option.

As well as file names, you can also specify various option words, which are distinguished by the word being prefixed with -. These flagged words (or flags) may be any of the option words allowed on the NetRexx **options** instruction (see the NetRexx languagen documentation, and the below paragraph). These options words can be freely mixed with file specifications. To see a full list of options, execute the NetRexxC with the –help option command without specifying any files. As this command states, all options may have prefix 'no' added for the inverse effect.

5.2.1 Options

Here are some examples:

[5]The programs do, however, maintain their independence (that is, they may have different **options**, **import**, and **package** instructions).

```
java org.netrexx.process.NetRexxC hello -keep -strictargs
java org.netrexx.process.NetRexxC -keep hello wordclock
java org.netrexx.process.NetRexxC hello wordclock -nocompile
nrc hello
nrc hello.nrx
nrc -run hello
nrc -run Spectrum -keep
nrc hello -binary -verbose1
nrc hello -noconsole -savelog -format -keep
```

Option words may be specified in lowercase, mixed case, or uppercase. File specifications are platform-dependent and may be case sensitive, though NetRexxC will always prefer an exact case match over a mismatch.

Note: The *-run* option is implemented by a script (such as *nrc.bat* or *NetRexxC.cmd*), not by the translator; some scripts (such as the *.bat* scripts) may require that the *-run* be the first word of the command arguments, and/or be in lowercase. They may also require that only the name of the file be given if the *-run* option is used. Check the commentary at the beginning of the script for details.

5.3 Compiling multiple programs and using packages

When you specify more than one program for NetRexxC to compile, they are all compiled within the same class context: that is, they can see the classes, properties, and methods of the other programs being compiled, much as though they were all in one file.

This allows mutually interdependent programs and classes to be compiled in a single operation. For example, consider the following two programs (assumed to be in your current directory, as the files *X.nrx* and *Y.nrx*):

```
/* X.nrx */
class X
  why=Y null

/* Y.nrx */
class Y
  exe=X null
```

Each contains a reference to the other, so neither can be compiled in isolation. However, if you compile them together, using the command:

```
nrc X Y
```

the cross-references will be resolved correctly.

The total elapsed time will be significantly less, too, as the classes on the CLASSPATH need to be located only once, and the class files used by the NetRexxC compiler or the programs themselves will also only be loaded (and JIT-compiled) once.

This example works as you would expect for programs that are not in packages. There is a restriction, though, if the classes you are compiling *are* in packages (that is, they include a **package** instruction). NetRexxC uses either the *javac* compiler or the Eclipse batch compiler *ecj* to generate the *.class* files, and for mutually-dependent files like these; both require the source files to be in the Java *CLASSPATH*, in the sub-directory described by the **package** instruction.

So, for example, if your project is based on the tree:

```
D:\myproject
```

if the two programs above specified a package, thus:

```
/* X.nrx */
package foo.bar
class X
  why=Y null

/* Y.nrx */
package foo.bar
class Y
  exe=X null
```

1. You should put these source files in the directory: *D:\myproject\foo\bar*
2. The directory *D:\myproject* should appear in your CLASSPATH setting (if you don't do this, *javac* will complain that it cannot find one or other of the classes).
3. You should then make the current directory be *D:\myproject\foo\bar* and then compile the programs using the command *nrc X Y*, as above.

With this procedure, you should end up with the *.class* files in the same directory as the *.nrx* (source) files, and therefore also on the CLASSPATH and immediately usable by other packages. In general, this arrangement is recommended whenever you are writing programs that reside in packages.

Notes:

1. When *javac* is used to generate the *.class* files, no new *.class* files will be created if any of the programs being compiled together had errors - this avoids accidentally generating mixtures of new and old *.class* files that cannot work with each other.
2. If a class is abstract or is an adapter class then it should be placed in the list before any classes that extend it (as otherwise any automatically generated methods will not be visible to the subclasses).

6

Using the NetRexxA API

As described elsewhere, the simplest way to use the NetRexx interpreter is to use the command interface (NetRexxC) with the *-exec* or *-arg* flags. There is a also a more direct way to use the interpreter when calling it from another NetRexx (or Java) program, as described here. This way is called the *NetRexxA Application Programming Interface* (API).
The *NetRexxA* class is in the same package as the translator (that is, *org.netrexx.process*), and comprises a constructor and two methods. To interpret a NetRexx program (or, in general, call arbitrary methods on interpreted classes), the following steps are necessary:

1. Construct the interpreter object by invoking the constructor *NetRexxA()*. At this point, the environment's classpath is inspected and known compiled packages and extensions are identified.
2. Decide on the program(s) which are to be interpreted, and invoke the NetRexxA *parse* method to parse the programs. This parsing carries out syntax and other static checks on the programs specified, and prepares them for interpretation. A stub class is created and loaded for each class parsed, which allows access to the classes through the JVM reflection mechanisms.
3. At this point, the classes in the programs are ready for use. To invoke a method on one, or construct an instance of a class, or array, etc., the Java reflection API (in *java.lang* and *java.lang.reflect*) is used in the usual way, working on the *Class* objects created by the interpreter. To locate these *Class* objects, the API's *getClassObject* method must be used.

Once step 2 has been completed, any combination or repetition of using the classes is allowed. At any time (provided that all methods invoked in step 3 have returned) a new or edited set of source files can be parsed as described in step 2, and after that, the new set of class objects can be located and used. Note that operation is undefined if any attempt is made to use a class object that was located before the most recent call to the *parse* method.
Here's a simple example, a program that invokes the *main* method of the *hello.nrx* program's class:

```
options binary
import org.netrexx.process.\nr{}A

interpreter=\nr{}A()             -- make interpreter

files=['hello.nrx']                -- a file to interpret
flags=['nocrossref', 'verbose0']   -- flags, for example
```

```
interpreter.parse(files, flags)    -- parse the file(s), using the
   flags

helloClass=interpreter.getClassObject(null, 'hello') -- find the
   hello Class

-- find the 'main' method; it takes an array of Strings as its
   argument
classes=[interpreter.getClassObject('java.lang', 'String', 1)]
mainMethod=helloClass.getMethod('main', classes)

-- now invoke it, with a null instance (it is static) and an empty
   String array
values=[Object String[0]]

loop for 10    -- let's call it ten times, for fun...
  mainMethod.invoke(null, values)
end
```

Compiling and running (or interpreting!) this example program will illustrate some important points, especially if a **trace all** instruction is added near the top. First, the performance of the interpreter (or indeed the compiler) is dominated by JVM and other start-up costs; constructing the interpreter is expensive as the classpath has to be searched for duplicate classes, etc. Similarly, the first call to the parse method is slow because of the time taken to load, verify, and JIT-compile the classes that comprise the interpreter. After that point, however, only newly-referenced classes require loading, and execution will be very much faster.
The remainder of this section describes the constructor and the two methods of the NetREXXA class in more detail.

6.1 The NetRexxA constructor

```
NetRexxA()
```

This constructor takes no arguments and builds an interpeter object. This process includes checking the classpath and other libraries known to the JVM and identifying classes and packages which are available.

6.2 The parse method

```
parse(files=String[], flags=String[]) returns boolean
```

The parse method takes two arrays of Strings. The first array contains a list of one or more file specifications, one in each element of the array; these specify the files that are to be parsed and made ready for interpretation.
The second array is a list of zero or more option words; these may be any option words understood by the interpreter (but excluding those known only to the NetREXXC command interface, such as *time*). [6] The parse method prefixes the

[6]Note that the option words are not prefixed with a -.

nojava flag automatically, to prevent *.java* files being created inadvertently. In the example, *nocrossref* is supplied to stop a cross-reference file being written, and *verbose0* is added to prevent the logo and other progress displays appearing.
The *parse* method returns a boolean value; this will be 1 (true) if the parsing completed without errors, or 0 (false) otherwise. Normally a program using the API should test this result an take appropriate action; it will not be possible to interpret a program or class whose parsing failed with an error.

6.3 The getClassObject method

```
getClassObject(package=String, name=String [,dimension=int]) returns
   Class
```

This method lets you obtain a Class object (an object of type *java.lang.Class*) representing a class (or array) known to the interpreter, including those newly parsed by a parse instruction.
The first argument, *package*, specifies the package name (for example, *com.ibm.math*). For a class which is not in a package, *null* should be used (not the empty string, ”).
The second argument, *name*, specifies the class name (for example, *BigDecimal*). For a minor (inner) class, this may have more than one part, separated by dots.
The third, optional, argument, specifies the number of dimensions of the requested class object. If greater than zero, the returned class object will describe an array with the specified number of dimensions. This argument defaults to the value 0.
An example of using the *dimension* argument is shown above where the *java.lang.String[]* array Class object is requested.
Once a Class object has been retrieved from the interpreter it may be used with the Java reflection API as usual. The Class objects returned are only valid until the parse method is next invoked.

7

Using NetREXX for Web applets

Web applets can be written one of two styles:

- Lean and mean, where binary arithmetic is used, and only core Java classes (such as *java.lang.String*) are used. This is recommended for World Wide Web pages, which may be accessed by people using a slow internet connection. Several examples using this style are included in the NetREXX package (eg., *NervousTexxt.nrx* or *ArchText.nrx*).
- Full-function, where decimal arithmetic is used, and advantage is taken of the full power of the NetREXX runtime (Rexx) class. This is appropriate for intranets, where most users will have fast connections to servers. An example using this style is included in the NetREXX package (*WordClock.nrx*).

If you write applets which use the NetREXX runtime (or any other Java classes that might not be on the client browser), the rest of this section may help in setting up your Web server.

A good way of setting up an HTTP (Web) server for this is to keep all your applets in one subdirectory. You can then make the NetREXX runtime classes (that is, the classes in the package known to the Java Virtual Machine as *netrexx.lang*) available to all the applets by unzipping NetREXXR.jar into a subdirectory *netrexx/lang* below your applets directory.
For example, if the root of your server data tree is

```
D:\mydata
```

you might put your applets into

```
D:\mydata\applets
```

and then the NetREXX classes (unzipped from NetREXXR.jar) should be in the directory

```
D:\mydata\applets\netrexx\lang
```

The same principle is applied if you have any other non-core Java packages that you want to make available to your applets: the classes in a package called *iris.sort.quicksorts* would go in a subdirectory below *applets* called *iris/sort/quicksorts*, for example.

Note that since Java 1.1 or later it is possible to use the classes direct from the NetREXXR.jar file. Please see the Java documentation for details.

8

Calling non-JVM programs

Non-JVM programs can be called using the `Address` instruction. For optimal flexibility in the handling of output, this sections describes how to use the native Java facilities for this. It is easy to call non-JVM programs from a NetRexx program - not as easy as calling a JVM class of course, but if the following recipe is observed, it will show not to be a major problem. The following example is reusable for many cases.

```
/* script NonJava.nrx

   This program starts an UNZIP program, redirects its output,
   parses the output and shows the files stored in the zipfile */

parse arg unzip zipfile .

-- check the arguments - show usage comments
if zipfile = '' then do
   say 'Usage: Process unzipcommand zipfile'
   exit 2
end

do
   say "Files stored in" zipfile
   say "-".left(39,"-") "-".left(39,"-")
   child = Runtime.getRuntime().exec(unzip ' -v' zipfile) -- program
      start

   -- read input from child process
   in   = BufferedReader(InputStreamReader(child.getInputStream()))
   line = in.readline

   start = 0    -- listing of files are not available yet
   count = 0
   loop while line \= null
      parse line sep  program
      if sep = '-------' then start = \start
      else
        if start then do
           count = count + 1
           if count // 2 > 0 then say program.word(program.words).
               left(39) '\-'
                                  else say program.word(program.words)
        end
      line = in.readline()
   end

   -- wait for exit of child process and check return code
   child.waitFor()
   if child.exitValue() \= 0 then
      say 'UNZIP return code' child.exitValue()

 catch IOException
    say 'Sorry cannot find' unzip
 catch e2=InterruptedException
    e2.printStackTrace()
end
```

Just firing off a program is no big deal, but this example (in script style) shows how easy it is to access the in- and output handles for the environment that executes the program, which enables you to capture the output the non-jvm program produces and do useful things with it.[7] Line 17 starts the external command using the JVM `Runtime` class in a process called `child`. In line 20 we create a `BufferedReader` from the `child` processes' output. This is called an InputStream but it might as well have been called an OutputStream- everything regarding I/O is relative - but fortunately the designers of the JVM took care of deciding this for you. In lines 25-35 we loop through the results and show the files stored in the zipfile. Note that we **do** (line 14) have to **catch** (line 42) the *IOException* that ensues if the runtime cannot find the `unzip` program, maybe because it is not on the path or was not delivered with your operating system.

Starting from JVM 1.5 releases, there is a new way to accomplish the same goal, in a cleaner manner and with the added bonus of being able to redirect streams, and use environment variables. In this regard, the environment variable has made an important comeback from having its calls deprecated, to easy to use support in the *ProcessBuilder* class.

```
/**
 * Class OSProcess implements ways to execute and get output from an
    OS Process
 */
class OSProcess

  properties indirect
  pid = Process
  returncode
  commandList = ArrayList()
  outList = ArrayList()

  properties private
  listeners = HashSet()
  /**
   * Default constructor
   */
  method OSProcess()
    return

    /*
     * helper method that makes an ArrayList of out a Rexx string for
        use
     * in the similarly named method that has an ArrayList as input
     */
```

[7]This is akin to what one would do with *queue* on z/VM CMS and *outtrap* on z/OS TSO in Classic Rexx.

```
method outtrap(command_=Rexx) returns ArrayList
  if command_ = '', command_ = null then return null
  a = ArrayList()
  loop until command_ = ''
    parse command_ first command_
    a.add(first.toString())
  end
  return this.outtrap(a)

  /*
   * helper method that makes an ArrayList of out a Rexx string for
       use
   * in the similarly named method that has an ArrayList as input
   */
method exec(command_=Rexx, wait=1)
  if command_ = '', command_ = null then return
  a = ArrayList()
  loop until command_ = ''
    parse command_ first command_
    a.add(first.toString())
  end
  this.exec(a,wait)

/**
 * Method outtrap starts an OS process from a command line in an
    ArrayList
 * @param command is a List that has the command to be executed as
    elements
 * @return List containing the output of the command
 */
method outtrap(command_=ArrayList) returns ArrayList
  this.commandList = command_
  do
    pb = ProcessBuilder(command_)
    pb.redirectErrorStream(1)
    this.pid = pb.start()
    in = BufferedReader(InputStreamReader(this.pid.getInputStream()
       ))
    line = Rexx in.readLine()
    loop while line <> null
 this.outList.add(line)
 line = Rexx in.readLine()
    end
    pid.waitFor()
    returncode = pid.exitValue()
    return this.outList
  catch iox=IOException
    say iox.getMessage()
    return ArrayList()
  catch InterruptedException
    say "interrupted"
    return ArrayList()
  end -- do

/**
 * Method exec starts an OS process from a command line in an
    ArrayList
 * @param then fires off outputEvent events to every registered
    listener
 * @return void
 */
method exec(command_=ArrayList,wait=1)
  this.commandList = command_
  do
    pb = ProcessBuilder(command_)
    pb.redirectErrorStream(1)
    this.pid = pb.start()
    if wait then do
 in = BufferedReader(InputStreamReader(this.pid.getInputStream()))
 line = in.readLine()
 loop while line <> null
   line = in.readLine()
   i = this.listeners.iterator()
   loop while i.hasNext()
     op = OutputEventListener i.next()
   op.outputReceived(OutputLineEvent(this,line,this.pid))
   end
 end
 pid.waitFor()
 returncode = pid.exitValue()
    end
  catch iox=IOException
    say iox.getMessage()
  catch InterruptedException
    say "interrupted"
  end -- do


/**
 * Method addOutputEventListener supports registering an event
    listener
 * @param listener_ is a OutputEventListener
 */
method addOutputEventListener(listener_=OutputEventListener)
  this.listeners.add(listener_)

/**
 * Method removeOutputEventListener supports de-registering an
    event listener
 * @param listener_ is a OutputEventListener
 */
method removeOutputEventListener(listener_=OutputEventListener)
  this.listeners.remove(listener_)
```

In the above sample, we are using two different ways to obtain the output from a process started by the JVM from our own program. The method *outtrap* waits until the invoked process is finished and returns all output lines in an `ArrayList`. Its name is not entirely coincidental with the similar TSO outtrap function.

Sometimes we cannot wait until the child process is finished, for example when it is a long running process and we need to capture the output on a line-by-line basis to see what is happening - in case of the example, this was done to capture the output as part of a testsuite for a multithreaded file transfer application, which has a server resident process that is not supposed to end, because one of its tasks is to poll a directory for incoming files with a specific pattern in the file names. This is implemented using an Event based pattern (as explained in 13.2 on page 35.

```
import java.util.EventObject
/**
 * Class OutputLineEvent embodies the OutputLineEvent
 */
class OutputLineEvent extends EventObject

  properties indirect
  pid = Process
  line
  /**
   * Default constructor
   */
  method OutputLineEvent(ob=Object,line_, pid_=Process)
    super(ob)
    this.line = line_
    this.pid = pid_
    return

import java.util.EventListener
/**
 * Interface OutputEventListener specifies the one mandatory method
    for this interface
 */
class OutputEventListener interface implements EventListener

  method outputReceived(ob=OutputLineEvent)
```

The call would look something like this:

```
    os = OSProcess()
    os.addOutputEventListener(this)
    os.exec(command)
```

The class must `extend OutputEvenListener`, and implement this method:

```
  method outputReceived(ob=OutputLineEvent)
    this.counter = this.counter+1
    say this.counter ob.getPid() ob.getLine()
```

9

Using NetRexx classes from Java

If you are a Java programmer, using a NetRexx class from Java is just as easy as using a Java class from NetRexx. NetRexx compiles to Java classes that can be used by Java programs. You should import the netrexx.lang package to be able to use the short class name for the Rexx (NetRexx string and numerics) class.

A NetRexx method without a returns keyword can return nothing, which is the void type in Java, or a Rexx string. NetRexx is case independent[8]; Java is case dependent. NetRexx generates the Java code with the case used in the class and method instructions. For example, if you named your class Spider in the NetRexx source file, the resulting Java class file is Spider.class. The public class name in your source program must match the NetRexx source file name. For example, if your source file is `SPIDER.NRX`, and your class is `Spider`, NetRexx generates a warning and changes the class name to `SPIDER` to match the file name. A Java program using the class name Spider would not find the generated class, because its name is `SPIDER.class` - if the compile succeeded, which is not guaranteed in case of casing mismatches. If you have problems, compile your NetRexx program with the **options -keepasjava -format**. You then can look at the generated java file for the correct spelling style and method parameters.

[8]With the default of `options nostrictcase` in effect.

10 Classes

Somewhere in the nineties Object Orientation became one of the mainstream ways to organize computer programs, and support for this was added to programming languages. C became C++ with a preprocessor that generates C[9] that is not entirely unlike the NetREXX translator produces Java. Java in itself is syntax-wise a cleaned up version of C++, but in essence an entirely different language. Its inventor and architect, James Gosling, has stated on various occasions that he was planning a fully different syntax for what finally became Java - but that Sun management more or less forced him to use a C++ derived syntax, because C++ compilers was what SUN did well at the time. With Brendan Eich having to base JavaScript qua naming and syntax on Java, the circle that brought the world terse, curly braces based notations, is complete.

For an audience of REXX programmers, the usual OO presentation goes into the advantages of the paradigm. Today, that is not really necessary, and OO is a given; it slightly deviates from earlier notation as result of trying to put data and procedure into *Objects*, but it is no great deal, and this NetREXX Programmer's Guide does not need a special section on the benefits of the OO paradigm. It is assumed that with a few examples everyone should be able to *get* it; some old programmers might resist but there is really no use in fighting the mainstream. Consequently, this section discusses the way to do this in NetREXX; the way NetREXX does it is for a very large part formed by the way the JVM dictates it, adapted to Rexx notational style and conventions.

Where traditional REXX would say:

```
l=left(ourstring,1)
```

the OO-versions of REXX would say:

```
l=ourstring.left(1)
```

As often the case, the hard part is in the notational ommission that OO has as its characteristic: the instance pointer is no part of the function call and has moved to the left (in what now is called a *method*. The weight has shifted from the operation to the object it is called on.

[9] Cfront

10.1 Classes

Classes represent a blueprint, ’cookie cutter’ approach in creating objects that do useful things. A class is defined in a file by the same name (exceptions here for dependent classes). So a class called Cookie is defined in a file called Cookie.nrx. Its *real*, which means its most specific name, including its package specification, is not given by the file name but by the combination of the class=file + the name given on the `package` statement. This enables one to put classes in different packages without having to change the file names.

10.2 Dependent Classes

Dependent Classes are the NetRexx way to implement Java minor classes. There is no in-line definition possible, and dependent classes need their own class definition, but can be defined in the same source file as the classes they depend on. The notational advantage of ’nested’ class definition, like customary in (for example) Java Swing programs is absent. What is present, is the way dependent classes can seamlessly access properties of their parent classes.

10.3 Properties

The properties statement enables us to define variables that are global to the class definition, and as such can be used by all methods of the class.

A properties statement needs at least one *visibility* or *modifier* keyword. When this is left out, a variable called “properties” is defined, which is not an error, but (most of the times) not what was intended.

Because the properties of a class can be externally visible (depending on *visibility* they need to have a type. When the type is omitted in the definition, they are of type `Rexx`. So-called *indirect properties*, defined with the `properties indirect` modifier, give rise to automated generation of *getter* and *setter* methods for use in Java Beans.

11

Using Packages

Any non-toy, non-trivial program needs to be in a package. Only examples in programming books (present company included) have programs without package statements. The reason for this is that there is a fairly large chance that you will give something a name that is already used by someone else for something else. Things are not their names[10], and the same names are given to wildly dissimilar things. The *package* construct is the JVM's approach to introducing *namespaces* into the total set of programs that programmers make. Different people will probable write some method that is called `listDifferences` sometime. With all my software in a package called `com.frob.nitz` and yours in a package called `com.frob.otzim`, there is no danger of our programs calling the wrong class and listing the wrong differences.

It is imperative to understand this chapter before continuing - it is a mechanical nuts-and-bolts issue but an essential one at that.

11.1 The package statement

The final words about the NetREXX **package** statement is in the NetREXX Language Reference, but the final statement about the package *mechanism* is in the JVM documentation.

11.2 Translator performance consequences

Because the NetREXX translator has to scan all packages that it can see (meaning a recursive scan of the directories below its own level in the directory tree, and on its classpath, it is often advisable (and certainly if . (a dot, representing the current directory) is part of the classpath) to do development in a subdirectory, instead of, for example, the top level home directory. If a large number of packages and classes are visible to the translator, compile times will be negatively impacted.

11.3 Some NetREXX package history

All IBM versions of NetREXX had the translator in a package called

[10]Willard Van Orman Quine, Word and Object, MIT Press, 1960, ISBN 0-262-67001-1

```
COM.ibm.netrexx.process
```

The official, SUN ordained convention for package names was, to prepend the reversed domain name of the vendor to the package name, while uppercasing the top level domain. NetRexx, being one of the first programs to make use of packages, followed this convention, that was quickly dropped by SUN afterwards, probably because someone experienced what trouble it could cause with version management software that adapted to case-*sensitive* and case-*insensitive* file systems. For NetRexx, which had started out keenly observing the rules, this insight came late. With the first RexxLA release of NetRexx in 2011, the package name was changed to `org.netrexx`, while the runtime package name was kept as `netrexx.lang`, also because some major other languages follow this convention.

12

JPMS, The Java Platform Module System

Java9+ introduced the Java Platform Module System (JPMS) per JSR 376. While Java 9 still loads external classes from files and jar-files, all run-time packages now are bundled in modules. NetREXX 3.xx was not capable to load classes from the JPMS.

NetREXX 4 is now supporting the JPMS to find and load its needed run-time packages.[11]

From a NetREXX source code perspective, nothing changes. NetRexx is agnostic about modules. It processes packages and classes whether found in directories, jar-files or - now - modules. All existing source code should run unmodified, with the exception that possibly classes could need to be called explicitly when short-named classes now exhibit ambiguous classes if the short-named class is found in more than one module/package. E.g.

```
/modules/java.base/java/util/spi/ToolProvider.class
/modules/java.compiler/javax/tools/ToolProvider.class
```

NetREXX 4 depends on JSR 203 (NIO.2) and thus requires a minimum JDK level of Java 7, whereas NetREXX 3 runs on Java 6. NetREXX 4 compiles and runs on Java 7/8 (without JPMS) and on Java 9+ (with JPMS).

12.1 CLASSPATH

Most implementations of Java use an environment variable called CLASSPATH to indicate a search path for Java classes. The Java Virtual Machine and the NetREXX translator rely on the CLASSPATH value to find directories, zip files, and jar files which may contain Java classes. The procedure for setting the CLASSPATH environment variable depends on your operating system (and there may be more than one way).

- For Linux and Unix (BASH, Korn, or Bourne shell), use:

```
        CLASSPATH=<newdir>:\$CLASSPATH
        export CLASSPATH
```

[11]You'll find, in the NetREXX source code, updates in RxClasser, where method importclasses() is extended to look for packages and classes in JPMS' jrt:/ file system.
Method *packmodfind()* walks the jrt:/ directory tree at initialisation and registers all found packages. Method modfind() registers classes when imported. Method loadclass() loads the class image from the JPMS.

- Changes for re-boot or opening of a new window should be placed in your /etc/profile, .login, or .profile file, as appropriate.
- For Linux and Unix (C shell), use:

  ```
  setenv CLASSPATH <newdir>:\$CLASSPATH
  ```

  Changes for re-boot or opening of a new window should be placed in your .cshrc file. If you are unsure of how to do this, check the documentation you have for installing the Java toolkit.
- For Windows operating systems, it is best to set the system wide environment, which is accessible using the Control Panel (a search for "environment" offsets the many attempts to relocate the exact dialog in successive Windows Control Panel versions somewhat).

The *Quick Start Guide* has more information about CLASSPATH.

13

Programming Patterns

Much has been made of patterns as aggregations of higher level embodiments of programming solutions. It has been observed[12] that of a number of the C++ oriented patterns in Design Patterns[13], some owe their existence to complications in the C++ language and are not readily reproducible in a Java Patterns or Ruby Patterns book. The same goes for NetRexx- in this chapter we would like to present a number of Java patterns usable in NetRexx, and a number of patterns that are unique to NetRexx.

13.1 Singleton

Sometimes we only want one instance of a class, and we want every user of the class to refer to that same instance. In this case, we need to adapt the class construction mechanism to make sure this happens. There are different ways to implement this, one way is shown below.

```
class TheGatherer

  properties static
  instance     = TheGatherer

  method getInstance() returns TheGatherer static protect
    if TheGatherer.instance <> null then
      do
   return TheGatherer.instance
      end
    else
      do
   TheGatherer.instance = TheGatherer()
   return TheGatherer.instance
      end

  /**
   * private constructor enforces singleton
   */
  method TheGatherer() private signals ClassNotFoundException
```

The way that has been chosen here is to make the constructor private, so no other class can use it. We need an alternative method to make the first and only instance of this class, and this is the `getInstance()` method. This checks if a

[12]This observation from a Java patterns book.
[13]Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional; 1994

static property `instance` is null, in which case the private constructor is run and its return value put in `instance`. Every subsequent call to `getInstance()` sees the value of the static variable instance being not null, and returns that value, which now refers to the single instance. There are several ways to enhance this method, but this is a simple way and it fits the bill. For added security, override the methods for class serialization.

There is a common naming pattern for Singletons, which is the prepend the name of the class with *The*, as in the above example.

13.2 Observable and Events

The observer pattern can also be referred to as *Callback*, and the Java Event class delivers support for it. It is very usable if some result needs to be available for a set of callers, where the set is 0 to many. It works as follows: (see a simple implementation in section 8.4 on page 26) An object, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar Model View Controller (MVC) architectural pattern. In the JVM, this object needs to implement the methods of the Listener interface; this interface specifies the addListener and RemoveListener methods; it keeps a collection in which references to the added listener objects are maintained. The listening is done to subclassed Java Event classes. The event specifies the method to be called when 'firing off' and event. This means that this method is called on every listener.

One of the larger benefits: it decouples the observer from the subject. The subject doesn't need to know anything special about its observers. Instead, the subject simply allows observers to subscribe. When the subject generates an event, it simply passes it to each of its observers. Another benefit is that event consuming classes don't have to wait until a process is finished, and can consume events as they come in. The OSProcess class on page 26) uses an event approach to consume output lines from a subprocess - in the version that puts the output in an ArrayList needs to wait for the subprocess to end, but the event driven version can monitor a long running process and analyze output lines whenever they are received.

13.3 Recursive Parse

This is a pattern somewhat unique to Rexx, by virtue of Rexx having the Parse statement. It also works in NetRexx.

```
/* process a string word by word */

testString = "Foo Bar Baz Frob Frobnitz Frobbotzim"

loop until testString = ""
```

```
-- copy the first word of testString into curName
-- remove it from testString

  parse testString curName testString

  say "Current word is " curName
  say "Remaining words in testString are " testString
  end -- loop until testString
```

This enables one to 'peel off' one word of a sentence at at time.

13.4 More Observer/Observable

Java has special support for the Observer/Observable pattern in the form of the *Observer* class and the *Observable* interface. In the following snippet one can see the Observer class in working.

The class is the same *singleton* as shown above, and starts several threads which need to be observed.

```
class TheGatherer implements Observer
```

The Observable threads need to implement the *Observable* interface, and to be able to be started as a thread, *Runnable*. This is how we start it; its definition follows.

```
  method TheGatherer() private signals ClassNotFoundException
    logger_.info( "TheGatherer: start")
    t1 = TransactionStatusMonitor(Rexx 10000)
    t1.addObserver(this)
    Thread(t1).start()
    logger_.info( "TheGatherer: started thread
       TransactionStatusMonitor")
```

We instantiate the Observable class as `t1`, and add the instance of our Observer class, TheGatherer, to it as Observer. After we have done this, we start it by instantiating a Thread object with it and calling the start method, which, by virtue of it implementing the *Runnable* interface, starts its **run** method.

Note that the TransactionStatusMonitor extends a class called `Monitor`, which in turn, implements the `Observable` interface. The reason for this is, we run several monitoring threads, and they all behave in the same way.

```
import java.util.Observable
class Monitor extends Observable

  properties public
  logger_ = Logger.getLogger(Monitor.class.getName())
  sleeptime

  properties static
  da    = TheDataAccess null

  method Monitor()
    this.da  = TheDataAccess.getInstance()
```

```
class TransactionStatusMonitor implements Runnable extends Monitor

  method TransactionStatusMonitor(s) signals ClassNotFoundException
    this.sleeptime = s

  method run()
    do
      loop forever
   pi      = this.da.idealq1()
   if pi.getID().length() < 5 then
     do
       nop
     end
   else do
     pi.setIdType('ideal')
     ibidp       = this.da.getIBPostIDStatuses(pi)
     setChanged()
     notifyObservers(ibidp.getStatusDelta())
   end

   Thread.currentThread().sleep(this.sleeptime) -- sleep for
      sleeptime seconds
      end
    catch InterruptedException
      parse source s
      say "thread interrupted:"  s
    end
```

In lines 17 an 18 the magic happens: the `setChanged()` method sets the status of this instance as updated, and the `notifyObservers()` method calls for all the registered *Observers* their `update()` methods; this has the following signature:

```
  method update(o=Observable,obj=Object) protect
    cl = o.getClass().getName()
```

The `update()` method receives an `Object`. The `.getClass.getName` call is for illustrative purposes and can be used to decide how to treat the received update object.

14

Incorporating Class Libraries

14.1 A Word About Java Generics

Many classes in Java are expressed as generics. It is important to note that the generic is a *compile time only* java type enforcement mechanism, and therefore does not affect NetRexx.

A generic class has, underlying it, a class that accepts one or more objects as parameters - taking as an example the `ArrayList` class, the Java documentation shows that this has a class signature of `public class ArrayList<E>` with one of the constructors being `ArrayList()` and, for example, a method `add(E e)`. If the `Arraylist` is instantiated in Java as follows:-

```
ArrayList<String> stringList = new ArrayList<String>();
```

then the Java compiler will note that the ArrayList is instantiated with a `<String>` object - and will enforce String usage everywhere else that the `<E>` is used in the class documentation - in this case the type `add(E e)`.
Thus

```
stringList.add(”Item”);
```

will be permitted by the compiler, since a string is being added. In contrast,

```
stringList.add(new Integer(7));
```

will fail since a string is not being added.

Remembering that the `ArrayList` deals directly with objects the following short NetRexx program will correctly use `ArrayList` without worrying about the ”complication” of generics.

```
a1 = ArrayList()  -- An ArrayList just deals with Objects

a1.add("Eric")    -- so we give it some Rexx objects
a1.add("Erica")
num = 0
a1.add(num)

say "There are" a1.size "elements in the list" -- and show they
   are present

/* Now, to retrieve them */

loop item over a1
   say item
```

```
    end
```

If one does not need generics, then it could be asked why they have been implemented at all - the answer is that they prevent many Java run-time errors resulting from a failure to cast the object used to the correct type. When programming in NetREXX the use of the ”universal” `Rexx` class means that this is rarely an issue. When retrieving objects from a generic class used from within Java one must remember to use the correct type, cast or the `binary` option just as would be expected when using a Java object in any other way.

14.2 The Collection Classes

The Java collections framework (JCF) is a set of classes and interfaces that implement commonly reusable collection data structures. The JCF provides both interfaces that define various collections and classes that implement them. Collection implementations in pre-JDK 1.2 versions of the Java platform included few data structure classes, but did not contain a collections framework. The standard methods for grouping Java objects were via the array, the Vector, and the Hashtable classes, which were not easy to extend, and did not implement a standard member interface. The collections framework was designed and developed primarily by Joshua Bloch, and was introduced in JDK 1.2.

Almost all collections in Java are derived from the `java.util.Collection` interface. Collection defines the basic parts of all collections. The interface states the `add()` and `remove()` methods for adding to and removing from a collection respectively. Also required is the `toArray()` method, which converts the collection into a simple array of all the elements in the collection. Finally, the `contains()` method checks if a specified element is in the collection. The Collection interface is a subinterface of `java.util.Iterable`, so any Collection is iterable (using an iterator for a loop over the contents). All collections have an iterator that goes through all of the elements in the collection.

The Collection framework is one of the aspects of where NetREXX relegates to Java for its implementation. Where ooREXX has had its collection classes in the language definition from day one, in NetREXX they are not part of the language; most of the data related support is in the indexed strings feature. This, in turn, makes use of the Dictionary mechanism already implemented in the earliest versions of Java; NetREXX language design was long complete when JDK 1.2 came out.

The Pre-Java Generic classes JFC had, in order to be generic, an interface in which objects could be added in as a java.lang.Object, but on return, that object needed to be typecast to the right type. Using collection classes did entail a good deal of casting return values, as type REXX was not part of the set of types
3.02 that collections had native support for. Modern NetREXX versions have builtin support for using type REXX in collection classes[14], so these can be added to and retrieved from collection classes without further ado.

[14]In actuality, the needed interfaces, like *Comparable* and *Comparator* are now provided in the REXX type

The NetRexx native REXX datatype contains a Java Hashtable which is part of the Collections Framework. New classes, constructors and methods have been added to implement the Java Map interface and allow better interoperation with Java. Some of the new collections support methods include `isindexed()` to check if a Map currently exists, `size()` to determine the count of map entries and `buildmap(sequence1,sequence2)` to construct Rexx maps from arrays or Java Lists. Other classes and methods are documented in the Java Collections Map interface Javadocs. ”isindexed()” returns 0 if no indexed values exist and 1 if there is at least one indexed value in a Rexx object. To build a new indexed Rexx map with the buildmap method you can do this: `Rexx(default).buildMap(keys, values)` where keys and values are any arrays or Java collections framework Lists and default is the default value for the Rexx variable (using the standard Rexx constructors).

All elements are converted to strings before being added to the indexed Rexx variable which is returned. Null can be passed for one of the keys or values parameters to default to a 1-n integer sequence matching the other parameter but if both parameters are provided they must have the same length. Note that arrays do not need to be string arrays and that primitive arrays such as int[] are also accepted.

Collection is a Java generic. Any collection can be written to store any class. For example, Collection<String> can hold strings, and the elements from the collection can be used as strings without any casting required. NetREXX 3.02 added `loop over` support in NetREXX programs for collection classes; this has been implemented without the need for Java generics. This makes it impossible to use the generics mechanism to constrain collection class membership to a specified type. This, however, can be easier accomplished by subclassing the collection class and overriding its constructors.

15

Input and Output

A NetRexx design decision was to leave I/O operations out of the language, and to depend on the JVM functionality for this. This turned out to be a good decision, as JVM I/O has been enhanced and changed over the years; also, the various environments in which NetRexx can be used as a programming language, are not limited to file I/O, but have various implementations to interact with the outside world. A NetRexx program that employs Web technology has different method calls to make than a program that uses ISPF for user interaction.

This does not preclude us to implement file I/O in a way that is reminiscent of Classic Rexx, and in fact this has been done, and the future might see a compatible implementation in NetRexx.

15.1 The File Class

The Java `File` class represents a file; various pieces of information can be requested from a instance of this class, when it points to a file on disk.

15.1.1 Example

```
/* file\FileInfo.nrx

   Display file/directory/path information */

   parse arg fileName .
   if fileName = "" then do
      say "Enter file or directory name to test ?"
      filename = ask
   end
    f1 = File(fileName)                            -- create file object
   if f1.exists() = 0 then do
      say 'File:' filename 'does not exist.'
      exit 8
   end

   say "System related information ----------------------"
   say "  pathSeparator     :" f1.pathSeparator         -- these are
      not methods
   say "  pathSeparatorChar :" f1.pathSeparatorChar     -- they are
      public
   say "  separator         :" f1.separator             --
      static
```

```
say "  separatorChar     :" f1.SeparatorChar        --
   class variables
say
say "File/directory related information --------------"
say "  canRead           :" f1.canRead()
say "  canWrite          :" f1.canWrite()
say "  isDirectory       :" f1.isDirectory()
say "  isFile            :" f1.isFile()
say "  length            :" f1.length()
say "  lastModified      :" f1.lastModified() "=" Date(f1.
   lastModified())
say "  isAbsolute        :" f1.isAbsolute()
say "  getAbsolutePath   :" f1.getAbsolutePath()
say "  getCanonicalPath  :" f1.getCanonicalPath()
say "  getPath           :" f1.getPath()

parent1 = f1.getParent()
if parent1 = null then parent1 = "null returned"
say "  getParent         :" parent1
say "  getName           :" f1.getName()
say "  toString          :" f1.toString()
say "  hashCode          :" f1.hashCode()

if f1.isDirectory() then do
  say
  say "List of this directory ---------------------------\n"
  list1 = f1.list()
  if list1.length = 0
     then say "  directory is empty"
     else
       loop i = 0 to list1.length -1
          f2 = File(f1.getAbsolutePath()''f1.separator''list1[i])
          if f2.isDirectory() then say "  Dir :" list1[i]
                                    else say "  File:" list1[i]
       end
end
say "\n-------------------------------------------------"
-- end fileinfo
```

15.1.2 Line mode I/O using BufferedReader and FileOutputStream

While standard Java I/O does not perform particularly well in the unbuffered version, a `BufferedReader` can be wrapped around any `Reader` in order to maximize the amount of data that is read in one I/O operation.

Example

```
/* linecomment.nrx -- convert appropriate block comments to line
   comments */

/* This is a sample file input and output program, showing how to
   open,
   check, and process text files, and handle exceptions.
   Note the use of the Reader and Writer classes, which convert your
   local computer's 'code page' (character encoding) to Unicode
      during
   reading and back again during writing. */

parse arg fin fout .   -- get the arguments: input and output files
if fout='' then do
  say '# Please specify both input and output files'
  exit 1
end

/* Open and check the files */
do
  infile=File(fin)
  instream=FileInputStream(infile)
  inhandle=BufferedReader(InputStreamReader(instream))
  outfile=File(fout)
  if outfile.getAbsolutePath=infile.getAbsolutePath then do
    say '# Input file cannot be used as the output file'
    exit 1
  end
  outstream=FileOutputStream(outfile)
  outhandle=OutputStreamWriter(outstream)
  say 'Processing' infile'...'
catch e=IOException
  say '# error opening file' e.getMessage
end

linesep=System.getProperty('line.separator') -- be platform-neutral

/* The main processing loop */
loop linenum=1 by 1
  line=Rexx inhandle.readLine           -- get next line [as Rexx
     string]
  if line=null then leave linenum       -- normal end of file

  parse line pre '/*' mid '*/' post     -- process the line
  if pre\='' then
   if mid\='' then
   if post=='' then
    line=pre'--'mid

  if linenum>1 then outhandle.write(linesep, 0, linesep.length)
  outhandle.write(line, 0, line.length)
catch e=IOException
  say '# error reading or writing file' e.getMessage
catch RuntimeException
  say '# processing ended'
finally do                              -- close files
    if inhandle\=null  then inhandle.close
    if outhandle\=null then outhandle.close
  catch IOException
    -- ignore errors during close
  end
end linenum

say linenum-1 'lines written'
```

15.2 Object Oriented I/O using Serialization

The serialization methods of a Class can be used to write a class as serialized binary data. Using the `writeObject()` method, an object can be written to a file using one call. Note that the REXX class is serializable for a long time already.

15.2.1 Example

```
/* file\SeriaIO.nrx

   Output of a Customer object with binary data using Serialization
      */

class SeriaIO
  Properties constant
    yes = boolean 1
    no  = boolean 0

  method main(args=String[]) static
    custDB = Customer2[4]                             -- allocate 4
       customers
    custRD = Customer2[]                              -- read back "x"
       customers

    -- instanciate objects
    custDB[0] = Customer2(101,"Ueli Wahli"         ,"U.S.A."
       ,500.5,25,yes)
    custDB[1] = Customer2(102,"Peter Heuchert"     ,"Germany"
       ,400.4,30,yes)
    custDB[2] = Customer2(103,"Frederik Haesbrouck","Belgium"
       ,350.9,24,no)
    custDB[3] = Customer2(104,"Norio Furukawa"     ,"Japan"
       ,250.5,39,no)

    -- writes the object variables to a file
    say 'Writing' custDB.length 'customers'
    os = ObjectOutputStream(FileOutputStream("seriaio.dat"))
    os.writeInt(custDB.length)                        -- number of objects

    os.writeObject(custDB)                            -- WRITE OBJECTS
       WITH ONE CALL

    os.flush()                                        -- force output
    os.close()

    -- reads the object variables from the file
    say 'Reading...'
    is = ObjectInputStream(FileInputStream("seriaio.dat"))
    n  = is.readInt()                                 -- number of
       customers
    say 'Display of' n 'customers:'

    custRD = Customer2[] is.readObject()              -- READ OBJECTS WITH
        ONE CALL
    loop i = 0 to custRD.length-1
      say custRD[i].getCustNo() (Rexx custRD[i].getName()).left(20) -
          (Rexx custRD[i].getAddress_()).left(10) -
          (Rexx custRD[i].getHourly() * custRD[i].getWork()).right
             (10) -
          custRD[i].getBool()
    end
    is.close()

/* ----------------------------------------------- */
/* Customer class                                  */
/* ----------------------------------------------- */
class Customer2 implements Serializable

  properties private                             -- various data
     types
    custNo   = String
    name     = String
    address_ = String
    hourly   = float
    work     = int
    bool     = boolean

  method Customer2(aCustNo=String, aName=String, aAddress_=rexx, -
                   aHourly=float, aWork=int, aBool=boolean)
    custNo = aCustNo; name = aName; address_ = aAddress_
    hourly = aHourly; work = aWork; bool = aBool

  method getCustNo() returns String
    return custNo
  method getName() returns String
    return name
  method getAddress_() returns Rexx
    return address_
  method getHourly() returns float
    return hourly
  method getWork() returns int
    return work
  method getBool() returns boolean
    return bool
-- end
```

15.3 Using the SAY instruction to write lines to a file

It used to be that a lot of programs started out with using `say` statements to write to the console, and later, when output became too voluminous, needed to be reworked to use output statements. `Say` has been extended to write to any (and multiple) outputstream(s).

The `setOutputStream()` Method takes an *OutputStream* which from that moment on is used for output. This can be `System.out` (which is the default) but also `System.err`, to direct error messages to. This outputstream can also be directed to a file, using a `FileOutputStream`.

In addition to the `setOutputStream()` operation, which replaces the previously set *OutputStream*, there are also `pushOutputStream()` and `popOutputStream()`, which add (push) and remove (pop) streams from a list. In this way, it is possible to direct output to, e.g. an System.out and System.err stream, and at the same time to a number of files.

These operations are not a good fit for multithreaded programs. For use in the heavily multithreaded Pipelines environment, the method `RexxIO.pipeSay` was designed, which is used in the Pipelines source code, but can also be employed in your own multithreaded programs.

Example

```
/*
 * Illustrates how the say statement became a bit more flexible
 * now able to direct to different output streams, or files
 */

say 'this is stdout'
RexxIO.pushOutputStream(System.err)
say 'stdout and stderr'
RexxIO.popOutputStream()
say 'only stdout'
RexxIO.popOutputStream()
say 'only stdout'
RexxIO.popOutputStream()
RexxIO.popOutputStream()
RexxIO.popOutputStream()

RexxIO.setOutputStream(FileOutputStream('testfile1.txt'))
say 'this goes to testfile1.txt'
RexxIO.pushOutputStream(FileOutputStream('testfile2.txt'))
say 'this goes to testfile2.txt'
```

15.4 Using RexxIO.forEachLine

A pattern in which there is an action for every line in a file, is now supported with the oneliner file handler, RexxIO.forEachLine. A Class that implements the interface `LineHandler` can read lines from a file using the `RexxIO().File('filename').forEachLine(x)` idiom, which is very compact. The `LineHandler` interface needs to implement the `handle()` method, which is fed the line that has been read.

Example

```
class testLine implements LineHandler
  method main(args=String[]) static

    RexxIO().File("legenda.txt").forEachline(testLine())
    RexxIO().File("legenda.txt").forEachline(testLine().testFile2())

  method handle(in)
    say in

  class testLine.testFile2 dependent implements LineHandler
```

```
method handle(in)
  say in
```

16 Algorithms in NetRexx

16.1 Factorial

A *factorial* is the product of an integer and all the integers below it; the mathematical symbol used is ! (the exclamation mark). For example 4! is equal to 24 (because 4*3*2*1=24). The following program illustrates a recursive (a method calling itself) and an iterative approach to calculating factorials.

```
/* NetRexx */

options replace format comments java symbols nobinary

numeric digits 64 -- switch to exponential format when numbers become
    larger than 64 digits

say 'Input a number: \-'
say
do
  n_ = long ask -- Gets the number, must be an integer

  say n_'! =' factorial(n_) '(using iteration)'
  say n_'! =' factorial(n_, 'r') '(using recursion)'

  catch ex = Exception
    ex.printStackTrace
end

return

method factorial(n_ = long, fmethod = 'I') public static returns Rexx
    signals IllegalArgumentException

  if n_ < 0 then -
    signal IllegalArgumentException('Sorry, but' n_ 'is not a
       positive integer')

  select
    when fmethod.upper = 'R' then -
      fact = factorialRecursive(n_)
    otherwise -
      fact = factorialIterative(n_)
    end

  return fact

method factorialIterative(n_ = long) private static returns Rexx
```

```
  fact = 1
  loop i_ = 1 to n_
    fact = fact * i_
    end i_

  return fact

method factorialRecursive(n_ = long) private static returns Rexx

  if n_ > 1 then -
    fact = n_ * factorialRecursive(n_ - 1)
  else -
   fact = 1

  return fact
```

Executing this program yields the following result:

```
===== Exec: RCFactorial =====
Input a number:
42
42! = 1405006117752879898543142606244511569936384000000000 (using iteration)
42! = 1405006117752879898543142606244511569936384000000000 (using recursion)
```

As you can see, fortunately, both approaches come to the same conclusion about the results. In the above program, both approaches are a bit intermingled; for more clarity about how to use recursion, have a look at this:

```
class Factorial
numeric digits 64

  method main(args=String[]) static
    say factorial_(42)

  method factorial_(number) static
    if number = 0 then return 1
    else return number * factorial_(number-1)
```

In this program we can clearly see that the `factorial_` method, that takes an argument `number` (which is of type Rexx if we do not specify it to be another type), calls itself in the method body. This means that at runtime, another copy of it is run, with as argument number that the first invocation returns (the result of 42*41), and so on.

In general, a recursive algorithm is considered more elegant, while an iterative approach has a better runtime performance. Some language environments are optimized for recursion, which means that their processors can spot a recursive algorithm and optimize it by not making many useless copies of the code. Some day in the near future the JVM will be such an environment. Also, for some problems, for example the processing of tree structures, using a recursive algorithm seems much more natural, while an iterative algorithm seems complicated or forced.

16.2 Fibonacci

```
/* NetRexx */
options replace format comments java symbols

numeric digits 210000                   /*prepare for some big ones.
       */
parse arg x y .                         /*allow a single number or
   range.*/
if x == '' then do                      /*no input? Then assume
   -30-->+30*/
 x = -30
 y = -x
 end

if y == '' then y = x              /*if only one number, show fib(n)*/
loop k = x to y                    /*process each Fibonacci request.*/
 q = fib(k)
 w = q.length                     /*if wider than 25 bytes, tell it*/
 say 'Fibonacci' k"="q
 if w > 25 then say 'Fibonacci' k "has a length of" w
 end k
exit

/*--------------------------------------FIB subroutine (non-recursive)
   ---*/
method fib(arg) private static
 parse arg n
 na = n.abs

 if na < 2 then return na              /*handle special cases.
             */
 a = 0
 b = 1

 loop j = 2 to na
   s = a + b
   a = b
   b = s
   end j

 if n > 0 | na // 2 == 1 then return  s /*if positive or odd negative
    ... */
                         else return -s /*return a negative Fib
                            number.  */
```

17

Using Parse

The `Parse` statement is one of the stalwarts of the Rexx family of languages, and allows one to easily split a string into parts without needing to resort to more traditional techniques of string processing.

The syntax of a parse statement is

```
parse term template
```

where `term` is a string or a previously initialised variable. The template is a list of instructions describing how to split the string.

17.1 Literal Parsing

The most common use of `parse` is to split a string up into parts separated with a delimiter - whilst the most common delimiter is a simple space any string may be used:-

```
log = "2014/05/15 21:35:47.012 - error in {{[findit]}}"
parse log year "/" month "/" day hour ":" minute ":" second "."
   msecond "-" text
say "On day" day "of month" month "at about" hour":"minute "we got"
   text
parse text "{{[" name "]}}"
say name
```

Here `log` is composed of a datestamp separated from a message by a hyphen. The datestamp is composed of a date separated from a time by a space - within the date the year month and day are delimited by a slash and within the date the hour, minute and second fields by a colon. The millisecond field is separated from the seconds by a decimal point.

The first `parse` divides these using the relevant delimiter - where there is no delimiter then a space is used.

The `term` is the variable `log` and the `template` is

```
year "/" month "/" day hour ":" minute ":" second "." msecond "-" text
```

This first template may be read as the following sequence of actions

1. Assign the contents of `log` to the variable `year` until a `/` is encountered (`2014`)
2. Following the `/` assign `month` with the sting found up until another `/` (`05`)

3. Place the contents following the / until a space into the variable `day` (15)
4. Following the space, assign the value found up until the : into the hour variable (21)
5. Repeat for the variable `minute` (35)
6. Assign the `second` value up until the .
7. Take the value for `msecond` until a delimiter of - is seen
8. Assign the remainder to variable `text`

The second `parse` statement shows how the delimiters can be more complex - the `template` is

```
"{{[" name "]}}"
```

and extracts the value between {{[and]}} to the variable (name)
Running the above example will produce the following output:-

```
At about 21:35 we got  error in {{[findit]}}
findit
```

As another example, consider

```
quote = "Now is the winter of our discontent"
loop forever
   parse quote word quote
   say word
   if quote = "" then leave
end
```

This will take the first word from `quote`, and assign the remainder back into `quote`, print the word taken and repeat until the variable `quote` is the empty string. The output from this will be

```
Now
is
the
winter
of
our
discontent
```

17.1.1 The Placeholder (dummy) Variable

The first example assigns values to several variables that are not used - this is unnecessary and can be avoided by the use of a placeholder variable which is the . character.
If this is done, the first parse statement becomes

```
parse log . "/" month "/" day hour ":" minute ":" . "." . "-" text
```

The output will remain the same.

17.2 Positional Parsing

Whilst the majority of parsing can be done using a fixed literal delimiter, the `parse` instruction also allows parsing based on positional patterns. This is achieved with the use of numerical values in the template - the values may also take a prefix of `+`, `-` or `=`

`no prefix` or `=` indicates that the number is an **absolute** column value in the string being parsed

`+` indicates a **relative** position that starts from the specified position *after* the position where the last match occurred

`-` indicates a **relative** position that starts from the specified position *before* the last match

These points are best illustrated by example

```
quote = "Now is the winter of our discontent"
tens  = "         1111111111222222222233333"
units = "12345678901234567890123456789012345"

say quote
say tens
say units

parse quote 10 str1 20 -8 str2 +6 str3
-- str1 starts at column 10 and is 10 chars long
say  str1 "("str1.length")"
-- str2 steps back 8 chars and is 6 chars long
say str2 "("str2.length")"
-- str3 is the remainder of the string (as should be expected)
say str3
```

Running this gives the following

```
Now is the winter of our discontent
         1111111111222222222233333
12345678901234567890123456789012345
e winter o (10)
winter (6)
 of our discontent
```

Both `literal` and `positional` parsing can be combined. Keen-eyed readers will have noted that the output from the first example contained an extra space before the word `error`

```
At about 21:35 we got  error in {{[findit]}}
Extra space here     ^^
```

This is the result of assigning the *remainder* of the string to the variable text - leading blanks are normally removed *except* in this special case.

One can use a positional pattern to eliminate this extra space:-

```
log = "2014/05/15 21:35:47.012 - error in {{[findit]}}"
parse log . "/" month "/" day hour ":" minute ":" . "." . "-" +2 text
```

```
say "On day" day "of month" month "at about" hour":"minute "we got"
    text
parse text "{{[" name "]}}"
say name
```

Note that the relative positional pattern used here is `+2` - `0` is the position of the last match which is the hyphen, `+1` is the position of the following space and thus `+2` is the start of the target string.

17.3 Variable Templates

Variables may be used as the pattern in the `templates` in order to accommodate the occasions when the pattern may need to be specified at runtime. An illustration of this is the following evolution of the first example that will correctly parse dates specified in two distinct ways

```
log = ""
log[1] = "2014/05/15 21:35:47.012 - error in {{[findit]}}"
log[2] = "2014-05-15 21:35:47.012 - error in {{[findit]}}"

loop i = 1 to 2
   dtsep = log[i].substr(5,1)
   parse log[i] . (dtsep) month (dtsep) day hour ":" minute ":" . "."
        . "-" +2 text
   say "On day" day "of month" month "at about" hour":"minute "we got
       " text
end
```

Note that he date separator `dtsep` is determined and then used in the parse pattern by enclosing it in parentheses, thus `(dtsep)`. The output of this program is

```
On day 15 of month 05 at about 21:35 we got error in {{[findit]}}
On day 15 of month 05 at about 21:35 we got error in {{[findit]}}
```

It can be seen that the date was successfully parsed in both cases.

It is important to note that any pattern specified by a variable *will be assumed to be literal unless it has a +, - or = prefix*. Should one wish to use positional patterns then the prefix **must** be used.

```
message = "this is a message that contains the number 10- just there,
     see?"
pat = "10"
parse message part1 5 (pat) part2
say "literal:" part1 part2
parse message part1 5 =(pat) part2
say "positional:" part1 part2
```

When run this illustrates the difference between the two parse statements

```
literal: this - just there, see?
positional: this  message that contains the number 10- just there, see?
```

18 Using Trace

The `trace` command is the inbuilt debugging facility of the REXX family, and, as might be expected from its name, allows one to trace the execution of your program. It is possible to trace both program statements and the state of variables within your code.

`(Trace)` is a compile-time option, and should be disabled once debugging as been completed.

The syntax of the trace command is

```
trace traceitem
```

where `traceitem` defines the behaviour of the trace command. Only one `traceitem` may be given, and only one of the program statement tracing options will be in use at any time. Variable tracing options, however, are *additive* and such statements may appear multiple times.

All `trace` output is headed by three hyphens followed by the source file name, as follows

```
--- TerribleExample.nrx
```

18.1 Tracing Program Statements

The `traceoptions` that affect the tracing of program statements are

all will display all statements as they are executed. Each line in the trace output will be prefixed with `*=*` or a `*-*` should output span subsequent lines.
The `trace all` statement can be placed anywhere in the program source.

methods will show the each method as it is invoked, along with any parameters to it. The trace output for method traces is prefixed by a `*=*` for the method call itself and a `>a>` indicating the assignment of a value to a method parameter. *No other program statements will be traced.*
The `trace methods` statement should be placed *before* the first method is defined in a class.

results acts as though the `trace all` statement had been given, and, if placed *before* any method will also act as though `trace methods` was also specified.
In addition to the `all` and `methods` tracing implied by `results` the following will also take place

Properties will have their assignments shown. These will be identified by `>p>`

Local variables will also be traced, with assignments identified by `>v>`

Expressions will have their evaluations shown if not shown for as a part of `properties` or `local variable` trace output. Such evaluations are indicated by »>

off `trace off` disables tracing. No further tracing output will take place.

18.2 Tracing Variables

The all-or-nothing tracing offered by, for instance `trace results` can lead to a deluge of trace information in many cases.

In these instances one may more finely control which variables one wishes to monitor using the `trace var` statement. The syntax of the `trace var` statement is

```
trace var var1 [var2...]
```

or

```
trace var -var1 [-var2...]
```

where the first form adds variables to the list that should be watched, and the second removes them. The forms may be mixed to add some variables and remove others simultaneously, as here:-

```
trace var var1 -var2 var3 -var4 -var5
```

to monitor `var1` and `var3` and remove `var2, var5` and `var5` from the list of watched variables.

Multiple `trace var` statements may be used, as mentioned above.

It is not an error to specify a variable name that does not exist.

Each variable can appear only *once* in a `trace` statement.

A variable name may that of any type - including arrays (without the `[]`).

Program tracing options never alter the list of watched variables. If tracing has previously been turned off then variable tracing may be resumed simply with a `trace var` statement.

18.3 Examples

18.3.1 Program Trace

Trace All

Running the program below

```
trace all

class traceExample

   properties
      aIs
      bIs

   method traceExample(a, b)
      aIs = a
      bIs = b

   method times
      retturn aIs * bIs

   method main($cmdin1=String[]) static
      arg=Rexx($cmdin1)
      te = traceExample(2, 3)
      fred = te.times
      say fred
```

gives trace output of

```
     --- traceExample.nrx
  16 *=*  method main($cmdin1=String[]) static
     >a> $cmdin1 "[Ljava.lang.String;@72ebbf5c"
  17 *=*   arg=Rexx($cmdin1)
  18 *=*   te = traceExample(2, 3)
   9 *=*  method traceExample(a, b)
     >a> a "2"
     >a> b "3"
  10 *=*   aIs = a
  11 *=*   bIs = b
  12 *-*
  19 *=*   fred = te.times
  13 *=*  method times
  14 *=*   return aIs * bIs
  20 *=*   say fred
```

This output may be read thus

— traceExample.nrx Identification of the program being traced. This is the `tracing context`.

16 *=* method main($cmdin1=String[ ) static] The first line that is actually executed is line 16.

>a> $cmdin1 "[Ljava.lang.String;@72ebbf5c" Variable `$cmdin1` is assigned a string value from the java virtual machine.

17 *=* arg=Rexx($cmdin1) Line 17 is executed next...

18 *=* te = traceExample(2, 3) followed by line 18

9 *=* method traceExample(a, b) Line 18 is a method call to a method on line 9...

>a> a "2" which assigns a value of 2 to parameter a

>a> b "3" and a value of 3 to parameter b
10 *=* aIs = a the following lines document only code execution
11 *=* bIs = b
12 *-*
19 *=* fred = te.times
13 *=* method times
14 *=* return aIs * bIs
20 *=* say fred

Trace Methods

Replacing the `trace all` from line 1 with `trace methods` gives trace output of

```
    --- traceExample.nrx
 16 *=*  method main($cmdin1=String[]) static
    >a> $cmdin1 "[Ljava.lang.String;@8094cc7"
  9 *=*  method traceExample(a, b)
    >a> a "2"
    >a> b "3"
 13 *=*  method times
```

As should be expected, this is a subset of the output provided when using `trace all`.

Trace Results

Replacing the `trace all` from line 1 with `trace results` would give

```
    --- traceExample.nrx
 16 *=*  method main($cmdin1=String[]) static
    >a> $cmdin1 "[Ljava.lang.String;@72ebbf5c"
 17 *=*   arg=Rexx($cmdin1)
    >>> "[Ljava.lang.String;@72ebbf5c"
    >v> arg ""
 18 *=*   te = traceExample(2, 3)
    >>> "2"
    >>> "3"
  9 *=*  method traceExample(a, b)
    >a> a "2"
    >a> b "3"
 10 *=*   aIs = a
    >p> aIs "2"
 11 *=*   bIs = b
 12 *-*
 11 >p> bIs "3"
 18 >v> te "traceExample@53606bf5"
 19 *=*   fred = te.times
 13 *=*  method times
 14 *=*   return aIs * bIs
    >>> "6"
 19 >v> fred "6"
 20 *=*   say fred
    >>> "6"
```

Here is can be seen that more information is available. Noticeably, the values of assignments are given. For instance

Line 17 now has an entry of **>v> arg ""** showing that hte value of the variable `arg` was set to the empty string
Line 18 now has the values of the specified parameters evaluated (**»> "2"** and **»» "3"**)
Lines 10 and 11 show that values were assigned to parameters (**>p> aIs "2"** and **>p> bIs "3"**)
Line 18 then shows the assignment of the instantiated class to variable `te`
Line 14 shows the evaluation of the multiplication (**»> "6"**), which is assigned to variable `fred` in line 19 (**>v> fred "6"**) on line 19.
Finally we see the evaluation of variable `fred` on line 20.

18.3.2 Variable Tracing

Consider the following example:-

```
a = "a"
b = "b"
c = 1
d = 2
e = 3

trace var a b c d e f y
z = a || b
y = c + d
f = y + 2
e = f

trace var -a -c -d -e
y = y * 2
a = y
e = a
```

Running this will produce the output below

```
    --- variableTraceExample.nrx
  9 *=* y = c + d
    >v> y "3"
 10 *=* f = y + 2
    >v> f "5"
 11 *=* e = f
    >v> e "5"
 14 *=* y = y * 2
```

```
>v> y "6"
```

It can be seen that only the lines that contain watched variables are traced. This the variable assignments on lines 9, 10 and 11 are displayed, since the variables being watched from line 7 to line 12 are `a, b, c, d, e, f` and `y`.

Following this, however only the assignment to variable `y` is shown, since the variables `a, b ,c d` and `e` are removed from the list with the command `trace var -a -c -d -e`.

18.4 Tracing Notes

One further prefix may be encountered in the trace outout `+++` which signifies an error.

Whenever tracing transfers to a different source file, a new `tracing context`, identified by the – prefix is output.

Tracing is expensive, and may dramatically impact the run-time performance of the program being traced. Judicious use may therefore be warranted.

19

Concurrency

19.1 Threads

Threads are a built-in multitasking feature of the JVM. Where earlier JVM implementations sometime ran on so-called *Green Threads*, which is a library that implements thread support for OS'ses that do not have this facility (an early version of Java was called *GreenTalk* for this reason), modern versions all use native OS thread support.

A new thread is created when we create an instance of the Thread class. We cannot tell a thread which method to run, because threads are not references to methods. Instead we use the Runnable interface to create an object that contains the run method:

Every thread begins its concurrent life by executing the run method. The run method does not have any parameters, does not return a value, and is not allowed to signal any exceptions. Any class that implements the Runnable interface can serve as a target of a new thread. An object of a class that implements the Runnable interface is used as a parameter for the thread constructor.

Threads can be given a name that is visible when listing the threads in your system. It is good practice to name every thread, because if something goes wrong you can see which threads are still running. Additionally, threads are grouped by thread groups. If you do not supply a thread group, the new thread is added to the thread group of the currently executing thread. The threads of a group and their subgroups can be destroyed, stopped, resumed, or suspended by using the ThreadGroup object.

The next two samples are used in the following programs that illustrate thread usage.

```
/* thread/ThrdTst1.nrx */

h1 = Hello1('This is thread 1')
h2 = Hello1('This is thread 2')

Thread(h1,'Thread Test Thread 1').start()
Thread(h2,'Thread Test Thread 2').start()

class Hello1 implements Runnable
  Properties inheritable
    message = String

  method Hello1( s = String)
    message = s
  method run()
    loop for 50
      say message
    end

/* thread/ThrdTst2.nrx */

h1 = Hello2('This is thread 1')
h2 = Hello2('This is thread 2')

h1.start()
h2.start()

class Hello2 extends Thread
  Properties inheritable
    message = String

  method Hello2( s = String)
    super('Thread Test - Message' s)
    message = s

  method run()
    loop for 50
      say message
      do
        sleep(10)
        catch InterruptedException
      end
    end
```

The second class, Hello2, does not *implement* the `Runnable` interface, but subclasses it, so it inherits its methods. This is a valid approach, and it is up to the developer to choose an implementation and worry about the semantics of an inherited thread interface. A newly created thread remains idle until the start method is invoked. The thread then wakes up and executes the run method of its target object. The start method can be called only once. The thread continues running until the run method completes or the stop method of the thread is called.

20

Using NetRexx for Web applets

Web applets can be written one of two styles:

- Lean and mean, where binary arithmetic is used, and only core Java classes (such as *java.lang.String*) are used. This is recommended for World Wide Web pages, which may be accessed by people using a slow internet connection. Several examples using this style are included in the NetRexx package (eg., *NervousTexxt.nrx* or *ArchText.nrx*).
- Full-function, where decimal arithmetic is used, and advantage is taken of the full power of the NetRexx runtime (Rexx) class. This is appropriate for intranets, where most users will have fast connections to servers. An example using this style is included in the NetRexx package (*WordClock.nrx*).

If you write applets which use the NetRexx runtime (or any other Java classes that might not be on the client browser), the rest of this section may help in setting up your Web server.

A good way of setting up an HTTP (Web) server for this is to keep all your applets in one subdirectory. You can then make the NetRexx runtime classes (that is, the classes in the package known to the Java Virtual Machine as *netrexx.lang*) available to all the applets by unzipping NetRexxR.jar into a subdirectory *netrexx/lang* below your applets directory.
For example, if the root of your server data tree is

```
D:\mydata
```

you might put your applets into

```
D:\mydata\applets
```

and then the NetRexx classes (unzipped from NetRexxR.jar) should be in the directory

```
D:\mydata\applets\netrexx\lang
```

The same principle is applied if you have any other non-core Java packages that you want to make available to your applets: the classes in a package called *iris.sort.quicksorts* would go in a subdirectory below *applets* called *iris/sort/quicksorts*, for example.

Note that since Java 1.1 or later it is possible to use the classes direct from the NetRexxR.jar file. Please see the Java documentation for details.

21

Database Connectivity with JDBC

For interfacing with Relational Database Management Systems (RDBMS) NetRexx uses the Java Data Base Connectivity (JDBC) model. This means that all important database systems, for which a JDBC driver has been made available, can be used from your NetRexx program. This is a large bonus when we compare this to the other open source scripting languages, that have been made go by with specific, nonstandard solutions and special drivers. In contrast, NetRexx programs can be made compatible with most database systems that use standard SQL, and, with some planning and care, can switch database implementations at will.

```
/* jdbc\JdbcQry.nrx

   This NetRexx program demonstrate DB2 query using the JDBC API.
   Usage: Java JdbcQry [<DB-URL>] [<userprefix>] */

import java.sql.

parse arg url prefix                       -- process arguments
if url = '' then
  url = 'jdbc:db2:sample'
else do                                    -- check for correct URL
  parse url p1 ':' p2 ':' rest
  if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
    say 'Usage: java JdbcQry [<DB-URL>] [<userprefix>]'
    exit 8
  end
end
if prefix = '' then prefix = 'userid'

do                                         -- loading DB2 support
  say 'Loading DB2 driver classes...'
  Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
  -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
catch e1 = Exception
  say 'The DB2 driver classes could not be found and loaded !'
  say 'Exception (' e1 ') caught : \n' e1.getMessage()
  exit 1
end                                        -- end : loading DB2 support

do                                         -- connecting to DB2 host
  say 'Connecting to:' url
  jdbcCon = Connection DriverManager.getConnection(url, 'userid', '
      password')
catch e2 = SQLException
  say 'SQLException(s) caught while connecting !'
  loop while (e2 \= null)
    say 'SQLState:' e2.getSQLState()
    say 'Message: ' e2.getMessage()
    say 'Vendor:  ' e2.getErrorCode()
    say
    e2 = e2.getNextException()
  end
  exit 1
end                                       -- end : connecting to DB2
   host

do                                        -- get list of departments
   with the managers
  say 'Creating query...'
  query = 'SELECT deptno, deptname, lastname, firstnme' -
          'FROM' prefix'.DEPARTMENT dep,' prefix'.EMPLOYEE emp'-
          'WHERE dep.mgrno=emp.empno ORDER BY dep.deptno'
  stmt = Statement jdbcCon.createStatement()
  say 'Executing query:'
  loop i=0 to (query.length()-1)%75
    say '    ' query.substr(i*75+1,75)
  end
  rs = ResultSet stmt.executeQuery(query)
  say 'Results:'
  loop row=0 while rs.next()
    say rs.getString('deptno') rs.getString('deptname') -
         'is directed by' rs.getString('lastname') rs.getString('
            firstnme')
  end
  rs.close()                              -- close the ResultSet
  stmt.close()                            -- close the Statement
  jdbcCon.close()                         -- close the Connection
  say 'Retrieved' row 'departments.'
catch e3 = SQLException
  say 'SQLException(s) caught !'
  loop while (e3 \= null)
    say 'SQLState:' e3.getSQLState()
    say 'Message: ' e3.getMessage()
    say 'Vendor:  ' e3.getErrorCode()
    say
    e3 = e3.getNextException()
  end
end                                       -- end: get list of
   departments
```

The first peculiarity of JDBC is the way the driver class is loaded. When most classes are 'pulled in' by the translator, a JDBC driver traditionally is loaded through the reflection API. This happens in line 22 with the `Class.forName` call. This implies that the library containing this class must be on the classpath.

In previous versions of JDBC, to obtain a connection, one first had to initialize the JDBC driver by calling the method Class.forName. Any JDBC 4.0 drivers that are found on the class path are automatically loaded. (However, one must manually load any drivers prior to JDBC 4.0 with the method Class.forName.)

In line 32 of the example we connect to the database using a url and a userid/-password combination. This is an easy way to do and test, but for most serious applications we do not want plaintext userids and passwords in the sourcecode, so most of the time we would store the connection info in a file that we store in encrypted form, or we use facilities of J2EE containers that can provide data sources that take care of this, while at the same time decoupling your application source from the infrastructure that it will run on.

In line 47 the query is composed by filling in variables in a Rexx string and making a `Statement` out of it, in line 50. In line 55, the `Statement` is executed, which yields a `ResultSet`. This has a *cursor* that moves forward with each `next` call. The `next` call returns *true* as longs as there are rows from the resultset to return.

The `ResultSet` interface implements *getter* methods for all JDBC Types. In the above example, all returned results are of type `String`.

```
/* jdbc\JdbcUpd.nrx

   This NetRexx program demonstrate DB2 update using the JDBC API.
   Usage: Java JdbcUpd [<DB-URL>] [<userprefix>] [U] */

import java.sql.

parse arg url prefix lowup                -- process arguments
if url = '' then
   url = 'jdbc:db2:sample'
else do                                   -- check for correct URL
     parse url p1 ':' p2 ':' rest
     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
        say 'Usage: java JdbcUpd [<DB-URL>] [<userprefix>] [U]'
        exit 8
     end
end
if prefix = '' then prefix = 'userid'
if lowup \= 'U' then lowup = 'L'

do                                        -- loading DB2 support
  say 'Loading DB2 driver classes...'
  Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
  -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
catch e1 = Exception
  say 'The DB2 driver classes could not be found and loaded !'
  say 'Exception (' e1 ') caught : \n' e1.getMessage()
  exit 1
end                                       -- end : loading DB2 support

do                                        -- connecting to DB2 host
  say 'Connecting to:' url
  jdbcCon = Connection DriverManager.getConnection(url, 'userid', '
      password')
catch e2 = SQLException
  say 'SQLException(s) caught while connecting !'
  loop while (e2 \= null)
     say 'SQLState:' e2.getSQLState()
     say 'Message: ' e2.getMessage()
```

```
      say 'Vendor:  ' e2.getErrorCode()
      say
      e2 = e2.getNextException()
    end
    exit 1
end                                    -- end : connecting to DB2
   host

do                                     -- retrieve employee, update
   firstname

  say 'Preparing update...'                          -- prepare
     UPDATE
  updateQ = 'UPDATE' prefix'.EMPLOYEE SET firstnme = ? WHERE empno =
     ?'
  updateStmt = PreparedStatement jdbcCon.prepareStatement(updateQ)
  say 'Creating query...'                            -- create
     SELECT
  query = 'SELECT firstnme, lastname, empno FROM' prefix'.EMPLOYEE'
  stmt = Statement jdbcCon.createStatement()
  rs = ResultSet stmt.executeQuery(query)            -- execute
     select

  loop row=0 while rs.next()                         -- loop
     employees
    firstname = String rs.getString('firstnme')
    if lowup = 'U' then firstname = firstname.toUpperCase()
    else do
       dChar = firstname.charAt(0)
       firstname = dChar || firstname.substring(1).toLowerCase()
    end
    updateStmt.setString(1, firstname)               -- parms for
       update
    updateStmt.setString(2, rs.getString('empno'))
    say 'Updating' rs.getString('lastname') firstname ': \0'
    say updateStmt.executeUpdate() 'row(s) updated'   -- execute
       update
  end

  rs.close()                           -- close the ResultSet
  stmt.close()                         -- close the Statement
  updateStmt.close()                   -- close the PreparedStatement
  jdbcCon.close()                      -- close the Connection
  say 'Updated' row 'employees.'
catch e3 = SQLException
  say 'SQLException(s) caught !'
  loop while (e3 \= null)
     say 'SQLState:' e3.getSQLState()
     say 'Message: ' e3.getMessage()
     say 'Vendor:  ' e3.getErrorCode()
     say
     e3 = e3.getNextException()
   end
end                                    -- end: empoyees
```

For database updates, we connect using the driver in the same way (line 23) and now prepare the statement used for the database update (line 50). In this example, we loop through the cursor of a select statement and update the row in line 66. The `executeUpdate` method of `PreparedStatement` returns the number of updated rows as an indication of success.

From JDBC 2.0 on, cursors are updateable (and scrollable, so they can move back and forth), so we would not have to go through this effort - but it is a valid example of an update statement.

22

WebSphere MQ

WebSphere MQ (also and maybe better known as MQ Series) is IBM's messaging and queing middleware, and is in use at a great many financial institutions and other companies. It has, from a programming point of view, two API's: JMS (Java Messaging Services), a generic messaging API for the Java world, and MQI, which is older and proprietary to IBM's product. The below examples show the MQI; other examples might show JMS applications.

This is the sample Java application for MQI, translated (and a lot shorter) to NetRexx.

```
import com.ibm.mq.MQException
import com.ibm.mq.MQGetMessageOptions
import com.ibm.mq.MQMessage
import com.ibm.mq.MQPutMessageOptions
import com.ibm.mq.MQQueue
import com.ibm.mq.MQQueueManager
import com.ibm.mq.constants.MQConstants

class MQSample
properties private

  qManager = "rjtestqm";
  qName = "SYSTEM.DEFAULT.LOCAL.QUEUE"

  method main(args=String[]) static binary
    m = MQSample()
    do
      say "Connecting to queue manager: " m.qManager
      qMgr = MQQueueManager(m.qManager)

      openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF | MQConstants.
          MQOO_OUTPUT

      say "Accessing queue: " m.qName
      queue = qMgr.accessQueue(m.qName, openOptions)

      msg = MQMessage()
      msg.writeUTF("Hello, World!")

      pmo = MQPutMessageOptions()

      say "Sending a message..."
      queue.put(msg, pmo)

      rcvMessage = MQMessage()

      gmo = MQGetMessageOptions()
      say "...and getting the message back again"
      queue.get(rcvMessage, gmo)

      msgText = rcvMessage.readUTF()
      say "The message is: " msgText

      say "Closing the queue"
      queue.close()

      say "Disconnecting from the Queue Manager"
      qMgr.disconnect()
      say "Done!"
   catch ex=MQException
     say "A WebSphere MQ Error occured : Completion Code " ex.
        completionCode "Reason Code " ex.reasonCode
   catch ex2=java.io.IOException
     say "An IOException occured whilst writing to the message buffer
        : " ex2
   end
```

This sample connects to the Queue Manager (called *rjtestqm*) in *bindings mode*, as opposed to *client mode*. Bindings mode is only a connection possibility for client programs that are running in the same OS image as the Queue Manager, on the server. Note that the application connects (line 19), accesses a queue (line 23), puts a message (line 32), gets it back (line 39) closes the queue (line 45) and disconnects (line 48) all without checking returncodes: the exception-handler takes care of this, and all irregulaties will be reported from the catch MQException block starting at line 50).

The main method does in this case not follow the canonical form, but has 'binary' as an extra option. Option binary can be defined on the command line as an option to the translator, as a program option, as a class option and as a method option. Here the smallest scope is chosen. There is a good reason to make this method a binary method: accessing a queue in MQ Series requires some options that are set using a mask of binary flags - this works, in current NetREXX versions, only in binary mode, because the operators have other semantics in nobinary mode.

```
import com.ibm.mq.

class MessageReader
  properties private

  qManager = "rjtestqm";
  qName     = "TESTQUEUE1"

  method main(args=String[]) static binary

    m = MessageReader()
    do
      MQEnvironment.hostname = 'localhost'
      MQEnvironment.port     = int 1414
      MQEnvironment.channel  = 'CHANNEL1'

      -- exit assignment
      exits          = TimeoutChannelExit()
      MQEnvironment.channelReceiveExit = exits
      MQEnvironment.channelSendExit = exits
      MQEnvironment.channelSecurityExit   = exits

      say "Connecting to QM: " m.qManager
      qMgr = MQQueueManager(m.qManager)

      openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF

      say "Accessing Queue : " m.qName
      queue = qMgr.accessQueue(m.qName, openOptions)

      gmo = MQGetMessageOptions() -- essential here is that we have
         MQGMO_WAIT; otherwise we cannot timeout
      gmo.Options = MQConstants.MQGMO_WAIT | MQConstants.
         MQGMO_FAIL_IF_QUIESCING | MQConstants.MQGMO_SYNCPOINT
      gmo.WaitInterval = MQConstants.MQWI_UNLIMITED

      loop forever
   rcvMessage = MQMessage()
   queue.get(rcvMessage, gmo)
   msgText = rcvMessage.readUTF()
   say "Got a message; the message is: " msgText
   say
      end

    catch ex=MQException
      say "A WebSphere MQ Error occured : Completion Code " ex.
         completionCode "Reason Code " ex.reasonCode
      say "Closing the queue"
      queue.close()
      say "Disconnecting from the Queue Manager"
      qMgr.disconnect()
      say "Done!"
    end
```

In contrast to the previous sample the MessageReader sample only has one import statement. This is always hotly debated in project teams, one school likes the succinctness of including only the top level import, and only goes deeper when there is ambiguity detected; another school spells out the all imports to the bitter end.

The MessageReader sample connects to another queue, called TESTQUEUE1 (specified in line 7) but here we connect in *client mode*, as indicated by lines 13-15 which specify an MQEnvironment. Other options are using an MQSERVER environment variable or a *Channel Definition Table*.

This program is also uncommon in that it uses `MQConstants.MQGMO_WAIT` as an option instead of being triggered as a process by a message on a trigger queue. Using this option means that the program waits (stays active, not really busy polling but depending on an OS event) until a new message arrives, which will be processed immediately.

In lines 18-21 a *Channel Exit* is specified. This exit is show in the following example.

```
import com.ibm.mq.
import java.nio.

class TimeoutChannelExit implements WMQSendExit, WMQReceiveExit,
   WMQSecurityExit

  properties

  tTask = WatchdogTimer
  t = java.util.Timer
  timeout = long
  initialized = boolean

  method TimeoutChannelExit()
    say "TimeoutChannelExit Constructor Called"
    t = java.util.Timer()
    timeout = long 15000

  method channelReceiveExit(channelExitParms=MQCXP, -
              channelDefinition=MQCD, -
              agentBuffer=ByteBuffer) returns ByteBuffer
    do
      this.tTask.cancel() -- cancel the timer task whenever a message
          is read
    catch NullPointerException -- but catch the null pointer the
       first time
    end
    this.tTask = WatchdogTimer()
    this.t.schedule(this.tTask,this.timeout)
    return agentBuffer

  method channelSecurityExit(channelExitParms=MQCXP, -
               channelDefinition=MQCD, -
               agentBuffer=ByteBuffer) returns ByteBuffer
    return agentBuffer

  method channelSendExit(channelExitParms=MQCXP, -
          channelDefinition=MQCD, -
          agentBuffer=ByteBuffer) returns ByteBuffer
    return agentBuffer

class WatchdogTimer extends TimerTask

  method WatchdogTimer()
  method run()
    say  'WATCHDOG TIMER TIMEOUT: HPOpenView Alert Issued' Date()
```

MQ Series has traditional channel exits (programs that can look at the message contents before the application gets to it). In the MQI Java environment there is something akin to this functionality, but a Java channel exit for MQ Series has to be defined in the application, as shown in the previous example. The function of this particular exit is to implement a *Watchdog timer* - on a separate thread, as shown in the sample that follows the sample channel exit. The timer threatens here to have issues a HP OpenView alert, but that part has been left out.

This particular sample has been designed to do something that is normally a bit harder to do: signal the operations department when something does NOT happen - here the assumption is that there is a payment going over the queue at least once every 20 minutes - when that does not happen, an alert is issued. With every message that goes through, the timer thread is reset, and only when it is allowed to time out, action is undertaken.

```
import com.ibm.mq.

class MQPubSubSample

  properties inheritable
  queueManagerName = String
  syncPoint    = Object()
  props        = Hashtable
  topicString     = String
  topicObject     = String
  subscribers     = Thread[]
  subscriberCount  = int

  properties volatile inheritable
  readySubscribers = int 0 --must be defined volatile

  method MQPubSubSample()
    topicString        = null
    topicObject        = System.getProperty("com.ibm.mq.pubSubSample.
       topicObject", "TESTTOPIC")
    queueManagerName = System.getProperty("com.ibm.mq.pubSubSample.
       queueManagerName","rjtestqm")
    subscriberCount  = Integer.getInteger("com.ibm.mq.pubSubSample.
       subscriberCount", 100).intValue()
    this.props      = Hashtable()
    this.props.put("hostname", "127.0.0.1")
    this.props.put("port", Integer(1414))
    this.props.put("channel", "SYSTEM.DEF.SVRCONN")

  method main(agr=String[]) static binary
    sample = MQPubSubSample()
    sample.launchSubscribers()

    /*
     *  wait until all the subscriber threads have finished the
        subscription
     */
    do protect sample.syncPoint
      loop while sample.readySubscribers < sample.subscriberCount
      do
        sample.syncPoint.wait()
      catch InterruptedException
      end
      end -- loop while sample
    end -- do

    sample.doPublish()

  method launchSubscribers()
    say "Launching the subscribers"
    subscribers = Thread[subscriberCount]

    threadNo = int 0
    loop while threadNo < this.subscribers.length
      this.subscribers[threadNo] = MQPubSubSample.Subscriber("
          Subscriber" threadNo)
      this.subscribers[threadNo].start()
      threadNo = threadNo + 1
    end

  method doPublish() signals IOException
    say "method doPublish started"
    destinationType = int CMQC.MQOT_TOPIC
    do
      queueManager  = MQQueueManager(this.queueManagerName, this.
          props)
      messageForPut = MQMessage()
      say "***Publishing ***"
      messageForPut.writeString("Hello world!")
      queueManager.put(destinationType, topicObject, messageForPut)
    catch e=MQException
      say "Exception while publishing " e
    end

  class MQPubSubSample.Subscriber binary dependent extends Thread

    properties private
    myName = String
    openOptionsForGet = int CMQC.MQSO_CREATE | CMQC.
        MQSO_FAIL_IF_QUIESCING | CMQC.MQSO_MANAGED | CMQC.
        MQSO_NON_DURABLE

  method Subscriber(subscriberName=String)
    super(subscriberName)
    myName = subscriberName

  method run()
    do
      say myName " - ***Subscribing***"
      queueManager = MQQueueManager(parent.queueManagerName, parent.
          props)
      destinationForGet = queueManager.accessTopic(parent.topicString
          , parent.topicObject, CMQC.MQTOPIC_OPEN_AS_SUBSCRIPTION,
          openOptionsForGet)

      do protect parent.syncpoint
   parent.readySubscribers = parent.readySubscribers + 1
   parent.syncPoint.notify()
      end

      mgmo = MQGetMessageOptions()
      mgmo.options = CMQC.MQGMO_WAIT
      mgmo.waitInterval = 30000
      say myName " - ***Retrieving***"
      messageForGet = MQMessage()

      do protect getClass()
   destinationForGet.get(messageForGet, mgmo)
      end

      messageDataFromGet = String messageForGet.readLine()
```

```
      say myName " - Got [" messageDataFromGet "]"

    catch e=Exception
      say myName " " e
      e.printStackTrace()
    end
    parent.readySubscribers = parent.readySubscribers - 1
```

This sample shows the publish-subscribe interfaces that at some time have been added to the product. This specific sample shows some Java thread complexity but is a good example of doing publish/subscribe work in a multithreaded way, which is a natural fit for this type of work.

23

MQTT

23.1 Pub/Sub with MQ Telemetry

Publish/subscribe (pub/sub) is a model that lends itself very well to a number of one publisher, many subscriber type of applications; the tools to enter this technology have never been as available as they are now. Also, MQTT is a small protocol that needs to be taken seriously: Facebook has recently become one of the largest users.

Designed as a low-overhead on-the-wire protocol for brokers in the Internet-of-things age, MQTT is an exciting new development in the Messaging and Queueing realm. It is a good choice for any broker functionality, as the minimal message overhead is 2 bytes, but the maximum messages size, in one of the more popular open source brokers is a good 250MB, which give you a message size that is a lot higher than anything possible in the early years of MQ Series back in the nineties. It is now possible to do development with an entry level, entirely open source suite, and scale up to commercial, clustered and highly available implementations when needed, since the protocol has is supported by the base IBM WebSphere MQ product and is an added deliverable in WSMQ 7.5, after being available as an installable add-on for several years.

Here I will show how extremely straightforward it is to create a pub/sub application using this technology. These examples use NetRexx, the Eclipse PAHO Java client library and the open source Mosquitto broker; all these components are completely free and open source. I have installed Mosquitto on my MacBook using the brew system(fn), which makes it as much trouble as “sudo brew install mosquitto”. NetRexx is an excellent language for these examples, as it is compact and avoids the C-inspired ceremony of Java language syntax; if your project requires Java, you can just save the generated Java source (using the new –keepasjava option).

Mosquitto(fn) is written by Roger Light as an open source equivalent of IBM’s rsmb (real small message broker) example application, which is free but lacks source code. It is a small broker application that nevertheless runs production sized workloads. As MQTT, as opposed to the MQI or JMS API’s you use when developing a messaging application, is an on-the-wire protocol (commercial messaging systems tend to have their own, unpublished, on-the-wire protocols), we need an API to use it. This API consists of a set of calls that do the formatting of the messages to the requirements of the on-the-wire protocol for you. The messages themselves are just byte-arrays, which gives you the ultimate freedom in designing their content. It is not unusual for connected devices to encode their information in a few bits; on the other hand, there is no reason not to use extreme verbosity in messages; as long as you send the .getBytes that your String yields, MQTT will send it. When encoding information in a compact way, the protocol design will really pay off, because the protocol overhead, in comparison with http and other chatty protocols, is very low. A limited set of quality of service options (qos) will indicate if you want send and pray, acknowledged delivery or acknowledged one-time-only delivery.

The API library that was chosen for these examples is that from the Eclipse PAHO project. This project, which is in its early stages, has C, Javascript and Java client libraries available. I chose the Java client because the JVM environment is where most of the organizations that I work for will use it. The PAHO Java client library is donated by IBM and written by Dave Locke; it is in active development. If you want to see how the protocol moves in packets over the network, I can recommend Wireshark, which does a good job of recognizing them (if you run on the standard port 1883) and showing you the message types (like ACK) and their bytes.

After having put the NetRexx(.jar) and paho client jars on your classpath, you are good to go. The first example here is the publisher – this is not a fragment, but the complete code. For production code we might add some more checks, as enterprise environments always are prone to suddenly run low on disk space and suffer missing authorizations, but it works as it stands. Do note that you do not have to define a message topic in advance – just think of one any use it, at least if you are in your own environment. With Mosquitto, there wasn't anything to define in advance, and the running Publisher (happily lifted from the Java example) in NetRexx was actually the first time I talked to Mosquitto on my MacBook.

```
import java.sql.Timestamp
import org.eclipse.paho.client.mqttv3.

class Publish implements MqttCallback

  method Publish()
    conOpt    = MqttConnectOptions()
    conOpt.setCleanSession(0)
    tmpDir    = System.getProperty("java.io.tmpdir")
    dataStore = MqttDefaultFilePersistence(tmpDir)
    clientId  = MqttClient.generateClientId()
    topicName = "/world"
    payload   = "hello".toString().getBytes()
    qos        = 2

    do
      broker    = "localhost"
      port   = "1883"
      brokerUrl    = "tcp://"broker":"port
      client    = MqttClient(brokerUrl,clientId, dataStore)
      client.setCallback(this)
    catch e=mqttException
      say e.getMessage()
      e.printStackTrace()
    end -- do

    client.connect()
    log("Connected to "brokerUrl" with client ID "client.getClientId
       ())

    -- Get an instance of the topic
    topic = client.getTopic(topicName)

    message = MqttMessage(payload)
    message.setQos(qos)

    -- Publish the message
    time = Timestamp(System.currentTimeMillis()).toString()
    log('Publishing at: 'time' to topic "'topicName'" with qos 'qos)
    token = topic.publish(message)

    -- Wait until the message has been delivered to the server
    token.waitForCompletion()

    -- Disconnect the client
    client.disconnect()
    log("Disconnected")


  method log(line)
    say line

  method messageArrived(t=String,m=MqttMessage)
    log("Message Arrived: " t m)

  method deliveryComplete(t=IMqttDeliveryToken)
    log("Delivery Complete: " t)

  method connectionLost(t=Throwable)
    log("Connection Lost:" t.getMessage())

  method main(args=String[]) static
    Publish()
```

Topics can have a hierarchical organization; this structure is put in by composing trees of topics, which are strings separated by '/'. In this way, it is easy to compose a /news/economics/today topic string that gives some structure to the publication. The classification is entirely up to the designer.

Messaging in its original form is an asynchronous technology, and for this reason the API offers a callback option, where the callback receives the results of your publish action in an asynchronous way. The broker assigns a message id which you receive back.

The second source fragment (and again, it is no fragment but the entire application program) shows the subscriber.

```
import java.sql.Timestamp
import org.eclipse.paho.client.mqttv3.

class Subscribe implements MqttCallback

  properties private
```

```
client = MqttClient
conOpt    = MqttConnectOptions()
tmpDir    = System.getProperty("java.io.tmpdir")
clientId  = MqttClient.generateClientId()
topicName = "/world"
qos       = 2

method Subscribe()
  do
    connectAndSubscribe()
  catch mqx=MqttException
    log(mqx.getMessage())
  end
  -- Block until Enter is pressed
  log("Press <Enter> to exit");
  do
    System.in.read()
  catch IOException
  end

  -- Disconnect the client
  client.disconnect()
  log("Disconnected")

method connectAndSubscribe() signals MqttSecurityException,
   MqttException
  conOpt.setCleanSession(1)
  dataStore = MqttDefaultFilePersistence(tmpDir)
  do
    broker     = "localhost"
    port    = "1883"
    brokerUrl     = "tcp://"broker":"port
    client = MqttClient(brokerUrl,clientId, dataStore)
    client.setCallback(this)
  catch e=mqttException
    say e.getMessage()
    e.printStackTrace()
  end -- do

  this.client.connect()
  log("Connected to "brokerUrl" with client ID "client.getClientId
     ())

  -- Subscribe to the topic
  log('Subscribing to topic "'topicName'" qos 'qos)
  this.client.subscribe(topicName, qos)

method log(line)
  say line

method messageArrived(t=String,m=MqttMessage)
  log("Message Arrived: " t m)

method deliveryComplete(t=IMqttDeliveryToken)
  log("Delivery Complete: " t)

method connectionLost(t=Throwable)
  do
```

```
    connectAndSubscribe()
    catch mqx=MqttException
      log(mqx.getMessage())
    end

  method main(args=String[]) static
    Subscribe()
```

Security is outside of the scope of this introduction which shows you the source-code of a simple pub/sub application, but in Mosquitto the traffic can be secured using SSL certificates and userid/password combinations; also, the access to topics can be limited. In terms of availability, the Mosquitto configuration file offers an opportunity to send all messages for a defined set of topics to another connected broker, which might be in a different part of the world, or your home, to enable a redundant setup. While the broker does not offer the queue – transmission queue - channel setup with retrying channels that MQ does, the client API has some facilities to locally save the messages and retry if the communication was lost. Also, the last-will-and-testament facility is something that traditional MQ does not have.

24

Component Based Programming: Beans

JavaBeans is the name for the Java component model. It consists of two conventions, for the naming of *getter* and *setter* methods for properties, and the *event* mechanism for sending and receiving events. NetRexx adds support for the automatic generation of getter and setter methods, throught the **properties indirect** option on the properties statement.

25 Interfacing to Scripting Languages

NetRexx contains standardized Java Scripting support, and the NetRexxC.jar file is a self-contained JSR223 scripting engine. This facility opens up a number of possibilities to interface in a standardized manner with several scripting languages and other infrastructure, and offers an easy way for including interpreted NetRexx code in JVM applications. JSR223 is a standard for interacting with scripting languages that consists of:

1. A mechanism to find out for which scripting languages support is available
2. A way to choose one of them
3. An eval() call to dynamically specify and execute a program
4. A *bindings* mechanism to bind variable names to values, to exchange objects with scripts
5. Optionally, a way to execute methods, functions or routines from larger programs
6. Optionally, a way to keep already compiled scripts around for repeated execution (with associated higher performance)

The JSR223 specification[15] details the calls that are available in the `javax.scripting` package. To use the JSR223 interface, Java 6 or higher is required. The JAR file specification defines a service as a well-known set of interfaces and (usually) abstract classes. A service provider is a specific implementation of such a service. For scripting, the service consists of `javax.script.ScriptEngineFactory`. All classes that implement this interface are service providers. Service providers identify themselves by placing a so-called provider-configuration file in META-INF/services. Its filename corresponds to the fully qualified name of the service class, which is `javax.script.ScriptEngineFactory`. Each line of this file contains the fully qualified name of a service provider. The factory class of the NetRexx connector is `org.netrexx.jsr223.NetRexxScriptEngineFactory`. So the file `META-INF/services/javax.script.ScriptEngineFactory` contains one line with exactly this class name.

25.1 Which JSR223 engines are on my system?

The number of JSR223 engines available varies per JVM implementation. The following code can be used to list these.

[15] `http://www.jcp.org/en/jsr/detail?id=223`

```
import javax.script.ScriptEngine
import javax.script.ScriptEngineFactory
import javax.script.ScriptEngineManager

  method main(args=String[]) static
    manager = ScriptEngineManager()
    factories = manager.getEngineFactories()
    it=factories.iterator()
    loop while it.hasNext()
      factory=ScriptEngineFactory it.next()
      f=ScriptEngine factory.getScriptEngine()
      say "className      = " f.getClass.getName
      engineName    = factory.getEngineName()
      engineVersion = factory.getEngineVersion()
      if engineVersion = null then engineVersion = ''
      langName      = factory.getLanguageName()
      langVersion   = factory.getLanguageVersion()
      say "engineName     = " engineName engineVersion langName
         langVersion
      say
    end
```

For example, the Java 8 SE version by Oracle on macOS delivers out of the box:

```
className      =  jdk.nashorn.api.scripting.NashornScriptEngine
engineName     =  Oracle Nashorn 1.8.0_242 ECMAScript ECMA - 262
   Edition 5.1

className      =  com.oracle.truffle.js.scriptengine.
   GraalJSScriptEngine
engineName     =  Graal.js 20.0.0 ECMAScript ECMA - 262 Edition 9

className      =  org.netrexx.jsr223.NetRexxScriptEngine
engineName     =  NetRexx Script Engine V1.0.0 NetRexx 4.01
```

As one can see, the name of the engine, the language and its release are standard features for this query. The NetRexxC.jar file on the classpath adds the NetRexx implementation. There can be any number of additional jar archives on the classpath to deliver engines for different JSR223 implementations for different languages.

25.2 Selecting an engine

When developing a program one is probably interested in using a specific implementation, and it is possible to request the loading of a specific JSR223 engine by name.

```
import javax.script.

manager = ScriptEngineManager()
nrEngine = manager.getEngineByName("NetRexx")
```

The language engine can be selected by its short name, so there is no need to specify the longer name or its version.

25.3 Evaluating a script

This example shows how to do a simple thing that illustrates the value of being able to do this from other environments: calculating some number with *numeric precision* set to some value that other languages cannot handle.

```
/* simple script invocation */
nrEngine.eval('numeric digits 17; say 111111111 * 111111111')
```

The output from this script would be:

```
12345678987654321
```

25.4 Bindings

Bindings are name-value pairs whose keys are strings - they can be of REXX type. Their behavior is defined through the `javax.script.Bindings` interface. As for `ScriptContext`, a basic implementation is provided called `SimpleBindings`. Although bindings belong to script contexts, `ScriptEngine` provides `createBindings()`, which returns an uninitialized binding. Another method, `getBindings()`, exists to return the bindings of a certain scope. There are at least two scopes, `ScriptContext.GLOBAL_SCOPE` and `ScriptContext.ENGINE_SCOPE`. They represent key-value pairs that are either visible to all instances of a script engine that have been created by the same `ScriptengineManager`, or visible only during the lifetime of a certain script engine instance. The following program illustrates the use of bindings to store a value, 42, into the binding called `answer` and then using its retrieved value in the evaluation of the statement `'say "the answer is" answer '`. The next action uses the handle `one` for a value of 1, and uses its retrieved value to add it to the value previously contained in the binding `answer`.

```
import javax.script.

nrEngine = ScriptEngineManager().getEngineByName("NetRexx")

/* simple script invocation */
nrEngine.eval('numeric digits 17; say 111111111 * 111111111')

/* script invocation with bindings */
answer = 42
nrEngine.put("answer", answer)
nrEngine.eval('say ''the answer is ''answer')

one = 1
nrEngine.put("onemore",one)
nrEngine.eval('say ''one more is ''answer+onemore')
```

Note that in line two, the invocation is shortened a bit by getting rid of the intermediate `manager` object for instantiation of the language interface. Also note that in line 10, we chose, for illustration purposes, to store the `one` object into the bindings structure using a different name, `onemore`. This shows that the string used as identifier for the object is just a handle to it, and nothing more. This would yield:

```
12345678987654321
the answer is 42
one more is 43
```

The different possibilities and language combinations will be discussed in the paragraphs below.

25.4.1 Obtaining a returncode

The variable binding used for the return code from the NetRexx program is called `returnobject`. This program illustrates its use:

```
import javax.script.

nrEngine = ScriptEngineManager().getEngineByName("NetRexx")

/* check returncode */
say nrEngine.eval('NetRexxScriptEngine.instance.put("returnobject",
   "99")')
```

```
99
```

25.5 Interpreted execution of NetREXX scripts from jrunscript

Another way of calling any NetRexx program, for interpretation, is to use the standard jrunscript executable that is included in Java 1.6 and beyond. For example, in the examples/rosettacode directory, one could specify:

```
jrunscript -l netrexx -cp $CLASSPATH -f RCSortingHeapsort.nrx
```

The -l option instructs the jrunscript handler to choose NetRexx as its standard scripting language. For NetRexx to be eligible as a scripting language, NetRexxC.jar must be on the jrunscript classpath, which is a separate classpath from the standard one. In this setup, even NetREXX programs with a filename that is not valid as a classname, can be executed as an interpreted script.

25.6 Using JavaScript from NetREXX programs

JavaScript support is built in from Java 1.6 onwards, and using it does not require placing another library on the classpath. Using JavaScript from NetREXX can have benefits, for example when using types native to JavaScript, like the JSON data interchange format.

```
import javax.script.

jsEngine = ScriptEngineManager().getEngineByName("JavaScript")

jsEngine.eval('var foo = {};')
jsEngine.eval('foo.foundation = "RexxLA";')
jsEngine.eval('foo.model = "open";')
jsEngine.eval('foo.week = 42;')
jsEngine.eval('foo.transport = "car";')
jsEngine.eval('foo.month = 7;')

jsEngine.eval('bar = JSON.stringify(foo);')

jsonString = jsEngine.get('bar')
say jsonString
```

which yields the following result:

```
{"foundation":"RexxLA","model":"open","week":42,"transport":"car","
   month":7}
```

25.7 Using AppleScript on macOS

On macOS you can run an AppleScript using NetRexx.

```
import javax.script.
-- does not work in recents macos versions

/*
Instead of ScriptEngine engine = mgr.getEngineByName("AppleScript");
   you must use:
ScriptEngine engine = mgr.getEngineByName("AppleScriptEngine");

In your src directory create directory META-INF
In your src directory create directory META-INF/services
Create file META-INF/services/javax.script.ScriptEngineFactory
In this file is one line:
apple.applescript.AppleScriptEngineFactory
*/

appleEngine = ScriptEngineManager().getEngineByName("
   AppleScriptEngine")
context = appleEngine.getContext()
bindings = context.getBindings(ScriptContext.ENGINE_SCOPE)
bindings.put("javax_script_function", "getName")
bindings.put(ScriptEngine.ARGV, 'Stranger')

appleScript = 'on getName(default_) \n'-
         'tell application "Finder" \n'-
         'display dialog "What is your name?" default answer default_
             with icon note \n'-
         'set myName to the text returned of the result \n'-
         'delay 0.5 \n'-
         'display dialog "Hi there, " & myName & "! Welcome to
            AppleScript!" with icon note \n'-
         'end tell\n'-
         'return myName\n'-
              'end getName'

result = appleEngine.eval(appleScript,context)
say result
```

The AppleScript interpreter expects end-of-line characters at the end of every line, so make sure to include them in your script. The above script shows it is fairly straightforward to put a dialog box with a question on the screen. The example shows how to give an argument (ARGV) to a method, and how to put the method name in the bindings object in order to return the result upon evaluation.

25.8 Execution of NetRexx scripts from ANT tasks

The jsr223 engine enables us to execute NetRexx scripts from the ant[16] building tool using the `<script>` tag. This was already possible using the BSF library, where NetRexx was one of the originally supported languages, but has become more straightforward with jsr223 scripting.

```
<project name="MyProject" basedir=".">
  <description>
    demonstration of ant jsr223 netrexx scripting
  </description>

  <property name="divider" value="81" />
  <script language="netrexx" manager="javax">
    say "100/"divider '= ' 100/divider
  </script>
</project>
```

Note that properties can be set in other parts of the ant xml file and used in the ant script. This script yields the following output:

```
Buildfile: /Users/rvjansen/apps/netrexx-code/documentation/pg/
   antscript.xml
   [script] 100/81 =  1.2345679

BUILD SUCCESSFUL
Total time: 0 seconds
```

The task may use the BSF scripting manager or the JSR 223 manager that is included in JDK6 and higher. This is controlled by the manager attribute. The JSR 223 scripting manager is indicated by ”javax”, as shown on line 7.

All items (tasks, targets, etc) of the running project are accessible from the script, using either their name or id attributes (as long as their names are considered valid Java identifiers, that is). This is controlled by the ”setbeans” attribute of the task. The name ”project” is a pre-defined reference to the Project, which can be used instead of the project name. The name ”self” is a pre-defined

[16] `http://ant.apache.org`

reference to the actual <script>-Task instance. From these objects you have access to the Ant Java API.

A classpath for execution of the script can be set using the `classpath` attribute. A script contained in a separate file can be executed using the `src` attribute.

25.9 Integration of NetRexx scripting in applications

Several applications offer a facility to script functionality using the javax.scripting interface, akin to the way applications use the RexxSAA interface for this purpose.

25.10 Interfacing between ooRexx and NetRexx using BSF4ooRexx

BSF is a system for language interaction that originated in a research project at IBM, and predates JSR223 (and certainly its implementation in Java 6) for a number of years. BSF 2.x has its own interface, while modern BSF versions are an implementation of the JSR223 interfaces. BSF4ooRexx enables a bidirectional interface between ooRexx and Java, and enables one to use the large class library support for Java in ooRexx programs, but likewise the execution of ooRexx code from Java (including NetRexx) programs. BSF4ooRexx contains some special support for JVM programs written in NetRexx.

25.11 General jsr-223 Implementation Notes

This section describes some notes pertaining to specific jsr223 for NetRexx design and implementation decisions.

- All engine scope bindings are passed to the script as variables - note that binding names containing periods have the periods changed to underscores to be legal variable names.
- The NetRexx script engine is reused unless the script returned via an "exit" statement and the bindings are persistent which means that scripts will see the bindings (Objects) created by previous scripts
- Arguments are passed both as the normal `arg` string and as the array binding `javax.script.argv` i.e. script variable `javax_script_argv`.
- Scripts are executed via the NetRexxA API for interpreting a program from a string so they are not written to files.
- The current version of the engine has no other optimization and only support for bare minimum JSR223 features (No compilable, invokeable, preparse or caching or user profiles or console, etc.).
- When running as an Ant Script task, properties whose names contain periods are not passed to the bindings and must be accessed via project.getProperty('some.name'

The workaround is to define a local Ant property as a global first and the scriptengine will overlay the global value with the local value in the bindings map

- When running as an Ant Script task, properties can be set via project.setProperty('some.name 'some value')
- Script parms can be passed in an "arg" binding. Parse flags can be passed with a "netrexxflags" binding or in Ant with the usual "ant.netrexx.verbose", etc properties.
- Ant scripts can use the nested classpath facility - It is automatically added to the classpath that NetRexx scans. Likewise any path segments from a thread context URLclassloader are added.
- The engine will run programs (ie that have a main class) as well as scripts but bindings cannot then be auto added to the program namespace so programs have to load bindings like this: `NetRexxScriptEngine.getObject("objectname")`

26

NetRexx Tools

26.1 Editor support

This chapter lists editors that have plugin support for NetRexx, ranging from syntax coloring to full IDE support (specified), and Rexx friendly editors, that are extensible using Rexx as a macro language (which can be the first step to provide NetRexx editing support).

26.1.1 JVM - All Platforms

JEdit	Full support for NetRexx source code editing, to be found at `http://www.jedit.org`.
NetRexxDE	A revisions with additions of the NetRexx plugin for jEdit, moving to a full IDE for NetRexx. `http://kenai.com/projects/netrexx-misc`
Eclipse	Eclipse has a NetRexx plugin that provides a complete IDE environment for the development of NetRexx programs (in alpha release) by Bill Fenlason. The project is situated at SourceForge (`http://eclipsenetrexx.sourceforge.net/`).

26.1.2 Linux

Emacs	netrexx-mode.el (in the NetRexx package in the `tools` directory) runs on GNU Emacs, which is installed by default on most Linux developer distributions.
vim	vi with extensions

26.1.3 MS Windows

Emacs	netrexx-mode.el (in the NetRexx package in the `tools` directory) runs on GNU Emacs for Windows. `http://www.gnu.org/software/emacs/windows/faq.html`.
vim	vi with extensions

26.1.4 macOS

Aquamacs	A version of Emacs that is integrated with the macOS Aqua look and feel. (`http://www.aquamacs.org`). NetREXX mode is included in the NetREXX package in the `tools` directory.
Emacs	netrexx-mode.el (in the NetREXX package) runs on GNU Emacs for macOS. `http://www.gnu.org/software/emacs`.
Vim	Vi with extensions

26.2 Java to Nrx (java2nrx)

When working on a piece of Java code, or an example written in the language, sometimes it would be good if we could see the source in NetREXX to make it more readable. This is exactly what *java2nrx* by Marc Remes does. It has a Java 1.5 parser and an Abstract Syntax Tree that delivers a translation to NetRexx, to the extend of what is currently supported under NetRexx.

At the moment it is to be found at `gitclonegit://git.code.sf.net/p/netrexx/codenetrexx-code` in the tools directory.

It is started by the `java2nrx.sh` script; for convenience, place `java2nrx.sh` and `java2nrx.jar` in the same directory. NetRexxC and java must be available on the path.

Usage: Alternatively:

FIGURE 2: Java2nrx 1

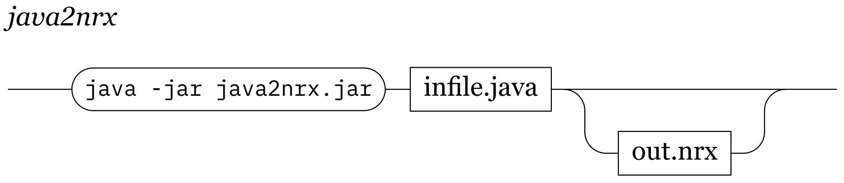
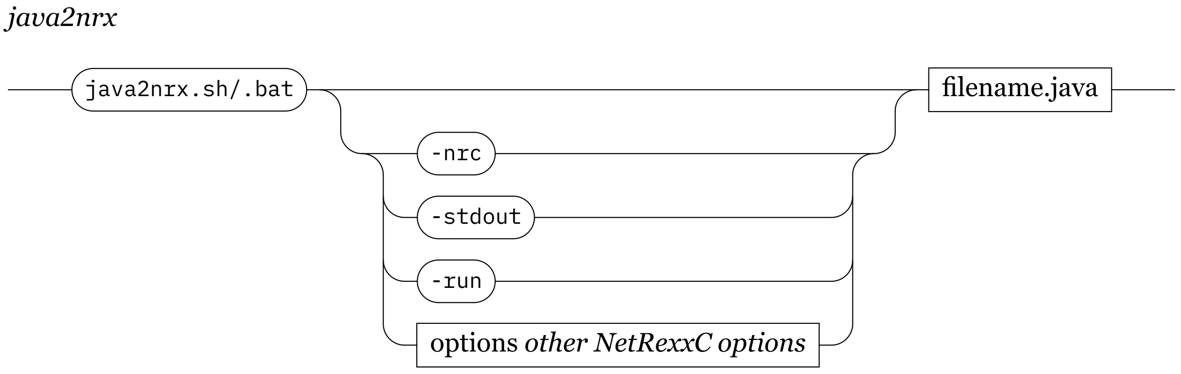


FIGURE 3: Java2nrx 2



-nrc runs NetRexxC compiler on output nrx file
-stdout prints NetRexx file on stdout
-run runs generated translated NetRexx output file

27

Using Eclipse for NetRexx Development

This is a guide for first time Eclipse users to set up a NetRexx development project. It is not a beginners guide to Eclipse, but is intended to explain how to download the NetRexx compiler source from SVN to be able to modify and build it using Eclipse[17].

It is detailed and hopefully foolproof for someone who has never used Eclipse. It assumes a Windows user, but if you are a Linux or Mac user, you will no doubt understand what to do.

This guide is for Eclipse 4.2 (Juno), written August, 2012. New Eclipse releases occur every 4 months, so there may be differences depending on what the current version is.

27.1 Downloading Eclipse

There are many different preconfigured versions of Eclipse. As you become more experienced with it you may wish to use a different distribution, but the one specified here makes some things simple. It does contain some things that you may never use.

1. Make a new folder for the project. Name it appropriately (e.g. EclipseNetRexx)
2. Browse to eclipse.org, and click on "Download".
3. Download the version named ECLIPSE IDE FOR JAVA DEVELOPERS for your your operating system.
4. The download is about 150 MB.
5. Unzip the downloaded file into your project folder.

27.2 Setting up the workspace

There are different strategies for managing Eclipse workspaces. Eclipse defaults to putting the workspace in your Windows documents folder - probably not what you want to do. The following is perhaps the most simple way.

[17] If you have questions or comments, feel free to contact Bill Fenlason at billfen@hvc.rr.com.

1. Open the project folder. It will now contain a folder named eclipse.
2. Add a new folder named "workspace" in the project folder to go along with the eclipse folder.
3. Open the eclipse folder, and create a shortcut to eclipse.exe.
4. Move the shortcut to the desktop and rename it to something like "Eclipse NetRexx".
5. Close the project folder, and double click the shortcut to start Eclipse.
6. The "Select a workspace" dialog comes up - don't use the default.
7. Browse to the workspace folder that you just created and select it.
8. Click (check) the "Use this as the default" box, and click OK.

27.3 Shellshock

If you have never used Eclipse, it can be a bit overwhelming. It is rather complicated, and has endless options, etc. In addition there are at least a thousand different plugins.

You will be greeted by a Welcome screen - you may find it interesting or boring. Exit from it via tback to the welcome screen from: Main Menu -> Help -> Welcome.

27.4 Installing Git

Modern versions of Eclipse come with Git support built in. If not, install it from the Eclipse Marketplace.

27.5 Downloading the NetRexx project from the Git repository

The Git repository on SourceForge contains the NetRexx compiler/translator, documentation, examples, etc. These instructions assume you want only the compiler project.

1. The NetRexx Git repository clobe command is: `gitclonegit://git.code.sf.net/p/netrexx/codenetrexx-code`
2. Copy it (for pasting) from above, or get it from the kenai or netrexx.org site.

27.6 Setting up the builds

Ant support is built into Eclipse, but it must be configured to be able to access the bootstrap NetRexx compiler.

1. Double click on the build.xml file name in the package explorer. Note that its icon is an ant.
2. The build file will open in an editor window.
3. Right click in the window to bring up a context menu, and select Run As -> 2 Ant Build
4. Do NOT select 1 Ant Build.
5. The Ant configuration dialog comes up - it will show you all the targets, etc.
6. Click on the Classpath tab, and then click on User Entries.
7. Now click on Add External Jars to bring up the Jar Selection dialog.
8. Navigate to the lib folder in the project folder. Make sure you are not in the build folder.
9. Double click on NetRexxC.jar to select it.
10. Click on the Refresh tab, and check the Refresh resources on completion box.
11. Click Run to build the distribution. The messages will appear in the console listing below.
12. The java doc step may fail.
13. Close the build.xml file (X on the tab).

You can configure the ant build by using the configuration dialog in Run As -> 2 Ant Build. You may want to check “compile” and “jars” to run those steps. Use Apply to save the configuration.

There are two different builds. The second build.xml file is in the project -> tools -> ant-task folder. Open it up and repeat the above steps for that build.xml file. Each build file has its own ant configuration, and once set selecting Run As -> 1 Ant Build will run it. Or just hit F11.

27.7 Using the NetRexx version of the NetRexx Ant task

The above process uses the standard NetRexx Ant task, not the new one. To use the new one:

1. Main Menu -> Window -> Preferences -> Ant -> Runtime.
2. Open up and select Ant Home Entries. Then click on Add External Jars
3. Navigate to the lib folder in the project and select ant-netrexx.jar
4. The jar will appear at the bottom of the list.
5. Use the UP button to move it up (ahead) of the apache ant version, click OK

27.8 Setting up the Eclipse NetRexx Editor Plugin (Optional)

The NetRexx Editor plugin provides syntax coloring and error checking for nrx files, as well as one click compiling and translating.

1. Click on Main Menu -> Help -> Eclipse MarketPlace.
2. Type NetRexx in the search box and hit enter.
3. Click the Install button next to the Eclipse NetRexx package.
4. Click Next, Accept the License, Finish, OK to unsigned content, and Yes to restart Eclipse.
5. Click Main Menu -> Window -> Preferences -> NetRexx Editor to explore it

28

Platform dependent issues

28.1 Mobile Platforms

Android™is a version of Linux with a runtime consisting of a variant of Java, and is friendly to NetRexx programs. Indeed, with NetRexx performing so much better than the closest competition (jRuby, jython) on these devices, there might be a bright future for NetRexx in these environments.

However, there are some drawbacks, caused by the security architecture put in place. Free, unfettered programming like one can do on a desktop machine is a rare occurrence on these devices, and to get programs running on them requires some knowledge of the security architecture that has been put in place for mobile operating systems.

While Apple development still employs a closed model that allows programming only by buying a license with accompanying certificates, and vetting by the App Store employees, and an assumption you will program in Objective-C, Android allows programming but not as straightforward as we know it. To make simple command-line NetRexx programs, both device types need to be *rooted* to allow optimal access. Android allows the installation of applications without vetting by third parties, but dictates a programming model that incurs some overhead - which is a drawback for the occasional scripter.

28.1.1 Android

The security model of Android is based on *least needed privilege* and is implemented by assigning each application a different userid, so that applications on the same device (be it a phone or a tablet) cannot get to each others data. The consequence of this is that simple NetRexx programming and scripting

28.1.2 Apple IOS

There is a number of ways NetRexx can be run on Apple IOS devices (iPhone and iPad). Both have drawbacks. With ISH, a 32-bit version of Linux is started on an emulated X86 processor; this has dire consequences for performance. The 'Jailbreak' solution runs with much better performance, but this approach is rather volatile and cannot be guaranteed to be feasible in the future, because Apple is actively fixing the holes that allow it.

ISH

The ISH application delivers a Linux shell on emulated hardware. The NetRexxC.jar can be transferred with scp to the storage of ISH, from where it can be run. It needs higher memory heap allocations than the standard; -Xms128M -Xmx128M is recommended here. Do not expect performance corresponding to the native ARM hardware in your device.

Jailbreak

Note: this chapter is out of date. Nonewithstanding the current intention of Apple to only allow Swift an d Objective-C as programming languages on the iPhone and iPad, NetREXX on IOS works fine. This is what one should do to make it work:

1. Jailbreak[18] the device. This is necessary until a more sensible setup is used. I used Spirit; it synchs the phone with the hack and then Cydia is installed, an application that does package management the Debian way
2. Choose the ”developer profile” on Cydia when asked. This applies a filter to the packages shown (or rather it doesn't) - but you need to do it in order to see the prerequisites
3. OpenTerminal will help you to do command line operations on the phone itself
4. The prerequisites are a Java VM (JamVM installs a VM and ClassPath, the open Java implementation) and Jikes, the Java compiler written in C and compiled to the native instruction set of the phone, which is ARM - most processors implementing this have *Jazelle*, a specials instructionset to accelerate Java bytecode. However, this feature is seldom used.

The phone can also be logged on to using ssh from your desktop. Do not forget to change the password for the 'root' user and the 'mobile' user, as instructed in the Cydia package. Note that this type of information will can be made inaccurate very swiftly.

When this is done, NetRexxC.jar can be copied to the phone. I did this using 'scp NetRexxC.jar mobile@10.0.0.76:' (use the password you just set for this userid) (and because my router assigned 10.0.0.76 to the phone today). I crafted a small 'nrc' script that does a translate and then a Java compile using jikes (and I actually wrote this on the phone using an application called 'iEdit' - nano, vim and other editors are also available but I found the keyboard scheme to type in ctrl-characters a bit tedious - you type a 'ball' character and then the desired ctrl char, while shifting the virtual keyboard through different modes):

nrc:

```
java -cp ~/NetRexxC.jar COM.ibm.netrexx.process.NetRexxC $*
```

[18]Note that jailbreaking an iPhone is against Apple's End Use License Agreement) and might be illegal in some jurisdictions.

Now we can do a compile of the customary hello.nrx with './nrc -keep -nocompile hello' (notice that this is all in the home directory of the 'mobile' user, just like the jar that I just copied. The resulting hello.java.keep can then be mv'ed to hello.java and compiled with 'jikes hello.java'. This produces a class that can be run with 'java -cp NetRexxC.jar hello'

28.2 IBM Mainframe: Using NetRexx programs in z/OS batch

Traditionally the mainframe was a batch oriented environment, and much of the workload that counts still executes in this way. To be able to use NetRexx with Job Control Language (JCL) in batch address spaces, accessing traditional datasets and interacting with the console when needed, we need to know a bit more. This will be explained in these paragraphs.

A standard component of z/OS since version 1.8 or so is `jzos`, which acts as glue between the unix-like abstractions the JVM works with and the time tested way of working on z/OS, with its SAM and VSAM datasets, its Partitioned Data Set (PDS) file organization, the ICF Catalogs and console address space; all of which in existence long before Java reared its head in our IT environments.

The manuals will teach you that there are several ways to interact with HFS/OMVS resources in JCL, but the alternatives to `jzos` have so many drawbacks that it is the only sensible way to run NetRexx programs in the batch environment.

28.2.1 Example

```
  //AB2217N1 JOB (7355,710,TC78JAN),'PGM',MSGCLASS=X,NOTIFY=AB2217
//JAVA EXEC PROC=JVMPRC60,
// JAVACLS='HelloWorld'
//STDENV DD *
. /etc/profile
export JAVA_HOME=/usr/lpp/java/J6.0
export PATH=/bin:"${JAVA_HOME}"/bin
LIBPATH=/lib:/usr/lib:"${JAVA_HOME}"/bin
LIBPATH="$LIBPATH":"${JAVA_HOME}"/lib/s390
LIBPATH="$LIBPATH":"${JAVA_HOME}"/lib/s390/j9vm
LIBPATH="$LIBPATH":"${JAVA_HOME}"/bin/classic
export LIBPATH="$LIBPATH":
APP_HOME=$JAVA_HOME
CLASSPATH=$APP_HOME:"${JAVA_HOME}"/lib:"${JAVA_HOME}"/lib/ext
for i in "${APP_HOME}"/*.jar; do
    CLASSPATH="$CLASSPATH":"$i"
    done
export CLASSPATH="$CLASSPATH":
IJO="-Xms16m -Xmx128m"
export IBM_JAVA_OPTIONS="$IJO "
//
```

29

Building the NetRexx translator

It is easy to build the NetRexx translator from source. Prerequisites are:

1. A Java Virtual Machine
2. A Git client

NetRexx can be built on all platforms that it runs on. NetRexx has been bootstrapped since 1996 and subsequently has been used to compile itself. Every checkout of the source code contains the 'bootstrap' compiler, which is normally the previous release version. Only the official release branches contain the same release of the compiler - to prove that it still can compile itself on release. Theoretically, it is possible to break things by introducing changes that preclude the compiler to compile itself - it is our job that these changes are not released to a wider audience, but rolled back in time.

29.1 Repository

The NetRexx source code repository is hosted at the SourceForge Git repository. To get the code on your system, you should register at the NetRexx project at SourceForce and clone the repository using Git. For this version management package there are many graphical user interfaces, but what is shown here, is the command line version. Choose a suitable place as working directory - you can later move it around as you please.

```
git clone git://git.code.sf.net/p/netrexx/code netrexx-code
```

Note: This will checkout the whole repository to your local system; including previous versions, experimental branches and personal sandboxes of other developers.

The master branch contains the most current version of the source code, including the documentation, examples and test cases.

29.2 The buildfile

The official buildfile is called `build.xml` and the `ant` utility is used for building NetRexx from source. This file contains a number of tasks. To build the translator, make sure that the top level directory that is cloned from git is the current directory, and issue the command:

```
java -jar ant/ant-launcher.jar compile
```

followed by

```
java -jar ant/ant-launcher.jar jars
```

This will build the compiler from source and create a `build` directory in the current directory. In `build/lib` the NetRexxC and NetRexxR jars are put by the archiving process that is started by the `jar` task. These new jars can be used immediately, by having them (NetRexxC will suffice) on the classpath.

29.3 Testing

Currently, there are two locations that contain the tests. The first is the `org.netrexx.process.diag` package, which currently is being integrated into the `trunk/test` directory. This directory contains, in addition to the traditional "diag" tests that have been modified to run under jUnit, some of the tests for the newer functionality. These tests are accessible using a `make` process that uses `makefile` as its build build file. The command

```
make test
```

will compile and run the tests; jUnit will report on progress and results.

30

The NetRexx Workspace - nrws

A read-evaluate-print loop, or REPL, is a very popular way for users to familiarize themselves with the language[19] and design and/or prototype programs. Martin Lafaix has contributed such a facility already in the year 2000, but the inclusion of his *Workspace for NetRexx* took some time. The JSR-199 scripting facility, which was added to the distribution earlier, could do something akin to this, but could not remember variable values over executions. The requirement to fix this issue, and the wish to have some facility that can execute Pipes for NetRexx in the fastest possible way, led to the resurrection of this nearly 20-year old code, with some updates for command history (up- down arrowing through it) and -editing, included multiline-editing. The NetRexx workspace has a requirement of Java 8. 3.08

30.1 Installation

nrws is included in both NetRexxF.jar and NetRexxC.jar. Wherever NetRexx works, its workspace will work. It is advisable to have a shortcut for starting it. In the bin directory (for windows users) a *nrws.bat* batchfile can be found. In that same directory a *.bash_aliases* file can be found, which adds a nrws command for unixlike systems like Linux and macOS. Both are short forms of running *java org.vpad.extra.workpad.Workspace*.

30.2 Starting nrws

To begin using Workspace for NetRexx, issue the command *nrws* to the operating system shell. There is a brief pause, some start-up messages, and then the first frame appears.

The standard prompt (which can be modified in various ways, through the *nrws.properties* file in the home directory) has a left and a right component. On the left side, the default is nrws>. On the right side, the default is that that current computation step in the current *frame* is indicated. The concepts of computation step and frame will be explained shortly. It is also possible to have an indication of the elapsed time for the last command in the righthand prompt.

When you want to enter input to Workspace for NetRexx, you do so on the same

[19]for example, Python, Ruby, Swift and Elixir have them, and there are used in all introductory literature

line after the left prompt. The ”1” in the right prompt is that computation step number and is incremented after you enter Workspace for NetREXX statements. Note, however, that a system command such as)clear all may change the step number in other ways.

30.3 Exit nrws

To exit from Workspace for NetREXX, type)quit at the input prompt and press the Enter key. It is possible to configure this to display the following message:

```
  Please enter ”y” or ”yes” if you really want to leave the interactive
  environment and return to the operating system.
You should enter yes, for example, to exit Workspace for \nr{}.
```

The is also a)pquit system command that always protects your exit from the workspace.

Because Workspace for NetREXX runs on a number of different machines and platforms, operating system shells and windowing environments, there is no standard appearance. You are to experiment with profiles and schemes for shells; one favourite is dark solarized (shown). You can also change the way that Workspace for NetREXX behaves via system commands described later in this chapter and in Appendix A. System commands are special commands, like)set, that begin with a closing parenthesis and are used to change your environment. For example, you can set a system variable so that you are not prompted for confirmation when you want to leave Workspace for NetREXX.

You are ready to begin your journey into the world of Workspace for NetREXX. Let’s proceed to the first step.

30.4 Exploring the NetREXX language

The NetREXX language is a rich language for performing interactive computations and for building components for the Java libraries. For a full description, please consult the *The NetREXX Language definition*.

30.5 Arithmetic Expressions

For arithmetic expressions, use the ”+” and ”-” operators as in mathematics. Use ”*” for multiplication, ”/” for division, and ”**” for exponentiation. When an expression contains several operators, those of highest precedence are evaluated first. For arithmetic operators, ”**” has highest precedence, ”*” and ”/” have the next highest precedence, and ”+” and ”-” have the lowest precedence.

```
say 1 + 2 - 3 / 4 * 3 ** 2 - 1
-4.75
```

NetREXX puts implicit parentheses around operations of higher precedence, and groups those of equal precedence from left to right. The above expression is equivalent to this.

```
say ((1 + 2) - ((3 / 4) * (3 ** 2))) - 1
-4.75
```

If an expression contains subexpressions enclosed in parentheses, the parenthesized subexpressions are evaluated first (from left to right, from inside out).

```
say 1 + 2 - 3 / (4 * 3 ** (2 - 1))
2.75
```

30.6 Some Types

Everything in NetREXX has a type. The type determines what operations can be performed on an object and how the object can be used. For the following, please keep in mind that sometimes a variable needs to be assigned a type first.

30.7 Symbols, Variables, Assignments, and Declarations

A symbol is a literal used for the input of things like keywords, the name of variables or to identify some algorithm.

A symbol has a name beginning with an uppercase or lowercase alphabetic character, '$', '(Euro)', or '_'. Successive characters (if any) can be any of the above, or digits. Case is by default undistinguished : the symbol points is no different from the symbol Points.

A symbol can be used in Workspace for NetREXX as a variable. A variable refers to a value. To assign a value to a variable, the operator "=" is used. A variable initially has no restriction on the kinds of values to which it can refer.

This assignment gives the value 4 to a variable names x:

```
x = 4
```

To restrict the type of objects that can be assigned to a variable, use a declaration:

```
y = int
```

The declaration for y forces values assigned to y to be converted to integer values. If no such conversion is possible, NetREXX refuses to assign a value to y:

```
y = 2/3
java.lang.NumberFormatException: Decimal part non-zero: 0.666666667
```

A type declaration can also be given together with an assignment. The declaration can assist NetREXX in choosing the correct operations to apply:

```
f = float 2/3
```

Any number of expressions can be given on input line. Just separate them by semicolons.

These two expressions have the same effect as the previous single expression:

```
f = float; f = 2/3
```

30.8 Conversion

Objects of one type can usually be ”converted” to objects of several other types. To convert an object to a new type, prefix the expression with the desired type.

```
say int sin(PI)
0
```

Some conversions can be performed automatically when NetRexx tries to evaluate input. Other conversions must be explicitly requested.

30.9 Calling Functions

As we saw earlier, when you want to add or subtract two values, you place the arithmetic operator ”+” or ”-” between the two arguments denoting the values. To use most of other NetRexx operations, however, you use another syntax: write the name of the operation first, then an open parenthesis, then each arguments separated by commas, and, finally, a closing parenthesis.

This calls the operation sqrt with the single integer argument 120:

```
say sqrt(120)
10.95445115010332
```

This is a call to max with the two integer arguments 125 and 7:

```
say max(125, 7)
125
```

This calls an hypothetical quatern operation with four floating-point arguments:

```
quatern(3.4, 5.6, 2.9, 0.1)
```

If the operation has no arguments, you can omit the parenthesis. That is, these two expressions are equivalent:

```
say random()
```

and

```
say random
```

30.10 Long Lines

When you enter expressions from your keyboard, there will be time when they are too long to fit on one line. Workspace for NetREXX does not care how long your lines are, so you can let them continue from the right margin to the left side of the next line.

Alternatively, you may want to enter several shorter lines and have Workspace for NetREXX glue them together. To get this glue, put an hyphen (-) at the end of each line you wish to continue.

```
say 2 -
+ -
3
```

is the same as if you had entered

```
say 2 + 3
```

Comment statements begin with two consecutive hyphens and continue until the end of the line.

```
say 2 + 3 -- this is rather simple, no?
```

The third way to accomplish this is to use the built-in multiline editing facility. Just press [Esq]-[Enter] to continue with the next line of a multiline block - with the first [Enter] key the whole block will be passed to the Workspace. These multiline blocks can also be recalled and edited with arrow-up.

30.11 Numbers

Workspace for NetREXX distinguishes very carefully between different kinds of numbers, how they are represented and what their properties are.

30.12 Data Structures

Workspace for NetREXX has a large variety of data structures available. Many data structures are particularly useful for interactive computation and others are useful for building applications. The data structures of Workspace for NetREXX are organized into class hierarchies.

A one-dimensional array is the most commonly used data structure in Workspace for NetREXX for holding objects all of the same type. One-dimensional arrays are inflexible—they are implemented using a fixed block of storage. They give equal access time to any element.

Write an array of elements using square brackets with commas separating the elements:

```
a = [1, -7, 11]
```

The index of the first element is zero. This is the value of the third element:

```
say a[2]
11
```

An important point about arrays is that they are *mutable*: their constituent elements can be changed *in place*:

```
a[2] = 5; say a[0] a[1] a[2]
1 -7 5
```

Examples of datatypes similar to one-dimensional arrays are: StringBuffer (arrays of characters), and BitSet (represented by array of bits).

```
say BitSet(32)
{}
```

A list is another data structure used to hold objects. Unlike arrays, lists can contain elements of different non-primitive types. Also, lists are usually flexible.

A simple way to create a list is to apply the operation asList to an array of elements.

A vector is a cross between a list and a one-dimensional array. Like a one-dimensional array, a vector occupies a fixed block of storage. Its block of storage, however, has room to expand! When it gets full, it grows (a new, larger block of storage is allocated); when it has too much room, it contracts.

This creates a vector of three elements:

```
f = Vector(asList([2, 7, -5]))
```

The addAll method inserts a list at a specified point. To insert some elements between the second and third elements, use:

```
f.addAll(2, asList([11, -3])); say f
[2, 7, 11, -3, -5]
```

Vectors are used to implement ”stacks”. A stack is an example of a data structure where elements are ordered with respect to one another.

An easy way to create a stack is to first create an empty stack and then to push elements on it:

```
s = Stack(); s.push(”element1”); s.push(”element2”); s.push(”element3”)
```

This loop extracts elements one-at-a-time from s until the stack is exhausted, displaying the elements starting from the top of the stack and going down to the bottom:

```
loop while \ s.empty; say s.pop; end
element3
element2
element1
```

(!!! to be continued)

30.13 Expanding to Higher Dimensions

To get higher dimensional aggregates, you can create one-dimensional aggregates with elements that are themselves aggregates, for example, arrays of arrays, vectors of sets, and so on.

(!!! to be continued)

30.14 Writing Your Own Functions

Java provides you with a very large library of predefined operations and objects to compute with. You can use the Java Class Libraries to create new objects dynamically of quite arbitrary complexity. Moreover, the libraries provides a wealth of operations that allow you to create and manipulate these objects.

For many applications, you need to interact with the interpreter and write some NetREXX programs to tackle your application. Workspace for NetREXX allows you to write functions interactively, thereby effectively extending the system library. Here I give a few simple examples, leaving the details to The NetREXX Language reference manual and related publications.

We begin by looking at several ways that the *factorial* function can be defined. The first way is to use an if-then-else instruction.

```
method fact(n) static; if n < 3 then return n; else return n*fact(n-1)

say fact(50)
30414093201713378043612608166064768844377641568960512000000000000
```

A second definition directly uses iteration.

```
method fac(n) static; a = 1; loop i = 2 to n; a = a * i; end; return a

say fac(50)
30414093201713378043612608166064768844377641568960512000000000000
```

(!!!to be continued)

30.15 A Typical Session

```
  (12) -> )clear all
  (1) -> f = Frame()
  (2) -> f.setTitle("Hello world!")
  (3) -> f.setSize(200, 300)
  (4) -> f.setPosition(20, 20)
   2 +++ f.setPosition(20, 20)
     +++   ^^^^^^^^^^^
     +++ Error: The method 'setPosition(byte,byte)' cannot be found in
```

```
class 'java.awt.Frame' or a superclass
(5) -> f.setLocation(20, 20)
(6) -> f.setVisible(1)
(7) -> l = Label('Hi there')
(8) -> say f.getLayout
java.awt.BorderLayout[hgap=0,vgap=0]
(9) -> f.add(l, BorderLayout.CENTER)
(10) -> f.doLayout
(11) ->
(12) -> l.setForeground(Color.red)
(13) -> f.dispose
(14) -> )quit
```

30.16 Running Pipelines

When an input is not a NetRexx clause, or prefixed by an ')' (and it is a system command, see next section) the only allowed command is 'pipe'. This enables us to run a pipeline exactly as one would do in z/VM CMS. The built-in NetRexx Pipelines component is used to execute a pipeline like one can do in the command shell of the operating system, but with quotes. More about Pipelines can be found in the *Pipelines Guide and Reference*. If you are used to running pipelines on CMS, you can just go ahead and try a few things.

30.17 System Commands

We conclude our tour of Workspace for NetRexx with a brief discussion of system commands. System commands are special statements that start with a closing parenthesis (")"). They are used to control or display your Workspace for NetRexx environment, start operating system commands and leave Workspace for NetRexx. For example, `)system` is used to issue commands to the operating system from Workspace for NetRexx. Here is a brief description of some of these commands.

Perhaps the most important user command is the `)clear all` command that initializes your environment. Every section and subsection in this document has an invisible)`clear all` that is read prior to the examples given in the section. `)clear` all gives you a fresh, empty environment with no user variables defined and the step number reset to 1. The `)clear` command can also be used to selectively clear values and properties of system variables.

Another useful system command is `)read`. A preferred way to develop an application in Workspace for NetRexx is to put your interactive commands into a file, say my.input file. To get Workspace for NetRexx to read this file, you use the system command `)read my.input`. If you need to make changes to your approach or definitions, go into your favorite editor, change my.input, then issue tge command again.

Other system commands include: `)history`, to display previous input lines; `)display`, to display properties and values of workspace variables; and `)what`.

This conclude your tour of Workspace for NetREXX. To disembark, issue the system command `)quit` to leave Workspace for NetREXX and return to the operating system.

30.18 Input Files and NetREXX Files

This section discusses how to collect Workspace for NetREXX statements and commands into files and then read the contents into the workspace. I also discuss NetREXX files, which are a variation of input files.

30.19 Input Files

In this section it is explained what an input file is and why you would want to know about it. It is shown where Workspace for NetREXX looks for input files and how you can direct it to look elsewhere, and also how to read the contents of an input file into the workspace and how to use the history facility to generate an input file from the statements you have entered directly into the workspace.

An input file contains NetREXX expressions and system commands. Anything that you can enter directly to Workspace for NetREXX can be put into an input file. This is how input functions and expressions can be saved that you wish to read into Workspace for NetREXX more than one time.

To read an input file into Workspace for NetREXX, use the)read system command. For example, you can read a file in a particular directory by issuing

```
)read /nrws/src/input/matrix.input
```

The ".input" is optional; this also works:

```
)read /nrws/src/input/matrix
```

What happens if you just enter)read matrix.input or even)read matrix? Workspace for NetREXX looks in your current working directory for input files that are not qualified by a directory name. Typically, this directory is the directory from which you invoked Workspace for NetREXX. To change the current working directory, use the)cd system command. The command)cd by itself shows the current working directory. To change it to the src/input subdirectory for user "bar", issue

```
)cd /user/bar/src/input
```

Workspace for NetREXX looks first in this directory for an input file. If it is not found, it looks in the system's directories, assuming you meant some input file that was provided with Workspace for NetREXX.

If you have the Workspace for NetREXX history facility turned on (which it is by default), you can save all the lines you have entered into the workspace by entering

```
)history )write
```

Workspace for NetREXX tells you what input file to edit to see your statements. The file is in your home directory or in the directory you specified with)cd.

30.20 The workspace.input File

When Workspace for NetREXX starts up, it tries to read the input file workspace.input from your home directory. If there is no workspace.input in your home directory, it reads the copy located in its own src/input directory. The file usually contains system commands to personalize your Workspace for NetREXX environment. In the remainder of this section I mention a few things that users frequently place in their workspace.input files.

If you do not want to be prompted for confirmation when you issue the)quit system command, place)set quit unprotected in workspace.input. If you then decide that you do want to be prompted, issue)set quit protected. This is the default setting so that new users do not leave Workspace for NetREXX inadvertently.

To see the other system variables you can set, issue)set.

30.21 The nrws.properties File

In this file, that is looked for in the home directory, a few parameters can be specified. For example,

```
settings.prompt=nrws>
settings.timer=on
settings.quit=unprotected
```

indicates that the prompt will be *nrws*>, and the right side of the screen shows the command exection time instead of the frame name. Further more, the)quit system command (see next) quits immediately instead of prompting.

30.22 The nrws.history file(s)

For easy command history retrieval (using the arrow keys) the Workspace for NetREXX stores executed commands in a nrws.history file in the current directory. This is buy design not a user global file, but is written to (and read from) the current directory because it is plausible that different projects call for different command history.

30.23 Workspace for NetRexx System Commands

This chapter describes system commands, the command-line facilities used to control the Workspace for NetRexx environment. The first section is an introduction and discusses the common syntax of the commands available.

30.24 Introduction

System commands are used to perform Workspace for NetRexx environment management. Among the commands are those that display what has been defined or computed, set up multiple logical Workspace for NetRexx environments (frames), clear definitions, read files of expressions and command, show what functions are available, and terminate Workspace for NetRexx.

Each command listing begins with one or more syntax pattern descriptions plus examples of related commands. The syntax descriptions are intended to be easy to read and do not necessarily represents the most compact way of specifying all possible arguments and options; the descriptions may occasionally be redundant.

All system commands begin with a right parenthesis which should be in the first available column of the input line (that is, immediately after the input prompt, if any). System commands may be issued directly to Workspace for NetRexx or be included in .input files.

A system command argument is a word that directly follows the command name and is not followed or preceded by a right parenthesis. A system command option follows the command and is directly preceded by a right parenthesis. Options may have arguments: they directly follow the option. This example may make it easier to remember what is an option and what is an argument:

```
)syscmd arg1 arg2 )opt1 opt1arg1 opt2arg2 )opt2 opt2arg1 ...
```

In the system command descriptions, optional arguments and options are enclosed in brackets ("[" and "]"). If an argument or option name is in italics, it is meant to be a variable and must have some actual value substituted for it when the system command call is made. For example, the syntax pattern description

```
)read fileName [)quietly]
```

would imply that you must provide an actual file name for fileName but need not to use the)quietly option. Thus

```
)read foo.input
```

is a valid instance of the above pattern.

System commands names and options may be abbreviated and may be in upper or lower case. The case of actual arguments may be significant, depending on the particular situation (such as in file names). System command names and options may be abbreviated to the minimum number of starting letters so that the name or option is unique. Thus

```
)s Integer
```

is not a valid abbreviation for the)set command, because both)set and)show begin with the letter ”s”. Typically, two or three letters are sufficient for disambiguating names. In my descriptions of the commands, I have used no abbreviations for either command names or options.

In some syntax descriptions I use a vertical line ”|” to indicate that you must specify one of the listed choices. For example, in

```
)set foobar on | off
```

only on and off are acceptable words for following foobar. I also sometimes use ”...” to indicate that additional arguments or options of the listed form are allowed. Finally, in the syntax descriptions I may also list the syntax of related commands.

30.25)cd

Command Syntax:

```
  )cd
  )cd directory
```

Command Description:

This command sets the Workspace for NetRexx working directory. The current directory is used for looking for input files (for)read) and for writing history input files (for)history)write).

If used with no argument, this command shows the current working directory. If an argument is used, it must be a valid directory name. Except for the ”)” at the beginning of the command, this has the same syntax as the operating system cd command.

Also See: ’)history’, and ’)read’.

30.26)clear

Command Syntax:

```
  )clear all
  )clear properties all
  )clear properties obj1 [obj2 ...]
```

Command Description:

This command is used to remove functions and variable declarations, definitions and values from the workspace. To empty the entire workspace and reset the step counter to 1, issue

```
)clear all
```

To remove everything in the workspace but not reset the step counter, issue

```
)clear properties all
```

To remove everything about the object x, issue

```
)clear properties x
```

To remove everything about the objects x, y and f, issue

```
)clear properties x y f
```

The word properties may be abbreviated to the single letter ”p”.

```
)clear p all
)clear p x
)clear p x y f
```

The)display names and)display properties commands may be used to see what is currently in the workspace.
Also See: ’)display’, ’)history’.

30.27)display

Command Syntax:

```
  )display all
  )display properties
  )display properties all
  )display properties [obj1 [obj2 ...]]
  )display type all
  )display type [obj1 [obj2 ...]]
  )display names
```

Command Description:
This command is used to display the contents of the workspace and signatures of functions with a given name.
The command

```
)display names
```

list the names of all user-defined objects in the workspace. This is useful if you do not wish to see everything about the objects and need only be reminded of their names.
The commands

```
)display all
)display properties
)display properties all
```

all do the same thing: show the values and types of all variables in the workspace. If you have defined functions, their signatures and definitions will also be displayed.

To show all information about a particular variable or user functions, for example, something named d, issue

```
)display properties d
```

The word properties may be abbreviated to the single letter ”p”.

```
)display p all
)display p
)display p d
```

To just show the declared type of d, issue

```
)display type d
)display t d
```

Also See: ’)clear’, ’)history’, ’)set’, ’)show’, ’)what’.

30.28)frame

Command Syntax:

```
  )frame new frameName
  )frame drop [frameName]
  )frame next
  )frame last
  )frame names
  )frame import frameName [objectName1 [objectName2 ...]]
  )set message prompt frame
```

Command Description:

A frame can be thought of as a logical session within the physical session that you get when you start the system. You can have as many frames as you want, within the limits of your computer’s storage, paging space, and so on. Each frame has its own step number, environment and history. You can have a variable named a in one frame and it will have nothing to do with anything that might be called a in any other frame.

To find out the names of all frames, issue

```
)frame names
```

It will indicate the name of the current frame.

You can create a new frame ”quark” by issuing

```
)frame new quark
```

If you wish to go back to what you were doing in the ”initial” frame, use

```
)frame next
```

or

```
)frame last
```

to cycle through the ring of available frames to get back to ”initial”.

If you want to throw away a frame (say ”quark”), issue

```
)frame drop quark
```

If you omit the name, the current frame is dropped.

You can bring things from another frame by using)frame import. For example, to bring the f and g from the frame ”quark” to the current frame, issue

```
)frame import quark f g
```

If you want everything from the frame ”quark”, issue

```
)frame import quark
```

You will be asked to verify that you really want everything.

There is one)set flag to make it easier to tell were you are.

```
)set message prompt frame
```

will give a prompt that looks like

```
initial (1) -> _
```

when you start up. In this case, the frame name and step make up the prompt.

Also See: ’)history’, ’)set’

30.29)help

Command Syntax:

```
  )help
  )help commandName
```

Command Description:

This command displays help information about system commands. If you issue

```
)help
```

a list of possible commands will be shown. You can also give the name or abbreviation of a system command to display information about it. For example,

```
)help clear
```

will display the description of the)clear system command.

30.30)history

Command Syntax:

```
  )history )on
  )history )off
  )history )show [n]
  )history )write historyInputFileName
  )set history on | off
  )set history write protected | unprotected
```

Command Description:

The history facility within Workspace for NetREXX allows you to restore your environment to that of another session and recall previous computational results. Additional commands allow you to create an .input file of the lines typed to Workspace for NetREXX.

Workspace for NetREXX saves your input if the history facility is turned on (which is the default). This information is saved if either of

```
)set history on
)history )on
```

has been issued. Issuing either

```
)set history off
)history )off
```

will discontinue the recording of information.

Each frame has its own history database.

The options to the)history commands are as follows:

```
)on
```

will start the recording of information. If the workspace is not empty, you will be asked to confirm this request. If you do so, the workspace will be cleared and history data will begin being saved. You can also turn the facility on by issuing)set history on.

```
)off
```

will stop the recording of information. The)history)show command will not work after issuing this command. Note that this command may be issued to save time, as there is some performance penalty paid for saving the environment data. You can also turn the facility off by issuing)set history off.

```
)show [n]
```

can show previous input lines.)show will display up to twenty of the last input lines (fewer if you haven't typed in twenty lines).)show n will display up to n of the last input lines.)write historyInputFile creates an .input file with the input typed since the start of the session/frame or the last)clear all. If historyInput-

File does not contain a period (”.”) in the filename, .input is appended to it. For example,)history)write chaos and)history)write chaos.input both write the input lines to a file called chaos.input in your current working directory. You can edit this file and then use)read to have Workspace for NetRexx process the contents. Also See: ’)frame’, ’)read’, ’)set’.

30.31)import

Command Syntax:

```
  )import query
  )import package packageName
  )import class fullClassName
  )import drop packageOrFullClassName
```

Command Description:

This command is used to query, set and remove imported packages.

When used with the query argument, this command may be used to list the names of all imported packages and classes.

The following command lists all imported packages and classes.

```
)import query
```

To remove an imported package or class, the remove argument is used. This is usually only used to correct a previous command that imported a package or a class. If, in fact, the imported package or class does exist, you are prompted for confirmation of the removal request. The following command will remove the imported package com.foo.bar from the system:

```
)import drop com.foo.bar
```

Also See: ’)set’

30.32)numeric

Command Syntax:

```
  )numeric
  )numeric digits number
  )numeric form scientific | engineering
  )set numeric digits number
  )set numeric form scientific | engineering
```

Command Description:

(!!! just like the numeric instruction)

30.33)options

Command Syntax:

```
  )options
  )options )default
  )options option [)off]
  )set option option on | off
```

Command Description:

This command is used to specify the options in use while interpreting statements.

To list all active options, simply issue

)options To restore options to their defaults settings, issue

```
)options )default
```

The possible value for option are

```
binary
decimal
explicit
strictargs
strictassign
strictcase
strictsignal
default :

nobinary
decimal
noexplicit
nostrictargs
nostrictassign
nostrictcase
nostrictsignal
```

Also See: ')set'

30.34)package

Command Syntax:

```
  )package
  )package )default
  )package packageName
  )set package default | packageName
```

Command Description:

(!!! just like the package instruction)

30.35)pquit

Command Syntax:

```
  )pquit
```

Command Description:

This command is used to terminate Workspace for NetREXX and return to the operating system. Other than by redoing all your computations, you cannot return to Workspace for NetREXX in the same state.

)pquit differs from the)quit in that it always asks for confirmation that you want to terminate Workspace for NetREXX (the ”p” is for ”protected”). When you enter the)quit command, Workspace for NetREXX responds

Please enter ”y” or ”yes” if you really want to leave the interactive environment and return to the operating system. If you respond with y or yes, Workspace for NetREXX will terminate and return you to the operating system (or the environment from which you invoked the system). If you responded with something other that y or yes, then Workspace for NetREXX would still be running.

Also See: ’)history’, ’)quit’, ’)system’.

30.36)quit

Command Syntax:

```
  )quit
  )set quit protected | unprotected
```

Command Description:

This command is used to terminate Workspace for NetREXX and return to the operating system. Other than by redoing all your computations, you cannot return to Workspace for NetREXX in the same state.

)quit differs from the)pquit in that it asks for confirmation only if the command

```
)set quit protected
```

has been issued. Otherwise,)quit will make Workspace for NetREXX terminate and return you to the operating system (or the environment from which you invoked the system).

The default setting is)set quit protected so that)quit and)pquit behave the same way. If you do issue

```
)set quit unprotected
```

I suggest that you do not (somehow) assign)quit to be executed when you press, say, a function key.

Also See: ')history', ')pquit', ')system'.

30.37)read

Command Syntax:

```
  )read [fileName]
  )read [fileName] [)quiet] [)ifthere]
```

Command Description:

This command is used to read .input files into Workspace for NetREXX. The command

```
)read matrix.input
```

will read the contents of the file matrix.input into Workspace for NetREXX. The ".input" file extension is optional. See Section 3.1 for more information about .input files.

This command remembers the previous file you read. If you do not specify a file name, the previous file will be read.

The)ifthere option checks to see whether the .input file exists. If it does not, the)read command does nothing. If you do not use this option and the file does not exist, you are asked to give the name of an existing .input file.

The)quiet option suppresses output while the file is being read.

Also See: ')history'

30.38)set

Command Syntax:

```
  )set
  )set label1 [... labelN]
  )set label1 [... labelN] newValue
```

Command Description:

The)set command is used to view or set system variables that control what messages are displayed, the type of output desired, the status of the history facility, and so on.

The following arguments are possible:

```
)set diag on | off
```

enables or disables verbose reporting of some run-time errors. (Used for debugging purpose.)

```
)set display depth depth
```

specify the maximum number of elements to display when showing an array. (Default value is 10.)

```
)set display depth
```

show the current display depth.

```
)set display level number
```

specify the maximum number of nested arrays to display when showing an array. (Default value is 4.)

```
)set display level
```

show the current display level.

```
)set history write protected | unprotected
```

specify whether or not to prompt for confirmation when attempting to overwrite an existing file with)history)write.

```
)set history on | off
```

enables or disables history.

```
)set import add class className
)set import add package packageName
)set import drop class className
)set import drop package packageName
```

adds or removes specified class or package from import list.

```
)set import
```

shows the currently imported statements.

```
)set interpreter on | off
```

set the interpreter status. If on, then valid statements will be executed. If off, then no execution will be attempted. (Mostly used for debugging purpose, or if you want to use Workspace for NetRexx on a pre-java2 platform.)

```
)set message prompt default
)set message prompt frame
)set message prompt label label
```

set the prompt status (frame displays the current frame name).

```
)set message prompt
```

shows the current prompt status.

```
)set numeric digits number
```

set the default numeric digits (i.e., for the current frame and all subsequent frames).

```
)set numeric digits
```

shows the current default numeric digits value.

```
)set numeric form scientific | engineering
```

set the default numeric form (i.e., for the current frame and all subsequent frames).

```
)set numeric form
```

shows the current default numeric form.

```
)set option option on | off
```

set the default activity of option option (i.e., for the current frame and all subsequent frames). option being one of : binary, decimal, explicit, strictargs, strictassign, strictcase, or strictsignal.

```
)set option option
```

shows the current option status.

```
)set package default
)set package packageName
```

set the current package name.

```
)set package
```

shows the current package name.

```
)set parser quiet | verbose
```

disables or enables verbose output from the parser. (Used for debugging purposes.)

```
)set quit protected | unprotected
```

set the quit status.

```
)set quit
```

shows the current quit status.

```
)set screen width number
```

set the screen width (in character).

```
)set screen width
```

shows the screen width.

```
)set show all | declared
```

set the amount of information displayed by the)show command.

```
)set show
```

shows the current show status.

```
)set trace
)set trace all | off | methods | results
```

set the default trace level (i.e., for the current frame and all subsequent frames).

```
)set use add className
)set use drop className
```

adds or removes specified class name from use list.

```
)set use
```

shows the current use list. Also See: ')quit', ')show'

30.39)show

Command Syntax:

```
  )show nameOrAbbrev
  )show nameOrAbbrev )operations
  )show nameOrAbbrev )attributes
  )set show all | declared
```

Command Description:

This commands displays information about classes. If no options are given, the)operations option is assumed. For example,

```
)show Rectangle
)show Rectangle )operations
)show java.awt.Rectangle
)show java.awt.Rectangle )operations
```

each display basic information about the java.awt.Rectangle class constructors and then provide a listing of operations.

The basic information displayed includes the signature of the constructors and the operations.

Also See: ')display', ')set'

30.40)synonym

Command Syntax:

```
  )synonym
  )synonym synonym fullCommand
  )what synonyms
```

Command Description:

This command is used to create short synonyms for system command expressions. For example, the following synonyms might simplify commands you often use.

```
)synonym prompt     set message prompt
)synonym mail       system mail
)synonym ls         system ls
```

Once defined, synonyms can be used in place of the longer command expressions. Thus

```
)prompt frame
```

is the same as the longer

```
)set message prompt frame
```

To list all defined synonyms, issue either of

```
)synonym
)what synonym
```

To list, say, all synonyms that contain the substring ”ap”, issue

```
)what synonym ap
```

Also See: ’)set’, ’what’

30.41)system

Command Syntax:

```
)system cmdExpression
```

Command Description:

This command may be used to issue commands to the operating system while remaining in Workspace for NetRexx. The cmdExpression is passed to the operating system for execution.

If you execute programs that misbehave you may not be able to return to Workspace for NetRexx. If this happens, you may have no other choice than to restart Workspace for NetRexx and restore the environment via)history)restore, if possible.

Also See: ’)pquit’, ’)quit’

30.42)trace

Command Syntax:

```
  )trace
  )trace off
  )trace all
  )trace methods
  )trace results
  )trace var [var1 [var2 ...]]
```

Command Description:

This command is used to trace the execution of statements and functions defined by users.

To list all currently enabled trace functions, simply issue

```
)trace
```

To untrace everything that is traced, issue

```
)trace off
```

(!!! to be continued, just like the trace instruction)

30.43)use

Command Syntax:

```
  )use query
  )use add className
  )use drop className
```

Command Description:

(!!! like the uses phrase in class instruction)

30.44)what

Command Syntax:

```
  )what commands pattern1 [pattern2 ...]
  )what synonym pattern1 [pattern2 ...]
  )what things pattern1 [pattern2 ...]
  )apropos pattern1 [pattern2 ...]
```

Command Description:

This command is used to display lists of things in the system. The patterns are all strings and, if present, restrict the contents of the lists. Only those items that contain one or more of the strings as substrings are displayed. For example,

```
)what synonyms
```

displays all command synonyms,

```
)what synonyms ver
```

displays all command synonyms containing the substring ”ver”,

```
)what synonyms ver pr
```

displays all command synonyms containing the substring ”ver” or the substring ”pr”. Output similar to the following will be displayed

——————— System Command Synonyms ———————

user-defined synonyms satisfying patterns: ver pr

```
  )apr ............................ )what things
  )apropos ........................ )what things
  )prompt ......................... )set message prompt
```

Several other things can be listed with the)what command:

commands displays a list of system commands available. To get a description of a particular command, such as ”)what”, issue)help what. synonyms lists system command synonyms. things displays all of the above types for items containing the pattern strings as substrings. The command synonym)apropos is equivalent to)what things. Also See: ’)display’, ’)set’, and ’)show’

31

Translator inner workings

This chapter includes all documentation on the inner workings of the translator that is available. Its purpose is to assist with debugging serious problems or ease the introduction to the toolset for programmers who want to help the open source effort forwards.

31.1 Translator source files

The translator source is part of the package `org.netrexx.process`. The runtime support, including the `Rexx` type, is in the package `netrexx.lang`.

The source files in table 3 all correspond to a specific NetRexx clause, all created by RxParser, and all implementing RxClauseParser. Each is responsible for syntax checking, semantic processing, and code generation for the corresponding clause. RxClass and RxMethod are the critical classes. RxNop is the simplest. Method-term instructions are currently handled in RxParser but should have a separate class in this list.

TABLE 1: Translator source files

NetRexxC.nrx	The 'main program'
nrc.prp	Error messages (becomes NetRexxC.properties resource bundle)
RxArray.nrx	Parsed array reference
RxClasser.nrx	The class 'factory'; finds classes and packages, loads classes, finds fields in packages, etc.
RxClassImage.nrx	Loads and parses a .class file (from zip or directory byte stream)
RxClassInfo.nrx	Known information about a class
RxClassPool.nrx	Collection of known classes (maintained by RxClasser)
RxClause.nrx	The tokens and object corresponding to a clause
RxClauseParser.nrx	Interface: all clause objects implement this
RxClauser.nrx	Tokenizer (lexical analysis/parse)
RxCode.nrx	Represents encoded piece of program (e.g., an expression or clause). Holds information about the source of the code, and the code itself (currently only Java source code). At present, RxCode is only used for terms and expressions; clauses will probably evolve to use RxCode objects too.
RxConvert.nrx	Holds the cost and type of a conversion
RxConverter.nrx	Determines and costs a conversion/coercion, and effects a particular conversion
RxError.nrx	Handle an Error (see also RxQuit and RxWarn)
RxException.nrx	Represents a Java exception
RxExprParser.nrx	Parse and generate RxCode for an expression
RxField.nrx	Represents a field (property or method)
RxFixup.nrx	Changes the sourcefile attribute in a .class file to point to Foo.nrx constant instead of Foo.java
RxFlag.nrx	Represents option flags
RxLanguage.nrx	Language version and date, and major change list
RxLevel.nrx	Represents a level of semantic nesting. 0=class, 1=method, 2 is method body (do groups, etc.)
RxMessage.nrx	Displays/queues an error or warning message. (Offspring of RxError, RxQuit, RxWarn)
RxPackageInfo.nrx	Describes a known package
RxParser.nrx	NetRexx-specific program/clause parser
RxProgram.nrx	Represents a compilation unit (==Program)
RxQuit.nrx	Handles severe errors (see also RxError, RxWarn)
RxSignature.nrx	Represents a type
RxStreamer.nrx	Handles input and output streams (files), including formatting of output Java source
RxTermParser.nrx	Parses terms in expressions
RxToken.nrx	Represents a lexical token (see RxClauser)
RxTracer.nrx	Generates code for tracing of various types
RxTranslator.nrx	'top-level' controller for parsing and compilation.

TABLE 2: Translator source files -2

RxVariable.nrx	Represents a local or class variable, and its cross-reference list
RxVarpool.nrx	Collection of known RxVariables
RxWarn.nrx	Handles Warnings
RxChunk.nrx	A chunk of Java sourcecode, destined for the output file (planned to be replaced by RxCode objects, long term)

TABLE 3: Translator source files -3

RxAssign.nrx	handles all assignment clauses
RxCatch.nrx	
RxClass.nrx	
RxDo.nrx	
RxElse.nrx	
RxEnd.nrx	
RxExit.nrx	
RxFinally.nrx	
RxIf.nrx	
RxImport.nrx	
RxIterate.nrx	
RxLeave.nrx	
RxLoop.nrx	
RxMethod.nrx	
RxNop.nrx	
RxNumeric.nrx	
RxOptions.nrx	
RxOtherwise.nrx	
RxPackage.nrx	
RxParse.nrx	
RxProperties.nrx	
RxReturn.nrx	
RxSay.nrx	
RxSelect.nrx	
RxSignal.nrx	
RxThen.nrx	
RxTrace.nrx	
RxWhen.nrx	

31.2 Method resolution

Method resolution in NetRexx proceeds as follows:

- If the method invocation is the first part (stub) of a term, then:
  1. The current class is searched for the method (see below for details of searching).
  2. If not found in the current class, then the superclasses of the current class are searched, starting with the class that the current class extends.
  3. If still not found, then the classes listed in the uses phrase of the class instruction are searched for the method, which in this case must be a static method. Each class from the list is searched for the method, and then its superclasses are searched upwards from the class; this process is repeated for each of the classes, in the order specified in the list.
  4. If still not found, the method invocation must be a constructor (see below) and so the method name, which may be qualified by a package name, should match the name of a primitive type or a known class (type). The specified class is then searched for a constructor that matches the method invocation.
- If the method invocation is not the first part of the term, then the evaluation of the parts of the term to the left of the method invocation will have resulted in a value (or just a type), which will have a known type (the continuation type). Then:
  1. The class that defines the continuation type is searched for the method (see below for details of searching).
  2. If not found in that class, then the superclasses of that class are searched, starting with the class that that class extends. If the search did not find a method, an error is reported. If the search did find a method, that is the method which is invoked, except in one case:
  3. If the evaluation so far has resulted in a value (an object), then that value may have a type which is a subclass of the continuation type. If, within that subclass, there is a method that exactly overrides the method that was found in the search, then the method in the subclass is invoked.

This case occurs when an object is earlier assigned to a variable of a type which is a superclass of the type of the object. This type simplification hides the real type of the object from the language processor, though it can be determined when the program is executed. Searching for a method in a class proceeds as follows:

1. Candidate methods in the class are selected. To be a candidate method:
   - the method must have the same name as the method invocation (independent of the case (see page 44) of the letters of the name)
   - the method must have the same number of arguments as the method invocation (or more arguments, provided that the remainder are shown as optional in the method definition)
   - it must be possible to assign the result of each argument expression to the type of the corresponding argument in the method definition (if

strict type checking is in effect, the types must match exactly).

2. If there are no candidate methods then the search is complete; the method was not found.
3. If there is just one candidate method, that method is used; the search is complete.
4. If there is more than one candidate method, the sum of the costs of the conversions from the type of each argument expression to the type of the corresponding argument defined for the method is computed for each candidate method.
5. The costs of those candidates (if any) whose names match the method invocation exactly, including in case, are compared; if one has a lower cost than all others, that method is used and the search is complete.
6. The costs of all the candidates are compared; if one has a lower cost than all others, that method is used and the search is complete.
7. If there remain two or more candidates with the same minimum cost, the method invocation is ambiguous, and an error is reported. Note: When a method is found in a class, superclasses of that class are not searched for methods, even though a lower-cost method may exist in a superclass.

Note that until version 3.01 of the NetRexx translator a slightly different way of method resolution was used. There is a very small (and almost improbable) chance of encountering differences when recompiling very old sources.

Index

ISBN 978-90-819090-0-6

9 789081 909006 >